scikits.learn user guide

Release 0.7

scikits.learn developers

March 02, 2011

CONTENTS

`scikits.learn` is a Python module integrating classic machine learning algorithms in the tightly-knit world of scientific Python packages (numpy, scipy, matplotlib).
It aims to provide simple and efficient solutions to learning problems that are accessible to everybody and reusable in various contexts: **machine-learning as a versatile tool for science and engineering**.

Features

- **Solid**: *Supervised learning*: *Support Vector Machines*, *Generalized Linear Models*.
- **Work in progress**: *Unsupervised learning*: *Clustering*, *Gaussian mixture models*, manifold learning, *ICA*, *Gaussian Processes*
- **Planed**: Gaussian graphical models, matrix factorization

License Open source, commercially usable: **BSD license** (3 clause)

Note: This document describes scikits.learn 0.7. For other versions and printable format, see *documentation_resources*.

CHAPTER

ONE

USER GUIDE

1.1 Installing *scikits.learn*

There are different ways to get scikits.learn installed:

- Install the version of scikits.learn provided by your *operating system distribution* . This is the quickest option for those who have operating systems that distribute scikits.learn.
- *Install an official release*. This is the best approach for users who want a stable version number and aren't concerned about running a slightly older version of scikits.learn.
- *Install the latest development version*. This is best for users who want the latest-and-greatest features and aren't afraid of running brand-new code.

1.1.1 Installing an official release

Installing from source

Installing from source requires you to have installed numpy, scipy, setuptools, python development headers and a working C++ compiler. Under debian-like systems you can get all this by executing with root privileges:

```
sudo apt-get install python-dev python-numpy python-setuptools python-scipy libatlas-dev g++
```

Easy install

This is usually the fastest way to install the latest stable release. If you have pip or easy_install, you can install or update with the command:

```
pip install -U scikits.learn
```

or:

```
easy_install -U scikits.learn
```

for easy_install. Note that you might need root privileges to run these commands.

From source package

Download the package from http://sourceforge.net/projects/scikit-learn/files , unpack the sources and cd into archive.

This packages uses distutils, which is the default way of installing python modules. The install command is:

```
python setup.py install
```

Windows installer

You can download a windows installer from downloads in the project's web page. Note that must also have installed the packages numpy and setuptools.

This package is also expected to work with python(x,y) as of 2.6.5.5.

1.1.2 Third party distributions of scikits.learn

Some third-party distributions are now providing versions of scikits.learn integrated with their package-management systems.

These can make installation and upgrading much easier for users since the integration includes the ability to automatically install dependencies (numpy, scipy) that scikits.learn requires.

The following is a list of linux distributions that provide their own version of scikits.learn:

Debian and derivatives (Ubuntu)

The Debian package is named python-scikits-learn and can be install using the following commands with root privileges:

```
apt-get install python-scikits-learn
```

Enthought python distribution

The Enthought Python Distribution already ships the latest version.

Macports

The macport's package is named py26-scikits-learn and can be installed by typing the following command:

```
sudo port install py26-scikits-learn
```

NetBSD

scikits.learn is available via pkgsrc-wip:

http://pkgsrc.se/wip/py-scikits_learn

1.1.3 Bleeding Edge

See section *Retrieving the latest code* on how to get the development version.

1.1.4 Testing

Testing requires having the nose library. After installation, the package can be tested by executing from outside the source directory:

```
python -c "import scikits.learn as skl; skl.test()"
```

This should give you a lot of output (and some warnings) but eventually should finish with the a text similar to:

```
Ran 601 tests in 27.920s
OK (SKIP=2)
```

otherwise please consider submitting a bug in the *bug_tracker* or to the *mailing_lists*.

scikits.learn can also be tested without having the package installed. For this you must compile the sources inplace from the source directory:

```
python setup.py build_ext --inplace
```

Test can now be run using nosetest:

```
nosetests scikits/learn/
```

If you are running the deveopment version, this is automated in the commands *make in* and *make test*.

Warning: Because nosetest does not play well with multiprocessing on windows, this last approach is not recommended on such system.

1.2 Getting started: an introduction to machine learning with scikits.learn

Section contents

In this section, we introduce the machine learning vocabulary that we use through-out *scikits.learn* and give a simple learning example.

1.2.1 Machine learning: the problem setting

In general, a learning problem considers a set of n *samples* of data and try to predict properties of unknown data. If each sample is more than a single number, and for instance a multi-dimensional entry (aka *multivariate* data), is it said to have several attributes, or *features*.

We can separate learning problems in a few large categories:

- **supervised learning**, in which the data comes with additional attributes that we want to predict. This problem can be either:
  - **classification**: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data. An example of classification problem would be the digit recognition example, in which the aim is to assign each input vector to one of a finite number of discrete categories.
  - **regression: if the desired output consists of one or more** continuous variables, then the task is called *regression*. An example of a regression problem would be the prediction of the length of a salmon as a function of its age and weight.

- **unsupervised learning, in which the training data consists of a** set of input vectors x without any corresponding target values. The goal in such problems may be to discover groups of similar examples within the data, where it is called *clustering*, or to determine the distribution of data within the input space, known as *density estimation*, or to project the data from a high-dimensional space down to two or thee dimensions for the purpose of *visualization*.

Training set and testing set

Machine learning is about learning some properties of a data set and applying them to new data. This is why a common practice in machine learning to evaluate an algorithm is to split the data at hand in two sets, one that we call a *training set* on which we learn data properties, and one that we call a *testing set*, on which we test these properties.

1.2.2 Loading an example dataset

scikits.learn comes with a few standard datasets, for instance the iris dataset, or the digits dataset:

```
>>> from scikits.learn import datasets
>>> iris = datasets.load_iris()
>>> digits = datasets.load_digits()
```

A dataset is a dictionary-like object that holds all the data and some metadata about the data. This data is stored in the *.data* member, which is a *n_samples, n_features* array. In the case of supervised problem, explanatory variables are stored in the *.target* member.

For instance, in the case of the digits dataset, *digits.data* gives access to the features that can be used to classify the digits samples:

```
>>> print digits.data
[[  0.   0.   5. ...,   0.   0.   0.]
 [  0.   0.   0. ...,  10.   0.   0.]
 [  0.   0.   0. ...,  16.   9.   0.]
 ...,
 [  0.   0.   1. ...,   6.   0.   0.]
 [  0.   0.   2. ...,  12.   0.   0.]
 [  0.   0.  10. ...,  12.   1.   0.]]
```

and *digits.target* gives the ground truth for the digit dataset, that is the number corresponding to each digit image that we are trying to learn:

```
>>> digits.target
array([0, 1, 2, ..., 8, 9, 8])
```

Shape of the data arrays

The data is always a 2D array, *n_samples, n_features*, although the original data may have had a different shape. In the case of the digits, each original sample is an image of shape *8, 8* and can be accessed using:

```
>>> digits.images[0]
array([[  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.],
       [  0.,   0.,  13.,  15.,  10.,  15.,   5.,   0.],
       [  0.,   3.,  15.,   2.,   0.,  11.,   8.,   0.],
       [  0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.],
       [  0.,   5.,   8.,   0.,   0.,   9.,   8.,   0.],
       [  0.,   4.,  11.,   0.,   1.,  12.,   7.,   0.],
       [  0.,   2.,  14.,   5.,  10.,  12.,   0.,   0.],
       [  0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.]])
```

The *simple example on this dataset* illustrates how starting from the original problem one can shape the data for consumption in the *scikit.learn*.

`scikits.learn` also offers the possibility to reuse external datasets coming from the http://mlcomp.org online service that provides a repository of public datasets for various tasks (binary & multi label classification, regression, document classification, ...) along with a runtime environment to compare program performance on those datasets. Please refer to the following example for for instructions on the `mlcomp` dataset loader: *example_mlcomp_document_classification.py*.

1.2.3 Learning and Predicting

In the case of the digits dataset, the task is to predict the value of a hand-written digit from an image. We are given samples of each of the 10 possible classes on which we *fit* an *estimator* to be able to *predict* the labels corresponding to new data.

In *scikit.learn*, an *estimator* is just a plain Python class that implements the methods *fit(X, Y)* and *predict(T)*.

An example of estimator is the class `scikits.learn.svm.SVC` that implements Support Vector Classification. The constructor of an estimator takes as arguments the parameters of the model, but for the time being, we will consider the estimator as a black box and not worry about these:

```
>>> from scikits.learn import svm
>>> clf = svm.SVC()
```

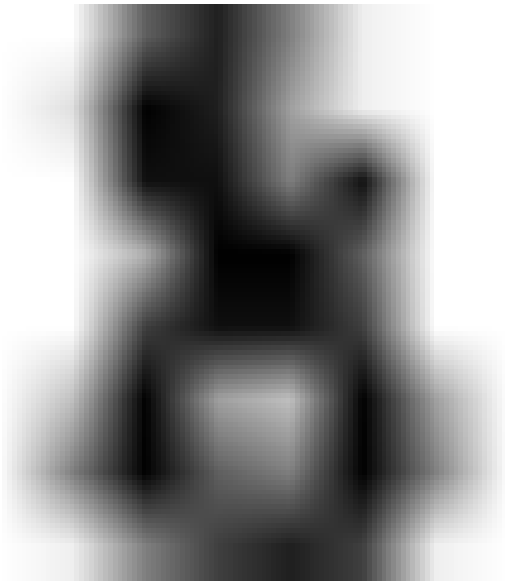
We call our estimator instance *clf* as it is a classifier. It now must be fitted to the model, that is, it must *learn* from the model. This is done by passing our training set to the `fit` method. As a training set, let us use all the images of our dataset apart from the last one:

```
>>> clf.fit(digits.data[:-1], digits.target[:-1])
SVC(kernel='rbf', C=1.0, probability=False, degree=3, coef0=0.0, eps=0.001,
  cache_size=100.0, shrinking=True, gamma=0.000556792873051)
```

Now you can predict new values, in particular, we can ask to the classifier what is the digit of our last image in the *digits* dataset, which we have not used to train the classifier:

```
>>> clf.predict(digits.data[-1])
array([ 8.])
```

The corresponding image is the following:



As you can see, it is a challenging task: the images are of poor resolution. Do you agree with the classifier?

A complete example of this classification problem is available as an example that you can run and study: *Recognizing hand-written digits*.

1.2.4 Model persistence

It is possible to save a model in the scikit by using Python's built-in persistence model, namely pickle.

```
>>> from scikits.learn import svm
>>> from scikits.learn import datasets
>>> clf = svm.SVC()
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf.fit(X, y)
SVC(kernel='rbf', C=1.0, probability=False, degree=3, coef0=0.0, eps=0.001,
  cache_size=100.0, shrinking=True, gamma=0.00666666666667)
>>> import pickle
>>> s = pickle.dumps(clf)
>>> clf2 = pickle.loads(s)
>>> clf2.predict(X[0])
array([ 0.])
>>> y[0]
0
```

In the specific case of the scikit, it may be more interesting to use joblib's replacement of pickle, which is more efficient on big data, but can only pickle to the disk and not to a string:

```
>>> from scikits.learn.externals import joblib
>>> joblib.dump(clf, 'filename.pkl')
```

1.3 Supervised learning

1.3.1 Generalized Linear Models

The following are a set of methods intended for regression in which the target value is expected to be a linear combination of the input variables. In mathematical notion, if $\hat{y}$ is the predicted value.

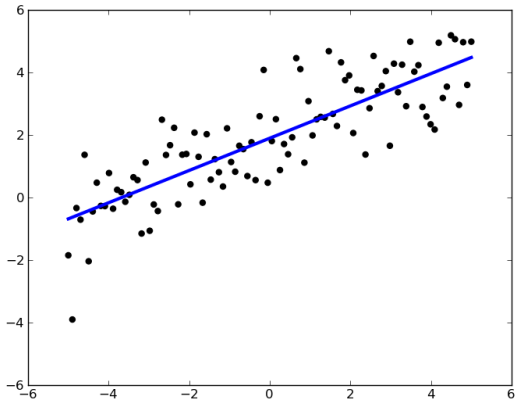
$$\hat{y}(\beta, x) = \beta_0 + \beta_1 x_1 + ... + \beta_D x_D$$

Across the module, we designate the vector $\beta = (\beta_1, ..., \beta_D)$ as `coef_` and β_0 as `intercept_`.

To perform classification with generalized linear models, see *Logistic regression*.

Ordinary Least Squares (OLS)

`LinearRegression` fits a linear model with coefficients $\beta = (\beta_1, ..., \beta_D)$ to minimize the residual sum of squares between the observed responses in the dataset, and the responses predicted by the linear approximation.



`LinearRegression` will take in its *fit* method arrays X, y and will store the coefficients w of the linear model in its *coef_* member.

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.LinearRegression()
>>> clf.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
LinearRegression(fit_intercept=True)
>>> clf.coef_
array([ 0.5,  0.5])
```

However, coefficient estimates for Ordinary Least Squares rely on the independence of the model terms. When terms are correlated and the columns of the design matrix X have an approximate linear dependence, the matrix $X(X^T X)^{-1}$ becomes close to singular and as a result, the least-squares estimate becomes highly sensitive to random errors in the observed response, producing a large variance. This situation of *multicollinearity* can arise, for example, when data are collected without an experimental design.

Examples:

- *Ordinary Least Squares*

OLS Complexity

This method computes the least squares solution using a singular value decomposition of X. If X is a matrix of size (n, p) this method has a cost of $O(np^2)$, assuming that $n \geq p$.

Ridge Regression

`Ridge` regression addresses some of the problems of *Ordinary Least Squares (OLS)* by imposing a penalty on the size of coefficients. The ridge coefficients minimize a penalized residual sum of squares,

$$\beta^{ridge} = \underset{\beta}{argmin} \sum_{i=1} N(y_i - \beta_0 - \sum_{j=1} px_ij\beta_j)^2 + \alpha \sum_{j=1} p\beta_j^2$$

Here, $\alpha \geq 0$ is a complexity parameter that controls the amount of shrinkage: the larger the value of α, the greater the amount of shrinkage.

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.Ridge (alpha = .5)
>>> clf.fit ([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
Ridge(alpha=0.5, fit_intercept=True)
>>> clf.coef_
array([ 0.34545455,  0.34545455])
>>> clf.intercept_
0.13636...
```

Ridge Complexity

This method has the same order of complexity than an *Ordinary Least Squares (OLS)*.

Generalized Cross-Validation

`RidgeCV` implements ridge regression with built-in cross-validation of the alpha parameter. The object works in the same way as GridSearchCV except that it defaults to Generalized Cross-Validation (GCV), an efficient form of leave-one-out cross-validation.

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.RidgeCV(alphas=[0.1, 1.0, 10.0])
>>> clf.fit ([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
RidgeCV(alphas=[0.10000000000000001, 1.0, 10.0], loss_func=None, cv=None,
    score_func=None, fit_intercept=True)
>>> clf.best_alpha
0.10000000000000001
```

References

- "Notes on Regularized Least Squares", Rifkin & Lippert (technical report, course slides).

Lasso

The `Lasso` is a linear model trained with L1 prior as regularizer. The objective function to minimize is:

$$0.5 * ||y - Xw||_2^2 + \alpha * ||w||_1$$

The lasso estimate thus solves the minimization of the least-squares penalty with $\alpha * ||w||_1$ added, where α is a constant and $||w||_1$ is the L1-norm of the parameter vector.

This formulation is useful in some contexts due to its tendency to prefer solutions with fewer parameter values, effectively reducing the number of variables upon which the given solution is dependent. For this reason, the Lasso and its variants are fundamental to the field of compressed sensing.

This implementation uses coordinate descent as the algorithm to fit the coefficients. See *Least Angle Regression* for another implementation.

```
>>> clf = linear_model.Lasso(alpha = 0.1)
>>> clf.fit ([[0, 0], [1, 1]], [0, 1])
Lasso(alpha=0.1, fit_intercept=True)
>>> clf.predict ([[1, 1]])
array([ 0.8])
```

The function `lasso_path()` computes the coefficients along the full path of possible values.

Examples:

- *Lasso regression example*,
- *Lasso parameter estimation with path and cross-validation*

Elastic Net

`ElasticNet` is a linear model trained with L1 and L2 prior as regularizer.

The objective function to minimize is in this case

$$0.5 * ||y - Xw||_2^2 + \alpha * \rho * ||w||_1 + \alpha * (1 - \rho) * 0.5 * ||w||_2^2$$

Examples:

- *Lasso regression example*
- *Lasso and Elastic Net*

Least Angle Regression

Least-angle regression (LARS) is a regression algorithm for high-dimensional data, developed by Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani.

The advantages of LARS are:

- It is computationally just as fast as forward selection and has the same order of complexity as an ordinary least squares.

- It produces a full piecewise linear solution path, which is useful in cross-validation or similar attempts to tune the model.
- If two variables are almost equally correlated with the response, then their coefficients should increase at approximately the same rate. The algorithm thus behaves as intuition would expect, and also is more stable.
- It is easily modified to produce solutions for other estimators, like the Lasso.
- It is effective in contexts where p >> n (i.e., when the number of dimensions is significantly greater than the number of points)

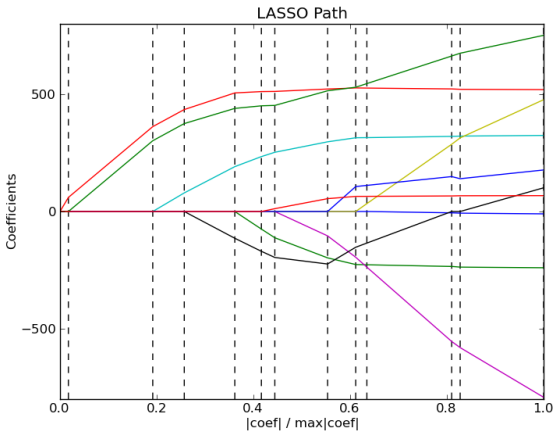
The disadvantages of the LARS method include:

- Because LARS is based upon an iterative refitting of the residuals, it would appear to be especially sensitive to the effects of noise. This problem is discussed in detail by Weisberg in the discussion section of the Efron et al. (2004) Annals of Statistics article.

The LARS model can be used using estimator `LARS`, or its low-level implementation `lars_path()`.

LARS Lasso

`LassoLARS` is a lasso model implemented using the LARS algorithm, and unlike the implementation based on coordinate_descent, this yields the exact solution, which is piecewise linear as a function of the norm of its coefficients.



Examples:

- *Lasso path using LARS*

The LARS algorithm provides the full path of the coefficients along the regularization parameter almost for free, thus a common operation consist of retrieving the path with function `lars_path()`

Mathematical formulation

The algorithm is similar to forward stepwise regression, but instead of including variables at each step, the estimated parameters are increased in a direction equiangular to each one's correlations with the residual.

Instead of giving a vector result, the LARS solution consists of a curve denoting the solution for each value of the L1 norm of the parameter vector. The full coeffients path is stored in the array `coef_path_`, which has size (n_features, max_features+1). The first column is always zero.

References:

- Original Algorithm is detailed in the paper Least Angle Regression by Hastie et al.

Bayesian Regression

Bayesian regression techniques can be used to include regularization parameters in the estimation procedure. This can be done by introducing some prior knowledge over the parameters. For example, penalization by weighted ℓ_2 norm is equivalent to setting Gaussian priors on the weights.

The advantages of *Bayesian Regression* are:

- It adapts to the data at hand.
- It can be used to include regularization parameters in the estimation procedure.

The disadvantages of *Bayesian Regression* include:

- Inference of the model can be time consuming.

Bayesian Ridge Regression

`BayesianRidge` tries to avoid the overfit issue of *Ordinary Least Squares (OLS)*, by adding the following prior on β:

$$p(\beta|\lambda) = \mathcal{N}(\beta|0, \lambda^{-1}\mathbf{I_p})$$

The resulting model is called *Bayesian Ridge Regression*, it is similar to the classical `Ridge`. λ is an *hyper-parameter* and the prior over β performs a shrinkage or regularization, by constraining the values of the weights to be small. Indeed, with a large value of λ, the Gaussian is narrowed around 0 which does not allow large values of β, and with low value of λ, the Gaussian is very flattened which allows values of β. Here, we use a *non-informative* prior for λ. The parameters are estimated by maximizing the *marginal log likelihood*. There is also a Gamma prior for λ and α:

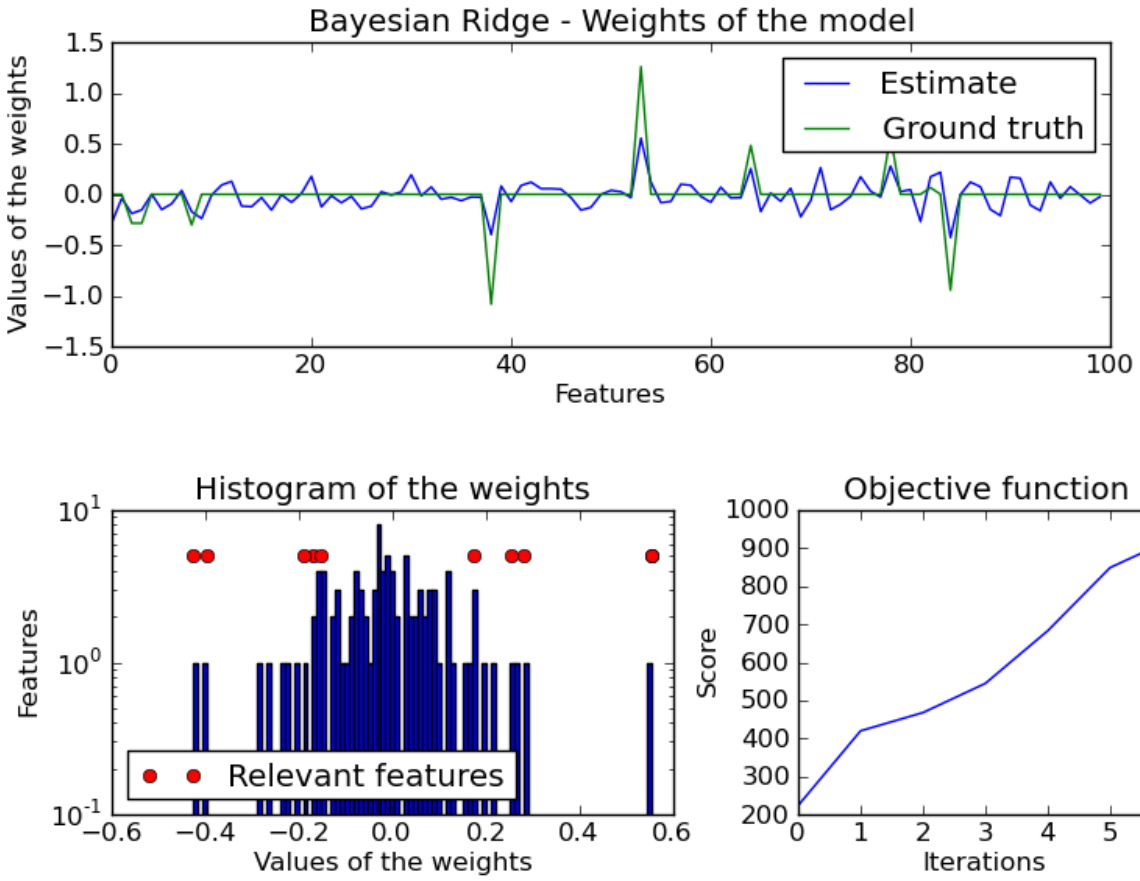
$$g(\alpha|\alpha_1, \alpha_2) = \frac{\alpha_2^{\alpha_1}}{\Gamma(\alpha_1)}\alpha^{\alpha_1-1}e^{-\alpha_2\alpha}$$

$$g(\lambda|\lambda_1, \lambda_2) = \frac{\lambda_2^{\lambda_1}}{\Gamma(\lambda_1)}\lambda^{\lambda_1-1}e^{-\lambda_2\lambda}$$

By default $\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 1.e^{-6}$**, *i.e.*** very slightly informative priors.

Bayesian Ridge Regression is used for regression:

```
>>> from scikits.learn import linear_model
>>> X = [[0., 0.], [1., 1.], [2., 2.], [3., 3.]]
>>> Y = [0., 1., 2., 3.]
>>> clf = linear_model.BayesianRidge()
```


Bayesian Ridge - Weights of the model
Values of the weights
Features
Estimate
Ground truth
Histogram of the weights
Features
Values of the weights
Relevant features
Objective function
Score
Iterations

```
>>> clf.fit (X, Y)
BayesianRidge(n_iter=300, verbose=False, lambda_1=1e-06, lambda_2=1e-06,
       fit_intercept=True, eps=0.001, alpha_2=1e-06, alpha_1=1e-06,
       compute_score=False)
```

After being fitted, the model can then be used to predict new values:

```
>>> clf.predict ([[1, 0.]])
array([ 0.50000013])
```

The weights β of the model can be access:

```
>>> clf.coef_
array([ 0.49999993,  0.49999993])
```

Due to the Bayesian framework, the weights found are slightly different to the ones found by *Ordinary Least Squares (OLS)*. However, *Bayesian Ridge Regression* is more robust to ill-posed problem.

Examples:

- *Bayesian Ridge Regression*

References

- More details can be found in the article Bayesian Interpolation by MacKay, David J. C.

Automatic Relevance Determination - ARD

`ARDRegression` adds a more sophisticated prior β, where we assume that each weight β_i is drawn in a Gaussian distribution, centered on zero and with a precision λ_i:

$$p(\beta|\lambda) = \mathcal{N}(\beta|0, A^{-1})$$

with $diag\ (A) = \lambda = \{\lambda_1, ..., \lambda_p\}$. There is also a Gamma prior for λ and α:

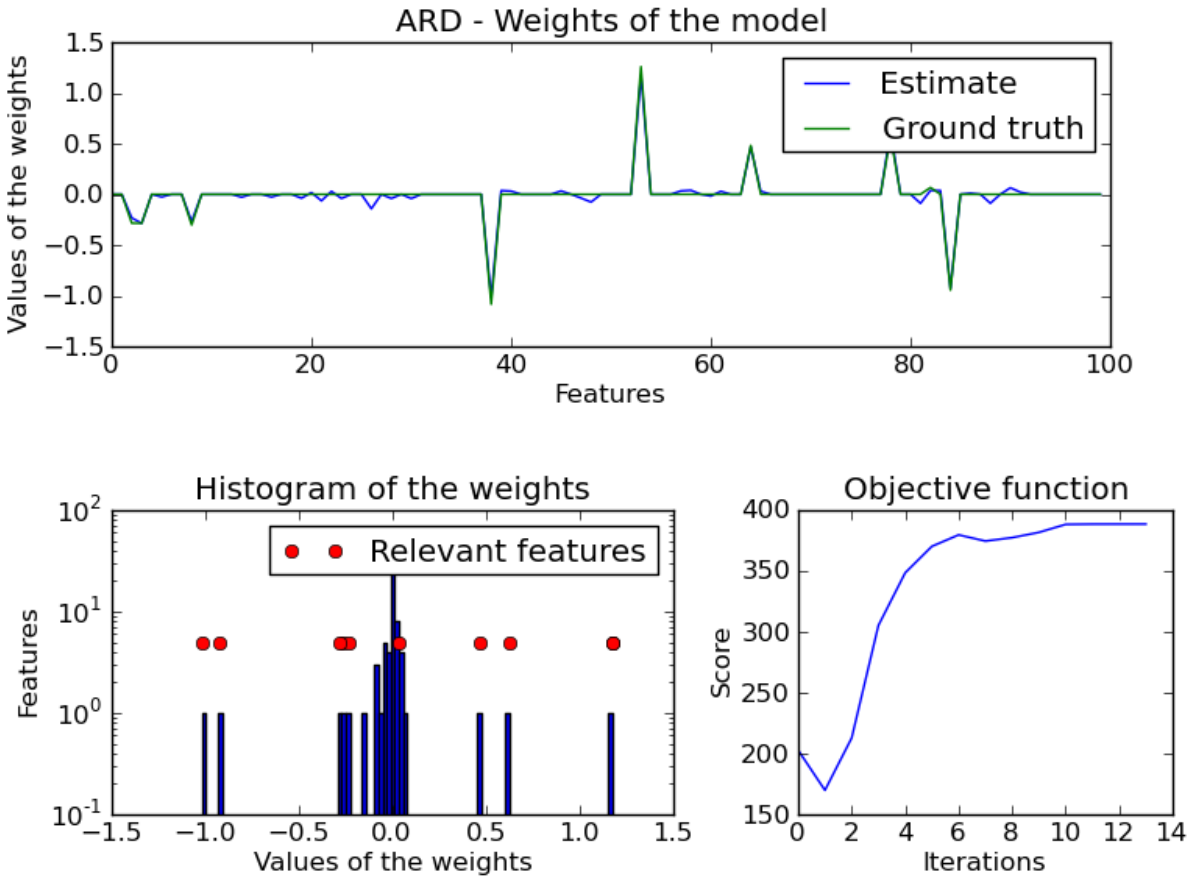
$$g(\alpha|\alpha_1, \alpha_2) = \frac{\alpha_2^{\alpha_1}}{\Gamma(\alpha_1)} \alpha^{\alpha_1 - 1} e^{-\alpha_2 \alpha}$$

$$g(\lambda|\lambda_1, \lambda_2) = \frac{\lambda_2^{\lambda_1}}{\Gamma(\lambda_1)} \lambda^{\lambda_1 - 1} e^{-\lambda_2 \lambda}$$

By default $\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 1.e-6$**, *i.e.*** very slightly informative priors.

Examples:

- *Automatic Relevance Determination Regression (ARD)*


ARD - Weights of the model
Values of the weights
Features
Estimate
Ground truth
Histogram of the weights
Features
Values of the weights
Relevant features
Objective function
Score
Iterations

Mathematical formulation

A prior is introduced as a distribution $p(\theta)$ over the parameters. This distribution is set before processing the data. The parameters of a prior distribution are called *hyper-parameters*. This description is based on the Bayes theorem :

$$p(\theta|\{X,y\}) = \frac{p(\{X,y\}|\theta)p(\theta)}{p(\{X,y\})}$$

With :

- $p(X,y|\theta)$ the likelihood : it expresses how probable it is to observe X, y given θ.
- $p(X,y)$ the marginal probability of the data : it can be considered as a normalizing constant, and is computed by integrating $p(X,y|\theta)$ with respect to θ.
- $p(\theta)$ the prior over the parameters : it expresses the knowledge that we can have about θ before processing the data.
- $p(\theta|X,y)$ the conditional probability (or posterior probability) : it expresses the uncertainty in θ after observing the data.

All the following regressions are based on the following Gaussian assumption:

$$p(y|X,w,\alpha) = \mathcal{N}(y|Xw,\alpha)$$

where α is the precision of the noise.

References

- Original Algorithm is detailed in the book *Bayesian learning for neural networks* by Radford M. Neal

Logisitic regression

If the task at hand is to do choose which class a sample belongs to given a finite (hopefuly small) set of choices, the learning problem is a classification, rather than regression. Linear models can be used for such a decision, but it is best to use what is called a logistic regression, that doesn't try to minimize the sum of square residuals, as in regression, but rather a "hit or miss" cost.

The `LogisticRegression` class can be used to do L1 or L2 penalized logistic regression, in order to have sparse predicting weights.

Examples:

- *Logistic Regression*
- *Path with L1- Logistic Regression*

Stochastic Gradient Descent - SGD

Stochastic gradient descent is a simple yet very efficient approach to fit linear models. It is particulary useful when the number of samples (and the number of features) is very large.

The classes `SGDClassifier` and `SGDRegressor` provide functionality to fit linear models for classification and regression using different (convex) loss functions and different penalties.

References

- *Stochastic Gradient Descent*

1.3.2 Support Vector Machines

Support vector machines (SVMs) are a set of supervised learning methods used for *classification*, *regression* and *outliers detection*.

The advantages of Support Vector Machines are:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different *Kernel functions* can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of Support Vector Machines include:

- If the number of features is much greater than the number of samples, the method is likely to give poor performances.
- SVMs do not directly provide probability estimates, these are calculated using five-fold cross-validation, and thus performance can suffer.

Classification

`SVC`, `NuSVC` and `LinearSVC` are classes capable of performing multi-class classification on a dataset.

`SVC` and `NuSVC` are similar methods, but accept slightly different sets of parameters and have different mathematical formulations (see section *Mathematical formulation*). On the other hand, `LinearSVC` is another implementation of Support Vector Classification for the case of a linear kernel. Note that `LinearSVC` does not accept keyword 'kernel', as this is assumed to be linear. It also lacks some of the members of `SVC` and `NuSVC`, like support_.

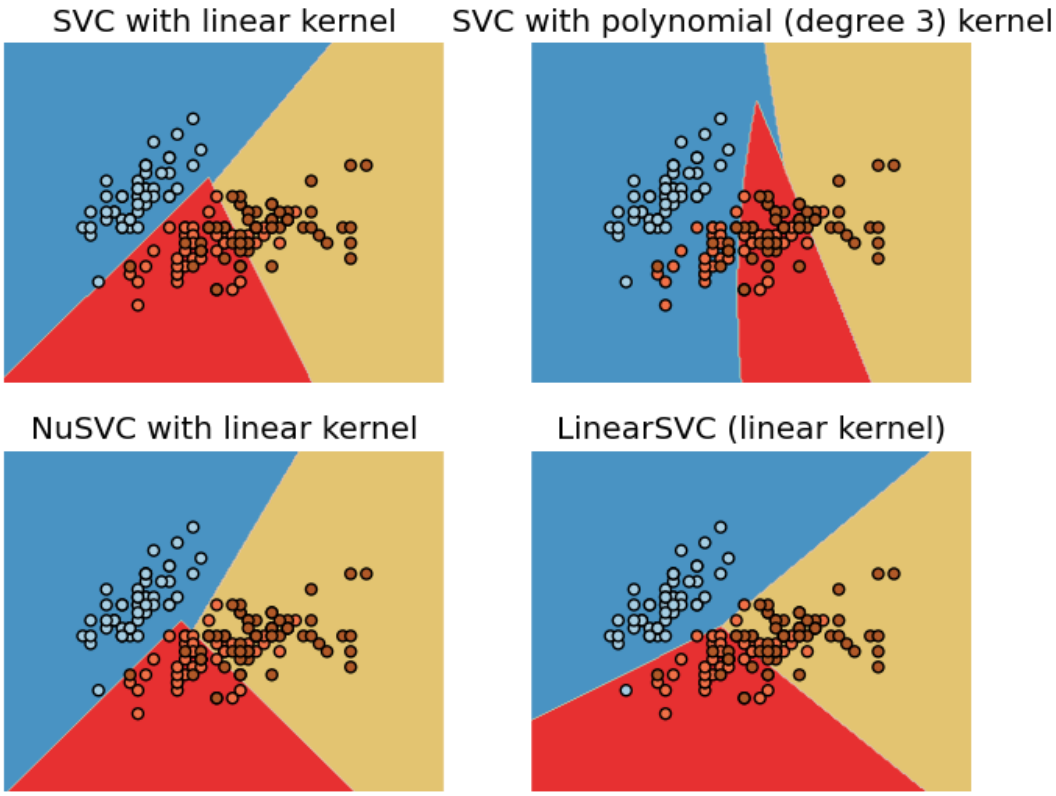
As other classifiers, `SVC`, `NuSVC` and `LinearSVC` take as input two arrays: an array X of size [n_samples, n_features] holding the training samples, and an array Y of integer values, size [n_samples], holding the class labels for the training samples:

```
>>> from scikits.learn import svm
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = svm.SVC()
>>> clf.fit(X, Y)
SVC(kernel='rbf', C=1.0, probability=False, degree=3, coef0=0.0, eps=0.001,
  cache_size=100.0, shrinking=True, gamma=0.5)
```

After being fitted, the model can then be used to predict new values:

```
>>> clf.predict([[2., 2.]])
array([ 1.])
```

SVMs decision function depends on some subset of the training data, called the support vectors. Some properties of these support vectors can be found in members *support_vectors_*, *support_* and *n_support*:


SVC with linear kernel
SVC with polynomial (degree 3) kernel
NuSVC with linear kernel
LinearSVC (linear kernel)

```
>>> # get support vectors
>>> clf.support_vectors_
array([[ 0.,  0.],
       [ 1.,  1.]])
>>> # get indices of support vectors
>>> clf.support_
array([0, 1]...)
>>> # get number of support vectors for each class
>>> clf.n_support_
array([1, 1]...)
```

Multi-class classification

SVC and NuSVC implement the "one-against-one" approach (Knerr et al., 1990) for multi- class classification. If n_class is the number of classes, then n_class * (n_class - 1)/2 classifiers are constructed and each one trains data from two classes.

```
>>> X = [[0], [1], [2], [3]]
>>> Y = [0, 1, 2, 3]
>>> clf = svm.SVC()
>>> clf.fit(X, Y)
SVC(kernel='rbf', C=1.0, probability=False, degree=3, coef0=0.0, eps=0.001,
  cache_size=100.0, shrinking=True, gamma=0.25)
>>> dec = clf.decision_function([[1]])
>>> dec.shape[1] # 4 classes: 4*3/2 = 6
6
```

On the other hand, LinearSVC implements "one-vs-the-rest" multi-class strategy, thus training n_class models. If there are only two classes, only one model is trained.

```
>>> lin_clf = svm.LinearSVC()
>>> lin_clf.fit(X, Y)
LinearSVC(loss='l2', C=1.0, intercept_scaling=1, fit_intercept=True,
     eps=0.0001, penalty='l2', multi_class=False, dual=True)
>>> dec = lin_clf.decision_function([[1]])
>>> dec.shape[1]
4
```

See *Mathematical formulation* for a complete description of the decision function.

Examples:

- *Plot different SVM classifiers in the iris dataset*,
- *SVM: Maximum margin separating hyperplane*,
- *SVM-Anova: SVM with univariate feature selection*,
- *Non-linear SVM*

Regression

The method of Support Vector Classification can be extended to solve regression problems. This method is called Support Vector Regression.

The model produced by support vector classification (as described above) depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin.

Analogously, the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction.

There are two flavors of Support Vector Regression: `SVR` and `NuSVR`.

As with classification classes, the fit method will take as argument vectors X, y, only that in this case y is expected to have floating point values instead of integer values.

```
>>> from scikits.learn import svm
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> clf = svm.SVR()
>>> clf.fit(X, y)
SVR(kernel='rbf', C=1.0, probability=False, degree=3, shrinking=True,
  eps=0.001, p=0.1, cache_size=100.0, coef0=0.0, nu=0.5, gamma=0.5)
>>> clf.predict([[1, 1]])
array([ 1.5])
```

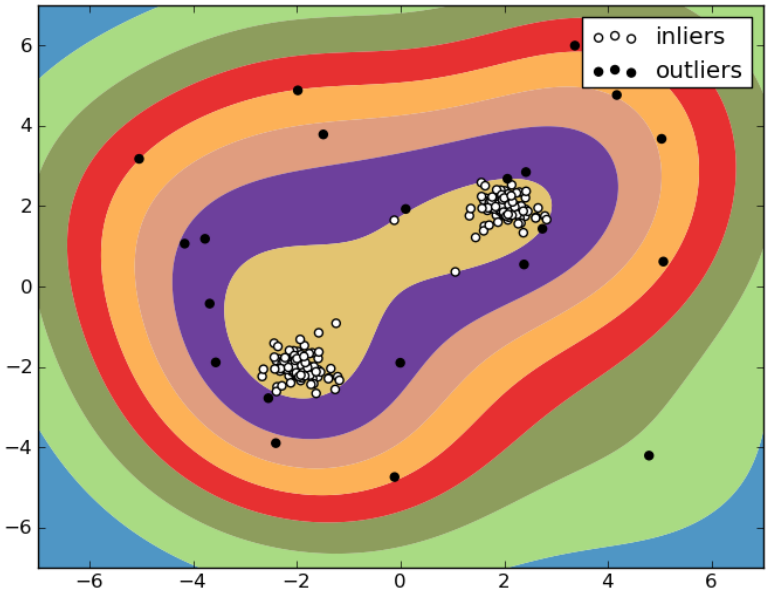
Examples:

- *Support Vector Regression (SVR) using linear and non-linear kernels*

Density estimation, outliers detection

One-class SVM is used for outliers detection, that is, given a set of samples, it will detect the soft boundary of that set so as to classify new points as belonging to that set or not. The class that implements this is called `OneClassSVM`

In this case, as it is a type of unsupervised learning, the fit method will only take as input an array X, as there are no class labels.



Examples:

- *One-class SVM with non-linear kernel (RBF)*
- *Species distribution modeling*

Support Vector machines for sparse data

There is support for sparse data given in any matrix in a format supported by scipy.sparse. Classes have the same name, just prefixed by the *sparse* namespace, and take the same arguments, with the exception of training and test data, which is expected to be in a matrix format defined in scipy.sparse.

For maximum efficiency, use the CSR matrix format as defined in scipy.sparse.csr_matrix.

Implemented classes are `SVC`, `NuSVC`, `SVR`, `NuSVR`, `OneClassSVM`, `LinearSVC`.

Complexity

Support Vector Machines are powerful tools, but their compute and storage requirements increase rapidly with the number of training vectors. The core of an SVM is a quadratic programming problem (QP), separating support vectors from the rest of the training data. The QP solver used by this libsvm-based implementation scales between $O(n_{features} \times n_{samples}^2)$ and $O(n_{features} \times n_{samples}^3)$ depending on how efficiently the libsvm cache is used in practice (dataset dependent). If the data is very sparse $n_{features}$ should be replaced by the average number of non-zero features in a sample vector.

Also note that for the linear case, the algorithm used in `LinearSVC` by the liblinear implementation is much more efficient than its libsvm-based `SVC` counterpart and can scale almost linearly to millions of samples and/or features.

Tips on Practical Use

- Support Vector Machine algorithms are not scale invariant, so it is highly recommended to scale your data. For example, scale each attribute on the input vector X to [0,1] or [-1,+1], or standardize it to have mean 0 and variance 1. Note that the *same* scaling must be applied to the test vector to obtain meaningful results. See The CookBook for some examples on scaling.
- Parameter nu in NuSVC/OneClassSVM/NuSVR approximates the fraction of training errors and support vectors.
- If data for classification are unbalanced (e.g. many positive and few negative), set class_weight='auto' and/or try different penalty parameters C.
- Specify larger cache size (keyword cache) for huge problems.
- The underlying `LinearSVC` implementation uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller eps parameter.

Kernel functions

The *kernel function* can be any of the following:

- linear: $< x_i, x'_j >$.
- polynomial: $(\gamma < x, x' > +r)^d$. d is specified by keyword *degree*.

- rbf ($exp(-\gamma|x-x'|^2), \gamma > 0$). γ is specified by keyword gamma.
- sigmoid ($tanh(< x_i, x_j > +r)$).

Different kernels are specified by keyword kernel at initialization:

```
>>> linear_svc = svm.SVC(kernel='linear')
>>> linear_svc.kernel
'linear'
>>> rbf_svc = svm.SVC(kernel='rbf')
>>> rbf_svc.kernel
'rbf'
```

Custom Kernels

You can define your own kernels by either giving the kernel as a python function or by precomputing the Gram matrix.

Classifiers with custom kernels behave the same way as any other classifiers, except that:

- Field *support_vectors_* is now empty, only indices of support vectors are stored in *support_*
- A reference (and not a copy) of the first argument in the fit() method is stored for future reference. If that array changes between the use of fit() and predict() you will have unexpected results.

Using python functions as kernels You can also use your own defined kernels by passing a function to the keyword *kernel* in the constructor.

Your kernel must take as arguments two matrices and return a third matrix.

The following code defines a linear kernel and creates a classifier instance that will use that kernel:

```
>>> import numpy as np
>>> from scikits.learn import svm
>>> def my_kernel(x, y):
...     return np.dot(x, y.T)
...
>>> clf = svm.SVC(kernel=my_kernel)
```

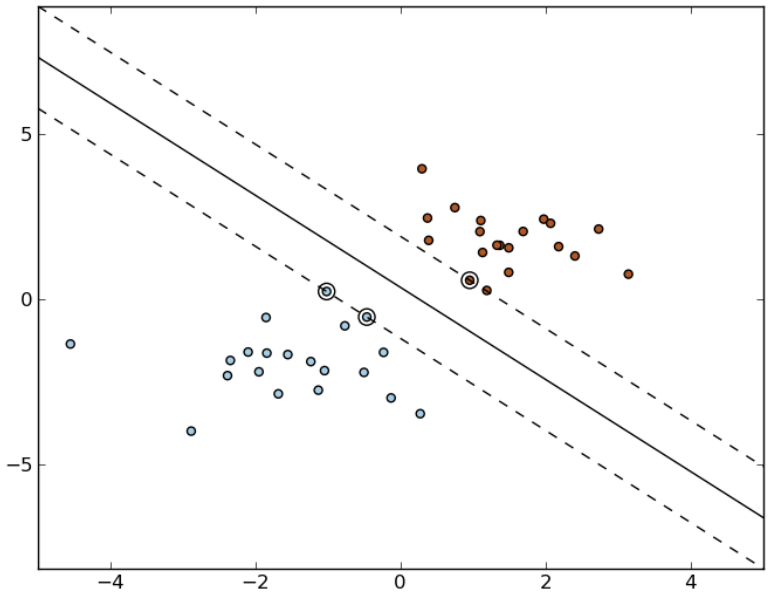
Using the Gram matrix Set kernel='precomputed' and pass the Gram matrix instead of X in the fit method.

Examples:

- *SVM with custom kernel*.

Mathematical formulation

A support vector machine constructs a hyper-plane or set of hyper-planes in a high or infinite dimensional space, which can be used for classification, regression or other tasks. Intuitively, a good separation is achieved by the hyper-plane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.



SVC

Given training vectors $x_i \in R^n$, i=1,..., l, in two classes, and a vector $y \in R^l$ such that $y_i \in 1, -1$, SVC solves the following primal problem:

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1,l} \zeta_i$$

$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$
$$\zeta_i \geq 0, i = 1, ..., l$$

Its dual is

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$
$$\text{subject to } y^T \alpha = 0$$
$$0 \leq \alpha_i \leq C, i = 1, ..., l$$

where e is the vector of all ones, C > 0 is the upper bound, Q is an l by l positive semidefinite matrix, $Q_i j \equiv K(x_i, x_j)$ and $\phi(x_i)^T \phi(x)$ is the kernel. Here training vectors are mapped into a higher (maybe infinite) dimensional space by the function ϕ

The decision function is:

$$sgn(\sum_{i=1}^{l} y_i \alpha_i K(x_i, x) + \rho)$$

This parameters can be accessed through the members *dual_coef_* which holds the product $y_i \alpha_i$, *support_vectors_* which holds the support vectors, and *intercept_* which holds the independent term $-\rho$:

References:

- "Automatic Capacity Tuning of Very Large VC-dimension Classifiers" I Guyon, B Boser, V Vapnik - Advances in neural information processing 1993,
- "Support-vector networks" C. Cortes, V. Vapnik, Machine Leaming, 20, 273-297 (1995)

NuSVC

We introduce a new parameter ν which controls the number of support vectors and training errors. The parameter $\nu \in (0, 1]$ is an upper bound on the fraction of training errors and a lower bound of the fraction of support vectors.

Frequently Asked Questions

- Q: Can I get the indices of the support vectors instead of the support vectors ?

  A: The underlying C implementation does not provide this information.

Implementation details

Internally, we use libsvm and liblinear to handle all computations. These libraries are wrapped using C and Cython.

References:

For a description of the implementation and details of the algorithms used, please refer to

- LIBSVM: a library for Support Vector Machines
- LIBLINEAR – A Library for Large Linear Classification

1.3.3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to discriminative learning of linear classifiers under convex loss functions such as (linear) Support Vector Machines and Logistic Regression. Even though SGD has been around in the machine learning community for a long time, it has received a considerable amount of attention just recently in the context of large-scale learning.

SGD has been successfully applied to large-scale and sparse machine learning problems often encountered in text classification and natural language processing. Given that the data is sparse, the classifiers in this module easily scale to problems with more than 10^5 training examples and more than 10^4 features.

The advantages of Stochastic Gradient Descent are:

- Efficiency.
- Ease of implementation (lots of opportunities for code tuning).

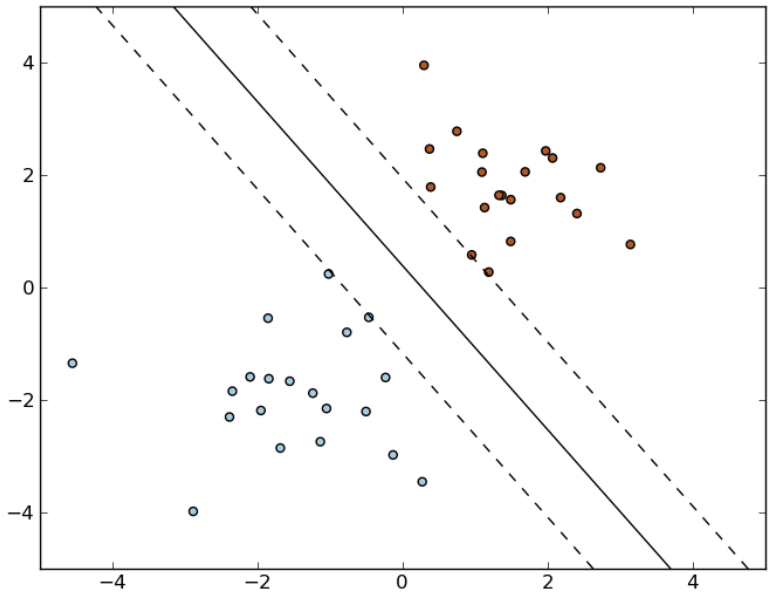
The disadvantages of Stochastic Gradient Descent include:

- SGD requires a number of hyperparameters including the regularization parameter and the number of iterations.
- SGD is sensitive to feature scaling.

Classification

Warning: Make sure you permute (shuffle) your training data before fitting the model or use *shuffle=True* to shuffle after each iterations.

The class `SGDClassifier` implements a plain stochastic gradient descent learning routine which supports different loss functions and penalties for classification.



As other classifiers, SGD has to be fitted with two arrays: an array X of size [n_samples, n_features] holding the training samples, and an array Y of size [n_samples] holding the target values (class labels) for the training samples:

```
>>> from scikits.learn.linear_model import SGDClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = SGDClassifier(loss="hinge", penalty="l2")
>>> clf.fit(X, y)
SGDClassifier(loss='hinge', n_jobs=1, shuffle=False, verbose=0, n_iter=5,
       fit_intercept=True, penalty='l2', rho=1.0, alpha=0.0001)
```

After being fitted, the model can then be used to predict new values:

```
>>> clf.predict([[2., 2.]])
array([ 1.])
```

SGD fits a linear model to the training data. The member *coef_* holds the model parameters:

```
>>> clf.coef_
array([ 9.90090187,  9.90090187])
```

Member *intercept_* holds the intercept (aka offset or bias):

```
>>> clf.intercept_
array(-9.9900299301496904)
```

Whether or not the model should use an intercept, i.e. a biased hyperplane, is controlled by the parameter *fit_intercept*.

To get the signed distance to the hyperplane use *decision_function*:

```
>>> clf.decision_function([[2., 2.]])
array([ 29.61357756])
```

The concrete loss function can be set via the *loss* parameter. `SGDClassifier` supports the following loss functions:

- *loss="hinge"*: (soft-margin) linear Support Vector Machine.
- *loss="modified_huber"*: smoothed hinge loss.
- *loss="log"*: Logistic Regression

The first two loss functions are lazy, they only update the model parameters if an example violates the margin constraint, which makes training very efficient. Log loss, on the other hand, provides probability estimates.

In the case of binary classification and *loss="log"* you get a probability estimate P(y=C|x) using *predict_proba*, where *C* is the largest class label:

```
>>> clf = SGDClassifier(loss="log").fit(X, y)
>>> clf.predict_proba([[1., 1.]])
array([ 0.99999949])
```

The concrete penalty can be set via the *penalty* parameter. *SGD* supports the following penalties:

- *penalty="l2"*: L2 norm penalty on *coef_*.
- *penalty="l1"*: L1 norm penalty on *coef_*.
- *penalty="elasticnet"*: Convex combination of L2 and L1; *rho * L2 + (1 - rho) * L1*.

The default setting is *penalty="l2"*. The L1 penalty leads to sparse solutions, driving most coefficients to zero. The Elastic Net solves some deficiencies of the L1 penalty in the presence of highly correlated attributes. The parameter *rho* has to be specified by the user.

`SGDClassifier` supports multi-class classification by combining multiple binary classifiers in a "one versus all" (OVA) scheme. For each of the *K* classes, a binary classifier is learned that discriminates between that and all other *K-1* classes. At testing time, we compute the confidence score (i.e. the signed distances to the hyperplane) for each classifier and choose the class with the highest confidence. The Figure below illustrates the OVA approach on the iris dataset. The dashed lines represent the three OVA classifiers; the background colors show the decision surface induced by the three classifiers.

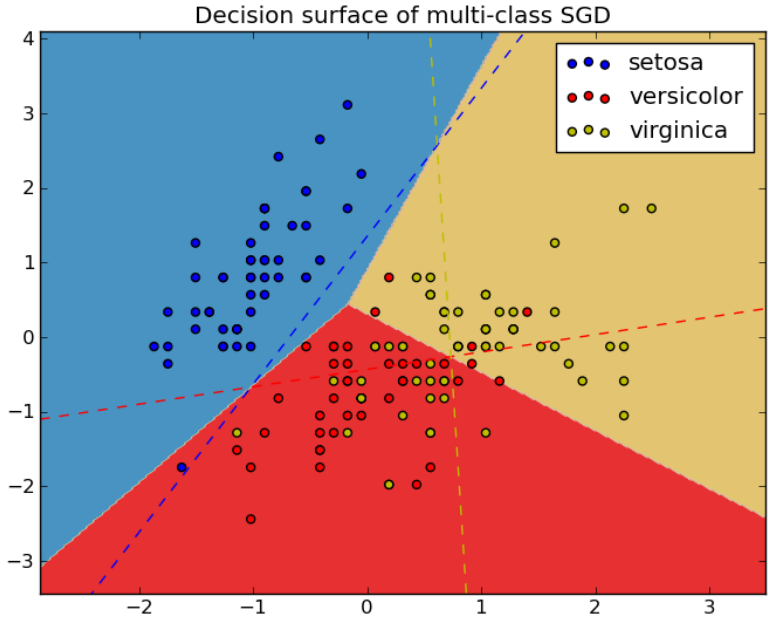
In the case of multi-class classification *coef_* is a two-dimensionaly array of shape [n_classes, n_features] and *intercept_* is a one dimensional array of shape [n_classes]. The i-th row of *coef_* holds the weight vector of the OVA classifier for the i-th class; classes are indexed in ascending order (see member *classes*).

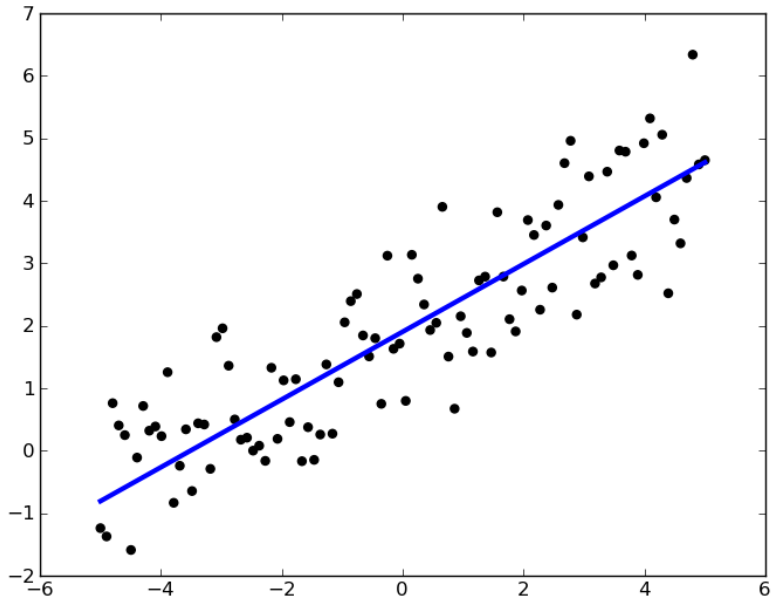
Examples:

- *SGD: Maximum margin separating hyperplane*,
- *Plot multi-class SGD on the iris dataset*

Regression

The class `SGDRegressor` implements a plain stochastic gradient descent learning routine which supports different loss functions and penalties to fit linear regression models.


Decision surface of multi-class SGD
setosa
versicolor
virginica
4
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3


7
6
5
4
3
2
1
0
−1
−2
−6
−4
−2
0
2
4
6

The concrete loss function can be set via the *loss* parameter. `SGDRegressor` supports the following loss functions:

- *loss="squared_loss"*: Ordinary least squares.
- *loss="huber"*: Huber loss for robust regression.

Examples:

- *Ordinary Least Squares with SGD*,

Stochastic Gradient Descent for sparse data

Note: The sparse implementation produces slightly different results than the dense implementation due to a shrunk learning rate for the intercept.

There is support for sparse data given in any matrix in a format supported by scipy.sparse. Classes have the same name, just prefixed by the *sparse* namespace, and take the same arguments, with the exception of training and test data, which is expected to be in a matrix format defined in scipy.sparse.

For maximum efficiency, use the CSR matrix format as defined in scipy.sparse.csr_matrix.

Implemented classes are `SGDClassifier` and `SGDRegressor`.

Examples:

- *Classification of text documents using sparse features*

Complexity

The major advantage of SGD is its efficiency, which is basically linear in the number of training examples. If X is a matrix of size (n, p) training has a cost of $O(knp\bar{p})$, where k is the number of iterations (epochs) and $\bar{p}$ is the average number of non-zero attributes per sample.

Recent theoretical results, however, show that the runtime to get some desired optimization accuracy does not increase as the training set size increases.

Tips on Practical Use

- Stochastic Gradient Descent is sensitive to feature scaling, so it is highly recommended to scale your data. For example, scale each attribute on the input vector X to [0,1] or [-1,+1], or standardize it to have mean 0 and variance 1. Note that the *same* scaling must be applied to the test vector to obtain meaningful results. See The CookBook for some examples on scaling. If your attributes have an intrinsic scale (e.g. word frequencies or indicator features) scaling is not needed.
- Finding a reasonable regularization term α is best done using grid search *for alpha in 10.0**-np.arange(1,7).*
- Empirically, we found that SGD converges after observing approx. 10^6 training samples. Thus, a reasonable first guess for the number of iterations is *n_iter = np.ceil(10**6 / n)*, where *n* is the size of the training set.

References:

- "Efficient BackProp" Y. LeCun, L. Bottou, G. Orr, K. Müller - In Neural Networks: Tricks of the Trade 1998.

Mathematical formulation

Given a set of training examples $(x_1, y_1), \ldots, (x_n, y_n)$ where $x_i \in \mathbf{R}^n$ and $y_i \in \{-1, 1\}$, our goal is to learn a linear scoring function $f(x) = w^T x + b$ with model parameters $w \in \mathbf{R}^m$ and intercept $b \in \mathbf{R}$. In order to make predictions, we simply look at the sign of $f(x)$. A common choice to find the model parameters is by minimizing the regularized training error given by

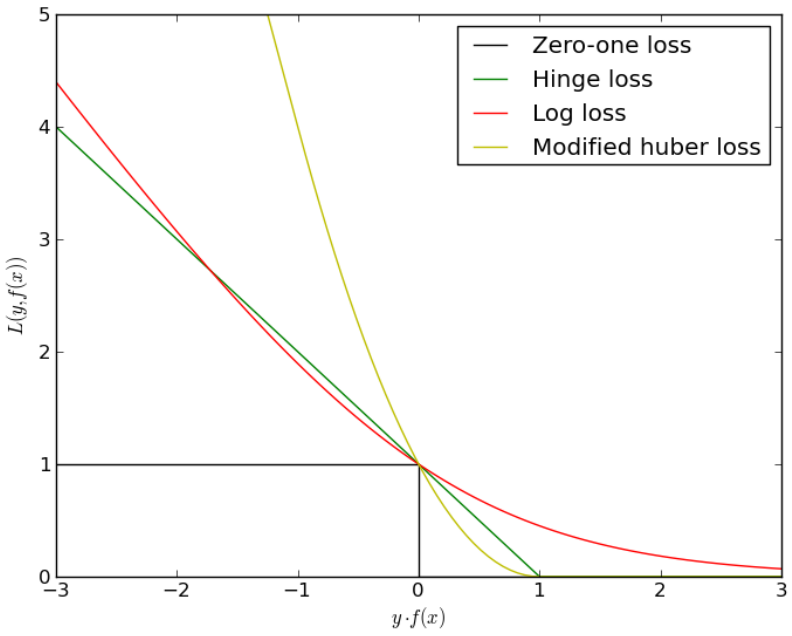
$$E(w, b) = \sum_{i=1}^{n} L(y_i, f(x_i)) + \alpha R(w)$$

where L is a loss function that measures model (mis)fit and R is a regularization term (aka penalty) that penalizes model complexity; $\alpha > 0$ is a non-negative hyperparameter.

Different choices for L entail different classifiers such as

- Hinge: (soft-margin) Support Vector Machines.
- Log: Logistic Regression.
- Least-Squares: Ridge Regression.

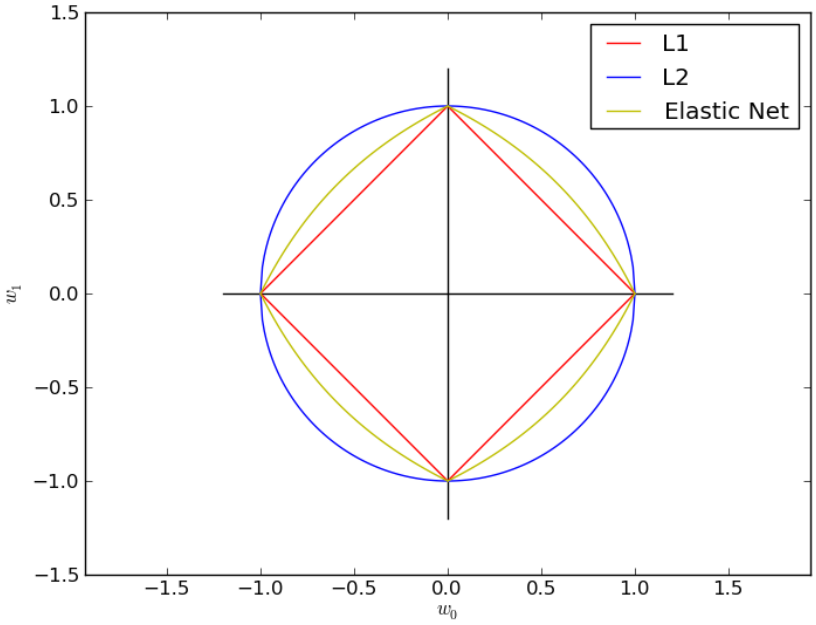
All of the above loss functions can be regarded as an upper bound on the misclassification error (Zero-one loss) as shown in the Figure below.



Popular choices for the regularization term R include:

- L2 norm: $R(w) := \frac{1}{2} \sum_{i=1}^{n} w_i^2$,
- L1 norm: $R(w) := \sum_{i=1}^{n} |w_i|$, which leads to sparse solutions.
- Elastic Net: $R(w) := \rho \frac{1}{2} \sum_{i=1}^{n} w_i^2 + (1 - \rho) \sum_{i=1}^{n} |w_i|$, a convex combination of L2 and L1.

The Figure below shows the contours of the different regularization terms in the parameter space when $R(w) = 1$.



SGD

Stochastic gradient descent is an optimization method for unconstrained optimization problems. In contrast to (batch) gradient descent, SGD approximates the true gradient of $E(w, b)$ by considering a single training example at a time.

The class `SGDClassifier` implements a first-order SGD learning routine. The algorithm iterates over the training examples and for each example updates the model parameters according to the update rule given by

$$w \leftarrow w - \eta(\alpha \frac{\partial R(w)}{\partial w} + \frac{\partial L(w^T x_i + b, y_i)}{\partial w})$$

where η is the learning rate which controls the step-size in the parameter space. The intercept b is updated similarly but without regularization.

The model parameters can be accessed through the members coef_ and intercept_:

- Member coef_ holds the weights w
- Member intercept_ holds b

References:

- "Solving large scale linear prediction problems using stochastic gradient descent algorithms" T. Zhang - In Proceedings of ICML '04.
- "Regularization and variable selection via the elastic net" H. Zou, T. Hastie - Journal of the Royal Statistical Society Series B, 67 (2), 301-320.

Implementation details

The implementation of SGD is influenced by the Stochastic Gradient SVM of Léon Bottou. Similar to SvmSGD, the weight vector is represented as the product of a scalar and a vector which allows an efficient weight update in

the case of L2 regularization. In the case of sparse feature vectors, the intercept is updated with a smaller learning rate (multiplied by 0.01) to account for the fact that it is updated more frequently. Training examples are picked up sequentially and the learning rate is lowered after each observed example. We adopted the learning rate schedule from Shalev-Shwartz et al. 2007. For multi-class classification, a "one versus all" approach is used. We use the truncated gradient algorithm proposed by Tsuruoka et al. 2009 for L1 regularization (and the Elastic Net). The code is written in Cython.

References:

- "Stochastic Gradient Descent" L. Bottou - Website, 2010.
- "Pegasos: Primal estimated sub-gradient solver for svm" S. Shalev-Shwartz, Y. Singer, N. Srebro - In Proceedings of ICML '07.
- "Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty" Y. Tsuruoka, J. Tsujii, S. Ananiadou - In Proceedings of the AFNLP/ACL '09.

1.3.4 Nearest Neighbors

The principle behind nearest neighbor methods is to find the k training points closest in euclidean distance to x0, and then classify using a majority vote among the k neighbors.

Despite its simplicity, nearest neighbors has been successful in a large number of classification problems, including handwritten digits or satellite image scenes. It is ofter successful in situation where the decision boundary is very irregular.

Classification

The `NeighborsClassifier` implements the nearest-neighbors classification method using a vote heuristic: the class most present in the k nearest neighbors of a point is assigned to this point.

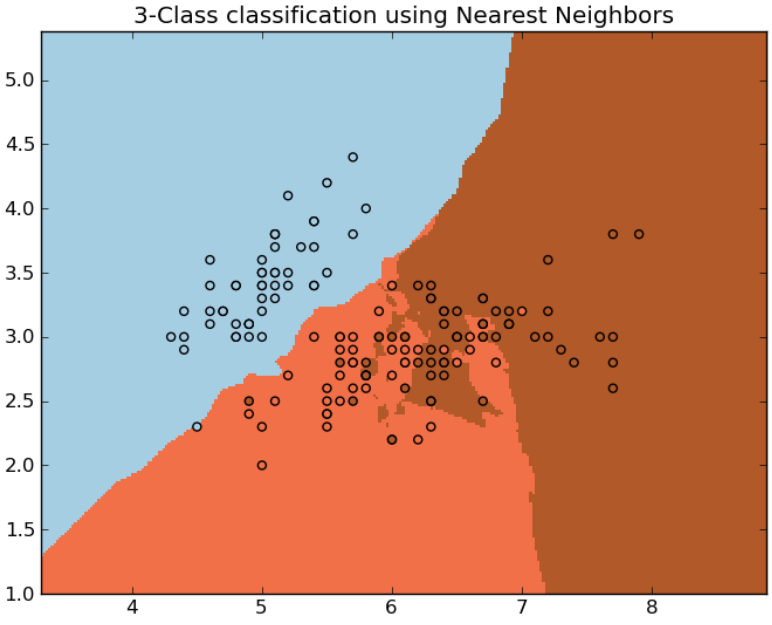
It is possible to use different nearest neighbor search algorithms by using the keyword `algorithm`. Possible values are `'auto'`, `'ball_tree'`, `'brute'` and `'brute_inplace'`. `'ball_tree'` will create an instance of `BallTree` to conduct the search, which is usually very efficient in low-dimensional spaces. In higher dimension, a brute-force approach is prefered thus parameters `'brute'` and `'brute_inplace'` can be used . Both conduct a brute-force search, the difference being that `'brute_inplace'` does not perform any precomputations, and thus is better suited for low-memory settings. Finally, `'auto'` is a simple heuristic that will guess the best approach based on the current dataset.

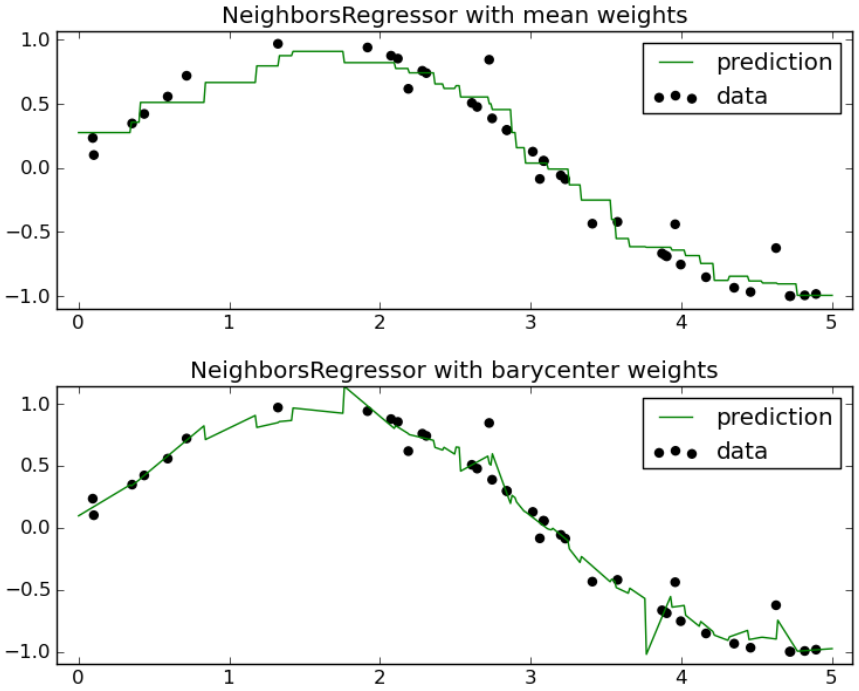
Examples:

- *Nearest Neighbors*: an example of classification using nearest neighbor.

Regression

The `NeighborsRegressor` estimator implements a nearest-neighbors regression method by weighting the targets of the k-neighbors. Two different weighting strategies are implemented: `barycenter` and `mean`. `barycenter` will apply the weights that best reconstruct the point from its neighbors while `mean` will apply constant weights to each point. This plot shows the behavior of both classifier for a simple regression task.


3-Class classification using Nearest Neighbors
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8


NeighborsRegressor with mean weights
prediction
data
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
NeighborsRegressor with barycenter weights
prediction
data
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5

Examples:

- *k-Nearest Neighbors regression*: an example of regression using nearest neighbor.

Efficient implementation: the ball tree

Behind the scenes, nearest neighbor search is done by the object `BallTree`, which is a fast way to perform neighbor searches in data sets of very high dimensionality.

This class provides an interface to an optimized BallTree implementation to rapidly look up the nearest neighbors of any point. Ball Trees are particularly useful for very high-dimensionality data, where more traditional tree searches (e.g. KD-Trees) perform poorly.

The cost is a slightly longer construction time, though for repeated queries, this added construction time quickly becomes insignificant.

A Ball Tree reduces the number of candidate points for a neighbor search through use of the triangle inequality:

$$|x + y| \leq |x| + |y|$$

Each node of the `BallTree` defines a centroid, C, and a radius r such that each point in the node lies within the hypersphere of radius r, centered at C. With this setup, a single distance calculation between a test point and the centroid is sufficient to determine a lower bound on the distance to all points within the node. Carefully taking advantage of this property leads to determining neighbors in O[log(N)] time, as opposed to O[N] time for a brute-force search.

A pure C implementation of brute-force search is also provided in function `knn_brute()`.

References:

- "Five balltree construction algorithms", Omohundro, S.M., International Computer Science Institute Technical Report

1.3.5 Feature selection

Univariate feature selection

Univariate feature selection works by selecting the best features based on univariate statistical tests. It can seen as a preprocessing step to an estimator. The *scikit.learn* exposes feature selection routines a objects that implement the *transform* method. The k-best features can be selected based on:

`scikits.learn.feature_selection.univariate_selection.`**`SelectKBest`**(*score_func*, *k=10*)

Filter : Select the k lowest p-values

or by setting a percentile of features to keep using

`scikits.learn.feature_selection.univariate_selection.`**`SelectPercentile`**(*score_func*, *percentile=10*)

Filter : Select the best percentile of the p_values

or using common statistical quantities:

scikits.learn.feature_selection.univariate_selection.**SelectFpr**(*score_func*, *alpha=0.050000000000000003*)

Filter : Select the pvalues below alpha

scikits.learn.feature_selection.univariate_selection.**SelectFdr**(*score_func*, *alpha=0.050000000000000003*)

Filter : Select the p-values corresponding to an estimated false discovery rate of alpha. This uses the Benjamini-Hochberg procedure

scikits.learn.feature_selection.univariate_selection.**SelectFwe**(*score_func*, *alpha=0.050000000000000003*)

Filter : Select the p-values corresponding to a corrected p-value of alpha

These objects take as input a scoring function that returns univariate p-values.

Examples:

Univariate Feature Selection

Feature scoring functions

Warning: Beware not to use a regression scoring function with a classification problem.

For classification

scikits.learn.feature_selection.univariate_selection.**f_classif**(*X*, *y*)

Compute the Anova F-value for the provided sample

Parameters **X** : array of shape (n_samples, n_features)

the set of regressors sthat will tested sequentially

y : array of shape(n_samples)

the data matrix

Returns **F** : array of shape (m),

the set of F values

pval : array of shape(m),

the set of p-values

For regression

scikits.learn.feature_selection.univariate_selection.**f_regression**(*X*, *y*, *center=True*)

Quick linear model for testing the effect of a single regressor, sequentially for many regressors This is done in 3 steps: 1. the regressor of interest and the data are orthogonalized wrt constant regressors 2. the cross correlation between data and regressors is computed 3. it is converted to an F score then to a p-value

Parameters **X** : array of shape (n_samples, n_features)

the set of regressors sthat will tested sequentially

y : array of shape(n_samples)

the data matrix

center : True, bool,

If true, X and y are centered

Returns **F** : array of shape (m),

the set of F values

pval : array of shape(m)

the set of p-values

1.3.6 Gaussian Processes

Gaussian Processes for Machine Learning (GPML) is a generic supervised learning method primarily designed to solve *regression* problems. It has also been extended to *probabilistic classification*, but in the present implementation, this is only a post-processing of the *regression* exercise.

The advantages of Gaussian Processes for Machine Learning are:

- The prediction interpolates the observations (at least for regular correlation models).
- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals and exceedence probabilities that might be used to refit (online fitting, adaptive fitting) the prediction in some region of interest.
- Versatile: different *linear regression models* and *correlation models* can be specified. Common models are provided, but it is also possible to specify custom models provided they are stationary.

The disadvantages of Gaussian Processes for Machine Learning include:

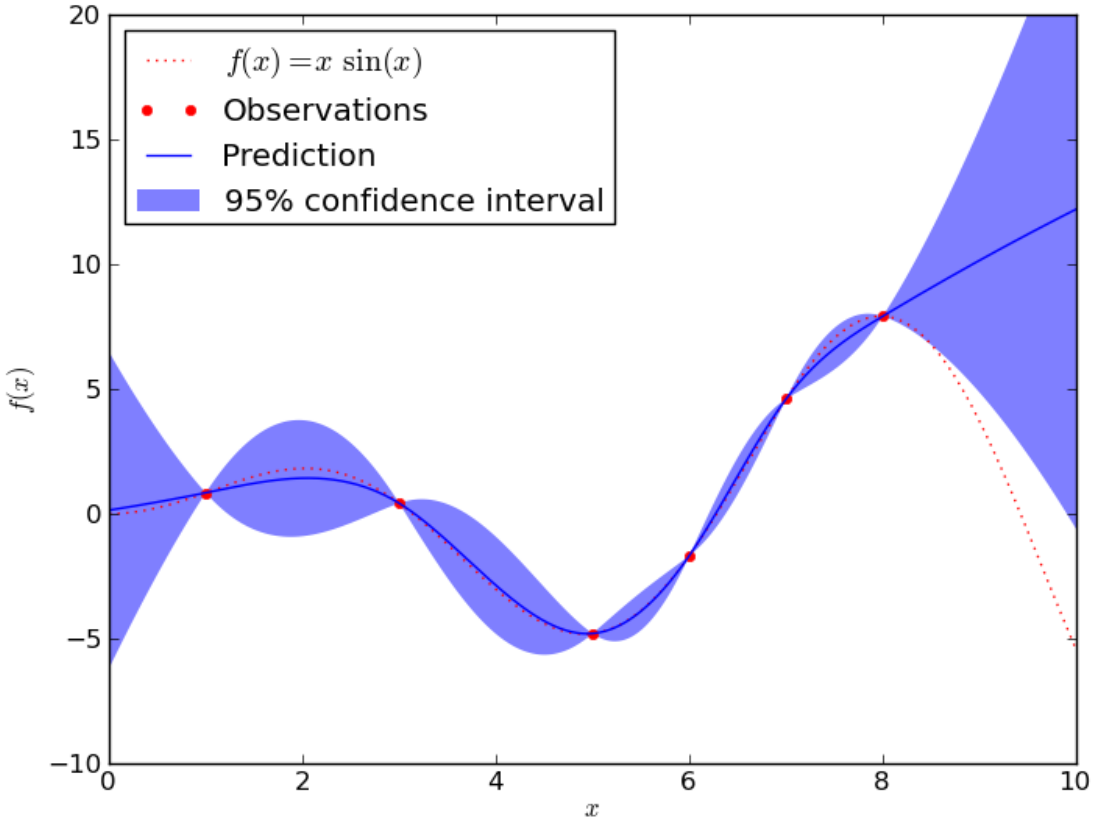
- It is not sparse. It uses the whole samples/features information to perform the prediction.
- It loses efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens. It might indeed give poor performance and it loses computational efficiency.
- Classification is only a post-processing, meaning that one first need to solve a regression problem by providing the complete scalar float precision output y of the experiment one attempt to model.

Thanks to the Gaussian property of the prediction, it has been given varied applications: e.g. for global optimization, probabilistic classification.

An introductory regression example

Say we want to surrogate the function $g(x) = x\sin(x)$. To do so, the function is evaluated onto a design of experiments. Then, we define a GaussianProcess model whose regression and correlation models might be specified using additional kwargs, and ask for the model to be fitted to the data. Depending on the number of parameters provided at instanciation, the fitting procedure may recourse to maximum likelihood estimation for the parameters or alternatively it uses the given parameters.

```
>>> import numpy as np
>>> from scikits.learn import gaussian_process
>>> def f(x):
...     return x * np.sin(x)
>>> X = np.atleast_2d([1., 3., 5., 6., 7., 8.]).T
>>> y = f(X).ravel()
>>> x = np.atleast_2d(np.linspace(0, 10, 1000)).T
>>> gp = gaussian_process.GaussianProcess(theta0=1e-2, thetaL=1e-4, thetaU=1e-1)
>>> gp.fit(X, y)
GaussianProcess(normalize=True, theta0=array([[ 0.01]]),
```


$f(x) = x\ \sin(x)$
Observations
Prediction
95% confidence interval
20
15
10
5
0
−5
−10
0
2
4
6
8
10
$f(x)$
x

```
        optimizer='fmin_cobyla', verbose=False, storage_mode='full',
        nugget=2.2204460492503131e-15, thetaU=array([[ 0.1]]),
        regr=<function constant at 0x...>, random_start=1,
        corr=<function squared_exponential at 0x...>, beta0=None,
        thetaL=array([[ 0.0001]]))
>>> y_pred, sigma2_pred = gp.predict(x, eval_MSE=True)
```

Other examples

- *Gaussian Processes classification example: exploiting the probabilistic output*
- *example_gaussian_process_plot_gp_diabetes_dataset.py*

Mathematical formulation

The initial assumption

Suppose one wants to model the output of a computer experiment, say a mathematical function:

$$g : \mathbb{R}^{n_{\text{features}}} \to \mathbb{R}$$
$$X \mapsto y = g(X)$$

GPML starts with the assumption that this function is *a* conditional sample path of *a* Gaussian process G which is additionally assumed to read as follows:

$$G(X) = f(X)^T\beta + Z(X)$$

where $f(X)^T\beta$ is a linear regression model and $Z(X)$ is a zero-mean Gaussian process with a fully stationary covariance function:

$$C(X, X') = \sigma^2 R(|X - X'|)$$

σ^2 being its variance and R being the correlation function which solely depends on the absolute relative distance between each sample, possibly featurewise (this is the stationarity assumption).

From this basic formulation, note that GPML is nothing but an extension of a basic least squares linear regression problem:

$$g(X) \approx f(X)^T\beta$$

Except we additionaly assume some spatial coherence (correlation) between the samples dictated by the correlation function. Indeed, ordinary least squares assumes the correlation model $R(|X - X'|)$ is one when $X = X'$ and zero otherwise : a *dirac* correlation model – sometimes referred to as a *nugget* correlation model in the kriging literature.

The best linear unbiased prediction (BLUP)

We now derive the *best linear unbiased prediction* of the sample path g conditioned on the observations:

$$\hat{G}(X) = G(X|y_1 = g(X_1), ..., y_{n_{\text{samples}}} = g(X_{n_{\text{samples}}}))$$

It is derived from its *given properties*:

- It is linear (a linear combination of the observations)

$$\hat{G}(X) \equiv a(X)^T y$$

- It is unbiased

$$\mathbb{E}[G(X) - \hat{G}(X)] = 0$$

- It is the best (in the Mean Squared Error sense)

$$\hat{G}(X)^* = \arg\min_{\hat{G}(X)} \mathbb{E}[(G(X) - \hat{G}(X))^2]$$

So that the optimal weight vector $a(X)$ is solution of the following equality constrained optimization problem:

$$a(X)^* = \arg\min_{a(X)} \mathbb{E}[(G(X) - a(X)^T y)^2]$$
$$\text{s.t. } \mathbb{E}[G(X) - a(X)^T y] = 0$$

Rewriting this constrained optimization problem in the form of a Lagrangian and looking further for the first order optimality conditions to be satisfied, one ends up with a closed form expression for the sought predictor – see references for the complete proof.

In the end, the BLUP is shown to be a Gaussian random variate with mean:

$$\mu_{\hat{Y}}(X) = f(X)^T \hat{\beta} + r(X)^T \gamma$$

and variance:

$$\sigma^2_{\hat{Y}}(X) = \sigma^2_Y (1 - r(X)^T R^{-1} r(X) + u(X)^T (F^T R^{-1} F)^{-1} u(X))$$

where we have introduced:

- the correlation matrix whose terms are defined wrt the autocorrelation function and its built-in parameters θ:

$$R_{i\,j} = R(|X_i - X_j|, \theta),\ i,\ j = 1, ..., m$$

- the vector of cross-correlations between the point where the prediction is

made and the points in the DOE:

$$r_i = R(|X - X_i|, \theta),\ i = 1, ..., m$$

- the regression matrix (eg the Vandermonde matrix if f is a polynomial basis):

$$F_{i\,j} = f_i(X_j),\ i = 1, ..., p,\ j = 1, ..., m$$

- the generalized least square regression weights:

$$\hat{\beta} = (F^T R^{-1} F)^{-1} F^T R^{-1} Y$$

- and the vectors:

$$\gamma = R^{-1}(Y - F\,\hat{\beta})$$
$$u(X) = F^T R^{-1} r(X) - f(X)$$

It is important to notice that the probabilistic response of a Gaussian Process predictor is fully analytic and mostly relies on basic linear algebra operations. More precisely the mean prediction is the sum of two simple linear combinations (dot products), and the variance requires two matrix inversions, but the correlation matrix can be decomposed only once using a Cholesky decomposition algorithm.

The empirical best linear unbiased predictor (EBLUP)

Until now, both the autocorrelation and regression models were assumed given. In practice however they are never known in advance so that one has to make (motivated) empirical choices for these models *correlation_models*.

Provided these choices are made, one should estimate the remaining unknown parameters involved in the BLUP. To do so, one uses the set of provided observations in conjunction with some inference technique. The present implementation, which is based on the DACE's Matlab toolbox uses the *maximum likelihood estimation* technique – see DACE manual in references for the complete equations. This maximum likelihood estimation problem is turned into a global optimization problem onto the autocorrelation parameters. In the present implementation, this global optimization is solved by means of the fmin_cobyla optimization function from scipy.optimize. In the case of anisotropy however, we provide an implementation of Welch's componentwise optimization algorithm – see references.

For a more comprehensive description of the theoretical aspects of Gaussian Processes for Machine Learning, please refer to the references below:

References:

- DACE, A Matlab Kriging Toolbox S Lophaven, HB Nielsen, J Sondergaard 2002
- Screening, predicting, and computer experiments WJ Welch, RJ Buck, J Sacks, HP Wynn, TJ Mitchell, and MD Morris Technometrics 34(1) 15–25, 1992
- Gaussian Processes for Machine Learning CE Rasmussen, CKI Williams MIT Press, 2006 (Ed. T Dietrich)
- The design and analysis of computer experiments TJ Santner, BJ Williams, W Notz Springer, 2003

Correlation Models

Common correlation models matches some famous SVM's kernels because they are mostly built on equivalent assumptions. They must fulfill Mercer's conditions and should additionaly remain stationary. Note however, that the choice of the correlation model should be made in agreement with the known properties of the original experiment from which the observations come. For instance:

- If the original experiment is known to be infinitely differentiable (smooth), then one should use the *squared-exponential correlation model*.
- If it's not, then one should rather use the *exponential correlation model*.
- Note also that there exists a correlation model that takes the degree of derivability as input: this is the Matern correlation model, but it's not implemented here (TODO).

For a more detailed discussion on the selection of appropriate correlation models, see the book by Rasmussen & Williams in references.

Regression Models

Common linear regression models involve zero- (constant), first- and second-order polynomials. But one may specify its own in the form of a Python function that takes the features X as input and that returns a vector containing the values of the functional set. The only constraint is that the number of functions must not exceed the number of available observations so that the underlying regression problem is not *underdetermined*.

Implementation details

The present implementation is based on a translation of the DACE Matlab toolbox.

References:

- DACE, A Matlab Kriging Toolbox S Lophaven, HB Nielsen, J Sondergaard 2002,

1.4 Unsupervised learning

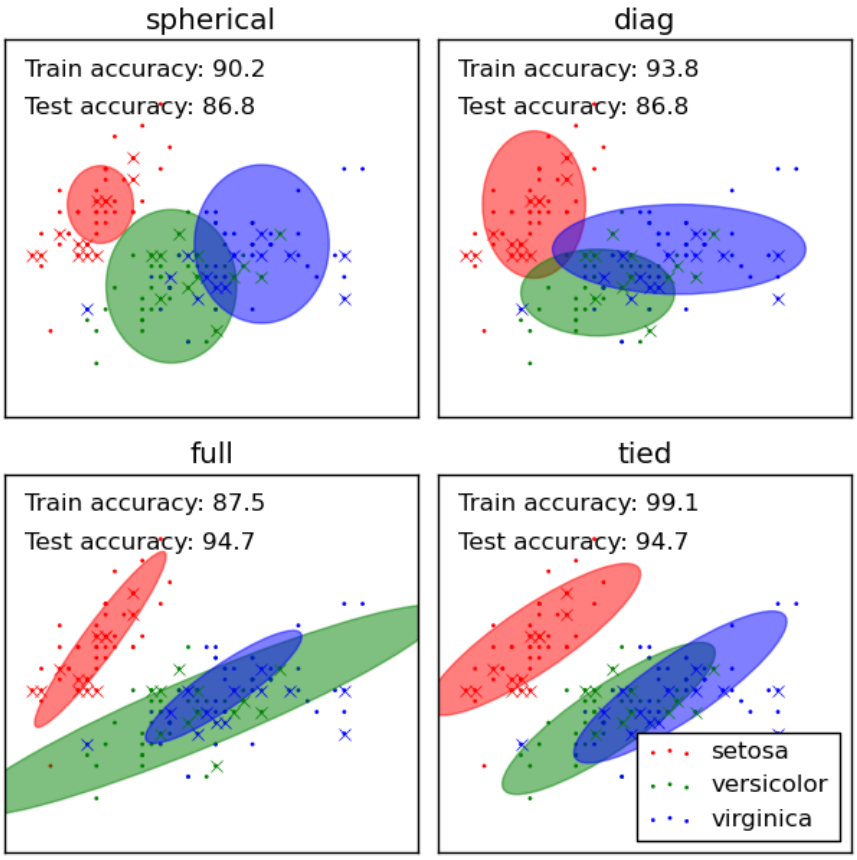
1.4.1 Gaussian mixture models

scikits.learn.mixture is a package which enables to create Mixture Models (diagonal, spherical, tied and full covariance matrices supported), to sample them, and to estimate them from data using Expectation Maximization algorithm. It can also draw confidence ellipsoides for multivariate models, and compute the Bayesian Information Criterion to assess the number of clusters in the data.

For the moment, only Gaussian Mixture Models (GMM) are implemented. These are a class of probabilistic models describing the data as drawn from a mixture of Gaussian probability distributions. The challenge that is GMM tackles is to learn the parameters of these Gaussians from the data.

GMM classifier

The `GMM` object implements a `GMM.fit()` method to learn a Gaussian Mixture Models from train data. Given test data, it can assign to each sample the class of the Gaussian it mostly probably belong to using the `GMM.predict()` method.



Examples:

- See *GMM classification* for an example of using a GMM as a classifier on the iris dataset.
- See *Gaussian Mixture Model Ellipsoids* for an example on plotting the confidence ellipsoids.
- See *Density Estimation for a mixture of Gaussians* for an example on plotting the density estimation.

1.4.2 Clustering

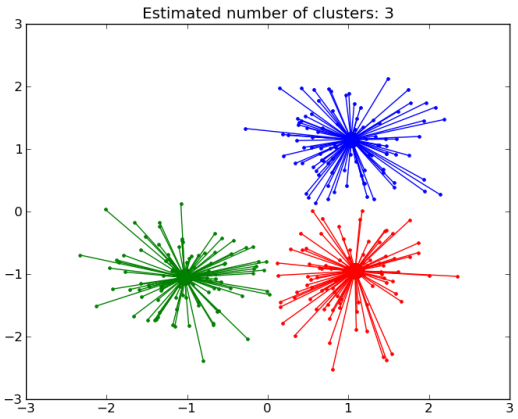
Clustering of unlabeled data can be performed with the module *scikits.learn.cluster*.

Each clustering algorithm comes in two variants: a class, that implements the *fit* method to learn the clusters on train data, and a function, that, given train data, returns an array of integer labels corresponding to the different clusters. For the class, the labels over the training data can be found in the *labels_* attribute.

One important thing to note is that the algorithms implemented in this module take different kinds of matrix as input. On one hand, `MeanShift` and `KMeans` take data matrices of shape [n_samples, n_features]. These can be obtained from the classes in the *scikits.learn.feature_extraction* module. On the other hand, `AffinityPropagation` and `SpectralClustering` take similarity matrices of shape [n_samples, n_samples]. These can be obtained from the functions in the *scikits.learn.metrics.pairwise* module. In other words, `MeanShift` and `KMeans` work with points in a vector space, whereas `AffinityPropagation` and `SpectralClustering` can work with arbitrary objects, as long as a similarity measure exists for such objects.

Affinity propagation

`AffinityPropagation` clusters data by diffusion in the similarity matrix. This algorithm automatically sets its numbers of cluster. It will have difficulties scaling to thousands of samples.

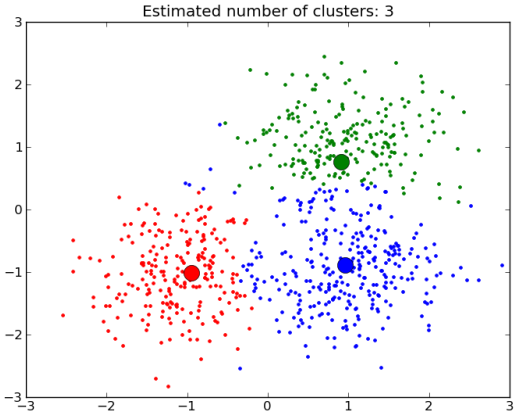


Examples:

- *Demo of affinity propagation clustering algorithm*: Affinity Propagation on a synthetic 2D datasets with 3 classes.
- *Finding structure in the stock market* Affinity Propagation on Financial time series to find groups of companies

Mean Shift

`MeanShift` clusters data by estimating *blobs* in a smooth density of points matrix. This algorithm automatically sets its numbers of cluster. It will have difficulties scaling to thousands of samples.



Examples:

- *A demo of the mean-shift clustering algorithm*: Mean Shift clustering on a synthetic 2D datasets with 3 classes.

K-means

The `KMeans` algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the 'inertia' of the groups. This algorithm requires the number of cluster to be specified. It scales well to large number of samples, however its results may be dependent on an initialisation.

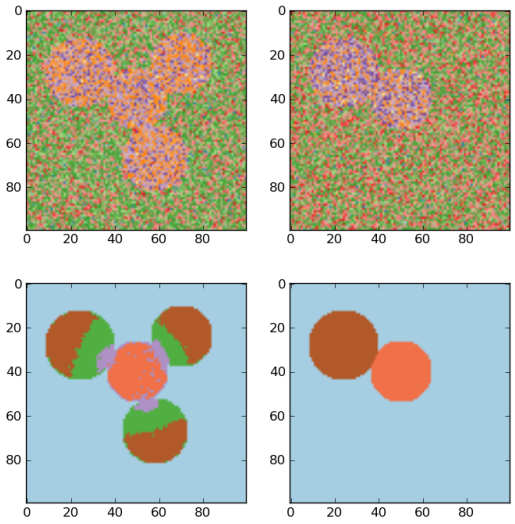
Spectral clustering

`SpectralClustering` does an low-dimension embedding of the affinity matrix between samples, followed by a KMeans in the low dimensional space. It is especially efficient if the affinity matrix is sparse and the pyamg module is installed. SpectralClustering requires the number of clusters to be specified. It works well for a small number of clusters but is not advised when using many clusters.

For two clusters, it solves a convex relaxation of the normalised cuts problem on the similarity graph: cutting the graph in two so that the weight of the edges cut is small compared to the weights in of edges inside each cluster. This criteria is especially interesting when working on images: graph vertices are pixels, and edges of the similarity graph are a function of the gradient of the image.

Examples:

- *Segmenting the picture of Lena in regions*: Spectral clustering to split the image of lena in regions.
- *Spectral clustering for image segmentation*: Segmenting objects from a noisy background using spectral clustering.



1.4.3 Decomposing signals in components (matrix factorization problems)

Principal component analysis (PCA)

Exact PCA and probabilistic interpretation

PCA is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance. In the scikit-learn, `PCA` is implemented as a *transformer* object that learns n components in its *fit* method, and can be used on new data to project it on these components.

The optional parameter *whiten=True* parameter make it possible to project the data onto the singular space while scaling each component to unit variance. This is often useful if the models down-stream make strong assumptions on the isotropy of the signal: this is for example the case for Support Vector Machines with the RBF kernel and the K-Means clustering algorithm. However in that case the inverse transform is no longer exact since some information is lost while forward transforming.

In addition, the `ProbabilisticPCA` object provides a probabilistic interpretation of the PCA that can give a likelihood of data based on the amount of variance it explains. As such it implements a *score* method that can be used in cross-validation.

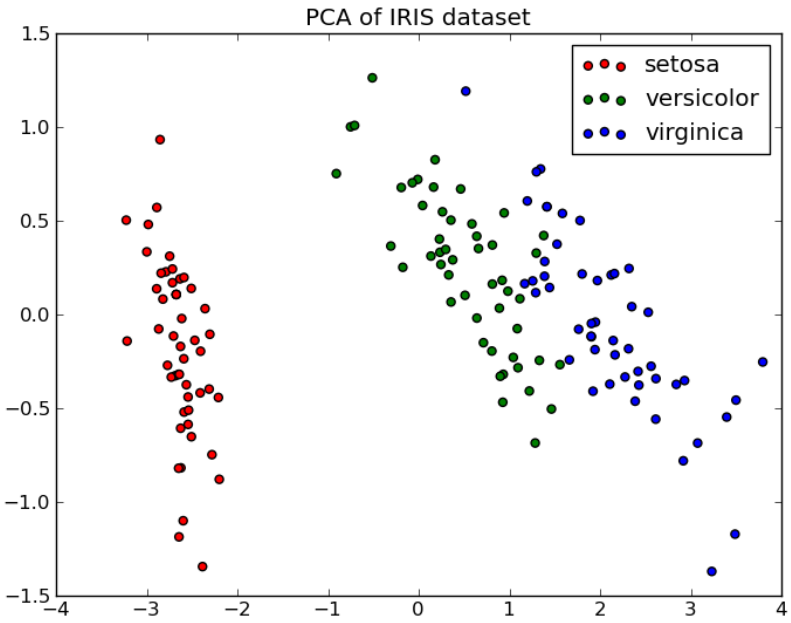
Below is an example of the iris dataset, which is comprised of 4 features, projected on the 2 dimensions that explain most variance:

Examples:

- *PCA 2D projection of Iris dataset*

Approximate PCA

Often we are interested in projecting the data onto a lower dimensional space that preseves most of the variance by droping the singular vector of components associated with lower singular values.



For instance for face recognition, if we work with 64x64 gray level pixel pictures the dimensionality of the data is 4096 and it is slow to train a RBF Support Vector Machine on such wide data. Furthermore we know that intrinsic dimensionality of the data is much lower than 4096 since all faces pictures look alike. The samples lie on a manifold of much lower dimension (say around 200 for instance). The PCA algorithm can be used to linearly transform the data while both reducing the dimensionality and preserve most of the explained variance at the same time.

The class `RandomizedPCA` is very useful in that case: since we are going to drop most of the singular vectors it is much more efficient to limit the computation to an approximated estimate of the singular vectors we will keep to actually perform the transform.

`RandomizedPCA` can hence be used as a drop in replacement for `PCA` minor the exception that we need to give it the size of the lower dimensional space *n_components* as mandatory input parameter.

If we note $n_{max} = max(n_{samples}, n_{features})$ and $n_{min} = min(n_{samples}, n_{features})$, the time complexity of `RandomizedPCA` is $O(n_{max}^2 \cdot n_{components})$ instead of $O(n_{max}^2 \cdot n_{min})$ for the exact method implemented in `PCA`.

The memory footprint of `RandomizedPCA` is also proportional to $2 \cdot n_{max} \cdot n_{components}$ instead of $n_{max} \cdot n_{min}$ for the exact method.

Furthermore `RandomizedPCA` is able to work with *scipy.sparse* matrices as input which make it suitable for reducing the dimensionality of features extracted from text documents for instance.

Note: the implementation of *inverse_transform* in `RandomizedPCA` is not the exact inverse transform of *transform* even when *whiten=False* (default).

Examples:

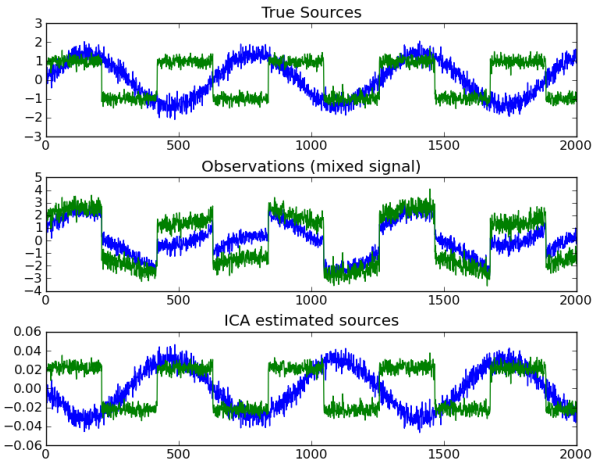
- *Faces recognition example using eigenfaces and SVMs*

References:

- "Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions" Halko, et al., 2009

Independent component analysis (ICA)

ICA finds components that are maximally independent. It is classically used to separate mixed signals (a problem know as *blind source separation*), as in the example below:



Examples:

- *Blind source separation using FastICA*
- *FastICA on 2D point clouds*

1.5 Model Selection

1.5.1 Cross-Validation

Learning the parameters of a prediction function and testing it on the same data yields a methodological bias. To avoid over-fitting, we have to define two different sets : a *learning set* X^l, y^l which is used for learning the prediction function (also called *training set*), and a *test set* X^t, y^t which is used for testing the prediction function. However, by defining these two sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular couple of *learning set* and *test set*.

A solution is to split the whole data in different learning set and test set, and to return the the averaged value of the prediction scores obtained with the different sets. Such a procedure is called *cross-validation*. This approach can be computationally expensive, but does not waste too much data (as it is the case when fixing an arbitrary test set), which is a major advantage in problem such as inverse inference where the number of samples is very small.

Examples

Receiver operating characteristic (ROC) with cross validation, *Parameter estimation using grid search with a nested cross-validation*, *example_rfe_with_cross_validation.py*,

Leave-One-Out - LOO

`LeaveOneOut` The *Leave-One-Out* (or LOO) is a simple cross-validation. Each learning set is created by taking all the samples except one, the test set being the sample left out. Thus, for n samples, we have n different learning sets and n different tests set. This cross-validation procedure does not waste much data as only one sample is removed from the learning set.

```
>>> import numpy as np
>>> from scikits.learn.cross_val import LeaveOneOut
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> Y = np.array([0, 1, 0, 1])
>>> loo = LeaveOneOut(len(Y))
>>> print loo
scikits.learn.cross_val.LeaveOneOut(n=4)
>>> for train, test in loo: print train, test
[False  True  True  True] [ True False False False]
[ True False  True  True] [False  True False False]
[ True  True False  True] [False False  True False]
[ True  True  True False] [False False False  True]
```

Each fold is constitued by two arrays: the first one is related to the *training set*, and the second one to the *test set*. Thus, one can create the training/test sets using:

```
>>> X_train, X_test, y_train, y_test = X[train], X[test], Y[train], Y[test]
```

If X or Y are *scipy.sparse* matrices, train and test need to be integer indices. It can be obtained by setting the parameter indices to True when creating the cross-validation procedure.

```
>>> import numpy as np
>>> from scikits.learn.cross_val import LeaveOneOut
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> Y = np.array([0, 1, 0, 1])
>>> loo = LeaveOneOut(len(Y), indices=True)
>>> print loo
scikits.learn.cross_val.LeaveOneOut(n=4)
>>> for train, test in loo: print train, test
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

Leave-P-Out - LPO

`LeavePOut` *Leave-P-Out* is very similar to *Leave-One-Out*, as it creates all the possible training/test sets by removing P samples from the complete set.

Example of Leave-2-Out:

```
>>> from scikits.learn.cross_val import LeavePOut
>>> X = [[0., 0.], [1., 1.], [-1., -1.], [2., 2.]]
>>> Y = [0, 1, 0, 1]
```

```
>>> loo = LeavePOut(len(Y), 2)
>>> print loo
scikits.learn.cross_val.LeavePOut(n=4, p=2)
>>> for train, test in loo: print train,test
[False False  True  True] [ True  True False False]
[False  True False  True] [ True False  True False]
[False  True  True False] [ True False False  True]
[ True False False  True] [False  True  True False]
[ True False  True False] [False  True False  True]
[ True  True False False] [False False  True  True]
```

All the possible folds are created, and again, one can create the training/test sets using:

```
>>> import numpy as np
>>> X = np.asanyarray(X)
>>> Y = np.asanyarray(Y)
>>> X_train, X_test, y_train, y_test = X[train], X[test], Y[train], Y[test]
```

K-fold

`KFold`

The *K-fold* divides all the samples in K groups of samples, called folds (if $K = n$, we retrieve the *LOO*), of equal sizes (if possible). The prediction function is learned using *K - 1* folds, and the fold left out is used for test.

Example of 2-fold:

```
>>> from scikits.learn.cross_val import KFold
>>> X = [[0., 0.], [1., 1.], [-1., -1.], [2., 2.]]
>>> Y = [0, 1, 0, 1]
>>> loo = KFold(len(Y), 2)
>>> print loo
scikits.learn.cross_val.KFold(n=4, k=2)
>>> for train, test in loo: print train,test
[False False  True  True] [ True  True False False]
[ True  True False False] [False False  True  True]
```

Stratified K-Fold

`StratifiedKFold`

The *Stratified K-Fold* is a variation of *K-fold*, which returns stratified folds, *i.e* which creates folds by preserving the same percentage for each class than in the complete set.

Example of stratified 2-fold:

```
>>> from scikits.learn.cross_val import StratifiedKFold
>>> X = [[0., 0.], [1., 1.], [-1., -1.], [2., 2.], [3., 3.], [4., 4.], [0., 1.]]
>>> Y = [0, 0, 0, 1, 1, 1, 0]
>>> skf = StratifiedKFold(Y, 2)
>>> print skf
scikits.learn.cross_val.StratifiedKFold(labels=[0 0 0 1 1 1 0], k=2)
>>> for train, test in skf: print train, test
[False  True False False  True False  True] [ True False  True  True False  True False]
[ True False  True  True False  True False] [False  True False False  True False  True]
```

Leave-One-Label-Out - LOLO

`LeaveOneLabelOut`

The *Leave-One-Label-Out* (LOLO) is a cross-validation scheme which removes the samples according to a specific label. Each training set is thus constitued by all the samples except the ones related to a specific label.

For example, in the cases of multiple experiments, *LOLO* can be used to create a cross-validation based on the different experiments: we create a training set using the samples of all the experiments except one.

```
>>> from scikits.learn.cross_val import LeaveOneLabelOut
>>> X = [[0., 0.], [1., 1.], [-1., -1.], [2., 2.]]
>>> Y = [0, 1, 0, 1]
>>> labels = [1, 1, 2, 2]
>>> loo = LeaveOneLabelOut(labels)
>>> print loo
scikits.learn.cross_val.LeaveOneLabelOut(labels=[1, 1, 2, 2])
>>> for train, test in loo: print train,test
[False False  True  True] [ True  True False False]
[ True  True False False] [False False  True  True]
```

Leave-P-Label-Out

`LeavePLabelOut`

Leave-P-Label-Out is similar as *Leave-One-Label-Out*, but removes samples related to *P* labels for each training/test set.

Example of Leave-2-Label Out:

```
>>> from scikits.learn.cross_val import LeavePLabelOut
>>> X = [[0., 0.], [1., 1.], [-1., -1.], [2., 2.], [3., 3.], [4., 4.]]
>>> Y = [0, 1, 0, 1, 0, 1]
>>> labels = [1, 1, 2, 2, 3, 3]
>>> loo = LeavePLabelOut(labels, 2)
>>> print loo
scikits.learn.cross_val.LeavePLabelOut(labels=[1, 1, 2, 2, 3, 3], p=2)
>>> for train, test in loo: print train,test
[False False False False  True  True] [ True  True  True  True False False]
[False False  True  True False False] [ True  True False False  True  True]
[ True  True False False False False] [False False  True  True  True  True]
```

1.5.2 Grid Search

Grid Search is used to optimize the parameters of a model (e.g. Support Vector Classifier, Lasso, etc.) using cross-validation.

Main class is `GridSearchCV`.

Examples

See *Parameter estimation using grid search with a nested cross-validation* for an example of Grid Search computation on the digits dataset.

See *Sample pipeline for text feature extraction and evaluation* for an example of Grid Search coupling parameters from a text documents feature extractor (n-gram count vectorizer and TF-IDF transformer) with a classifier (here a linear SVM trained with SGD with either elastic net or L2 penalty).

Notes

Computations can be run in parallel if your OS supports it, by using the keyword n_jobs=-1, see function signature for more details.

1.6 Class reference

1.6.1 Support Vector Machines

svm.SVC([C, kernel, degree, gamma, coef0, ...])	C-Support Vector Classification.
svm.LinearSVC([penalty, loss, dual, eps, C, ...])	Linear Support Vector Classification.
svm.NuSVC([nu, kernel, degree, gamma, ...])	Nu-Support Vector Classification.
svm.SVR([kernel, degree, gamma, coef0, ...])	Support Vector Regression.
svm.NuSVR([nu, C, kernel, degree, gamma, ...])	Nu Support Vector Regression.
svm.OneClassSVM([kernel, degree, gamma, ...])	Unsupervised Outliers Detection.

scikits.learn.svm.SVC

class scikits.learn.svm.**SVC**(*C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *eps=0.001*, *cache_size=100.0*)

C-Support Vector Classification.

Parameters **C** : float, optional (default=1.0)

penalty parameter C of the error term.

kernel : string, optional

Specifies the kernel type to be used in the algorithm. one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. If none is given 'rbf' will be used.

degree : int, optional

degree of kernel function is significant only in poly, rbf, sigmoid

gamma : float, optional

kernel coefficient for rbf and poly, by default 1/n_features will be taken.

coef0 : float, optional

independent term in kernel function. It is only significant in poly/sigmoid.

probability: boolean, optional (False by default) :

enable probability estimates. This must be enabled prior to calling prob_predict.

shrinking: boolean, optional :

wether to use the shrinking heuristic.

eps: float, optional :

precision for stopping criteria

cache_size: float, optional :

specify the size of the cache (in MB)

See Also:

SVR, LinearSVC

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from scikits.learn.svm import SVC
>>> clf = SVC()
>>> clf.fit(X, y)
SVC(kernel='rbf', C=1.0, probability=False, degree=3, coef0=0.0, eps=0.001,
  cache_size=100.0, shrinking=True, gamma=0.25)
>>> print clf.predict([[-0.8, -1]])
[ 1.]
```

Attributes

*support_*	array-like, shape = [n_SV]	Index of support vectors.
support_vectors	array-like, shape = [n_SV, n_features]	Support vectors.
*n_support_*	array-like, dtype=int32, shape = [n_class]	number of support vector for each class.
*dual_coef_*	array, shape = [n_class-1, n_SV]	Coefficients of the support vector in the decision function.
*coef_*	array, shape = [n_class-1, n_features]	Weights asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.
*intercept_*	array, shape = [n_class * (n_class-1) / 2]	Constants in decision function.

Methods

decision_function
fit
predict
predict_log_proba
predict_proba
score

__init__(*C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *eps=0.001*, *cache_size=100.0*)

decision_function(*T*)

Calculate the distance of the samples T to the separating hyperplane.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit (*X*, *y*, *class_weight={}*, *sample_weight=[]*, ***params*)

Fit the SVM model according to the given training data and parameters.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training vectors, where n_samples is the number of samples and n_features is the number of features.

y : array-like, shape = [n_samples]

Target values (integers in classification, real numbers in regression)

class_weight : dict | 'auto', optional

Weights associated with classes in the form {class_label : weight}. If not given, all classes are supposed to have weight one.

The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

sample_weight : array-like, shape = [n_samples], optional

Weights applied to individual samples (1. for unweighted).

Returns **self** : object

Returns self.

predict (*X*)

This function does classification or regression on an array of test vectors X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the function value of X calculated is returned.

For an one-class model, +1 or -1 is returned.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba (*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

predict_proba (*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

score (*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

scikits.learn.svm.LinearSVC

class scikits.learn.svm.**LinearSVC**(*penalty='l2'*, *loss='l2'*, *dual=True*, *eps=0.0001*, *C=1.0*, *multi_class=False*, *fit_intercept=True*, *intercept_scaling=1*)

Linear Support Vector Classification.

Similar to SVC with parameter kernel='linear', but uses internally liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should be faster for huge datasets.

Parameters **loss** : string, 'l1' or 'l2' (default 'l2')

Specifies the loss function. With 'l1' it is the standard SVM loss (a.k.a. hinge Loss) while with 'l2' it is the squared loss. (a.k.a. squared hinge Loss)

penalty : string, 'l1' or 'l2' (default 'l2')

Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to **coef_** vectors that are sparse.

dual : bool, (default True)

Select the algorithm to either solve the dual or primal optimization problem.

eps: float, optional :

precision for stopping criteria

multi_class: boolean, optional :

perform multi-class SVM by Cramer and Singer. If active, options loss, penalty and dual will be ignored.

intercept_scaling : float, default: 1

when self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased

See Also:

SVC

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller eps parameter.

References

LIBLINEAR – A Library for Large Linear Classification http://www.csie.ntu.edu.tw/~cjlin/liblinear/

Attributes

*coef_*	array, shape = [n_features] if n_classes == 2 else [n_classes, n_features]	Weights asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.
*intercept_*	array, shape = [1] if n_classes == 2 else [n_classes]	Constants in decision function.

Methods

decision_function
fit
predict
predict_proba
score
transform

__init__ (*penalty='l2'*, *loss='l2'*, *dual=True*, *eps=0.0001*, *C=1.0*, *multi_class=False*, *fit_intercept=True*, *intercept_scaling=1*)

decision_function (*X*)

Return the decision function of X according to the trained model.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class]

Returns the decision function of the sample for each class in the model.

fit (*X*, *y*, *class_weight={}*, ***params*)

Fit the model according to the given training data and parameters.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training vector, where n_samples in the number of samples and n_features is the number of features.

y : array-like, shape = [n_samples]

Target vector relative to X

class_weight : dict , {class_label

Weights associated with classes. If not given, all classes are supposed to have weight one.

Returns **self** : object

Returns self.

predict(*X*)

Predict target values of X according to the fitted model.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

score(*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

scikits.learn.svm.NuSVC

class scikits.learn.svm.**NuSVC**(*nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *eps=0.001*, *cache_size=100.0*)

Nu-Support Vector Classification.

Parameters **nu** : float, optional

An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken.

kernel : string, optional

Specifies the kernel type to be used in the algorithm. one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. If none is given 'rbf' will be used.

degree : int, optional

degree of kernel function is significant only in poly, rbf, sigmoid

gamma : float, optional

kernel coefficient for rbf and poly, by default 1/n_features will be taken.

probability: boolean, optional (False by default) :

enable probability estimates. This must be enabled prior to calling prob_predict.

coef0 : float, optional

independent term in kernel function. It is only significant in poly/sigmoid.

shrinking: boolean, optional :

wether to use the shrinking heuristic.

eps: float, optional :

precision for stopping criteria

cache_size: float, optional :

specify the size of the cache (in MB)

See Also:

SVC, LinearSVC, SVR

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from scikits.learn.svm import NuSVC
>>> clf = NuSVC()
>>> clf.fit(X, y)
NuSVC(kernel='rbf', probability=False, degree=3, coef0=0.0, eps=0.001,
   cache_size=100.0, shrinking=True, nu=0.5, gamma=0.25)
>>> print clf.predict([[-0.8, -1]])
[ 1.]
```

Attributes

*support_*	array-like, shape = [n_SV]	Index of support vectors.
support_vectors	array-like, shape = [n_SV, n_features]	Support vectors.
*n_support_*	array-like, dtype=int32, shape = [n_class]	number of support vector for each class.
*dual_coef_*	array, shape = [n_classes-1, n_SV]	Coefficients of the support vector in the decision function.
*coef_*	array, shape = [n_classes-1, n_features]	Weights asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.
*intercept_*	array, shape = [n_class * (n_class-1) / 2]	Constants in decision function.

Methods

fit(X, y)	self	Fit the model
predict(X)	array	Predict using the model.
predict_proba(X)	array	Return probability estimates.
predict_log_proba(X)	array	Return log-probability estimates.
decision_function(X)	array	Return distance to predicted margin.

__init__(*nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *eps=0.001*, *cache_size=100.0*)

decision_function(*T*)

Calculate the distance of the samples T to the separating hyperplane.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit (*X*, *y*, *class_weight={}*, *sample_weight=[]*, ***params*)

Fit the SVM model according to the given training data and parameters.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training vectors, where n_samples is the number of samples and n_features is the number of features.

y : array-like, shape = [n_samples]

Target values (integers in classification, real numbers in regression)

class_weight : dict | 'auto', optional

Weights associated with classes in the form {class_label : weight}. If not given, all classes are supposed to have weight one.

The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

sample_weight : array-like, shape = [n_samples], optional

Weights applied to individual samples (1. for unweighted).

Returns **self** : object

Returns self.

predict (*X*)

This function does classification or regression on an array of test vectors X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the function value of X calculated is returned.

For an one-class model, +1 or -1 is returned.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba (*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

predict_proba (*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

score(*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

scikits.learn.svm.SVR

class scikits.learn.svm.**SVR**(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *cache_size=100.0*, *eps=0.001*, *C=1.0*, *nu=0.5*, *p=0.10000000000000001*, *shrinking=True*, *probability=False*)

Support Vector Regression.

Parameters **nu** : float, optional

An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken. Only available if impl='nu_svc'

kernel : string, optional

Specifies the kernel type to be used in the algorithm. one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. If none is given 'rbf' will be used.

p : float

epsilon in the epsilon-SVR model.

degree : int, optional

degree of kernel function is significant only in poly, rbf, sigmoid

gamma : float, optional

kernel coefficient for rbf and poly, by default 1/n_features will be taken.

C : float, optional (default=1.0)

penalty parameter C of the error term.

probability: boolean, optional (False by default) :

enable probability estimates. This must be enabled prior to calling prob_predict.

eps: float, optional :

precision for stopping criteria

coef0 : float, optional

independent term in kernel function. It is only significant in poly/sigmoid.

cache_size: float, optional :

specify the size of the cache (in MB)

shrinking: boolean, optional :

wether to use the shrinking heuristic.

See Also:

NuSVR

Attributes

*support_*	array-like, shape = [n_SV]	Index of support vectors.
support_vectors	array-like, shape = [nSV, n_features]	Support vectors.
*dual_coef_*	array, shape = [n_classes-1, n_SV]	Coefficients of the support vector in the decision function.
*coef_*	array, shape = [n_classes-1, n_features]	Weights asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.
*intercept_*	array, shape = [n_class * (n_class-1) / 2]	Constants in decision function.

Methods

decision_function
fit
predict
predict_log_proba
predict_proba
score

__init__(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *cache_size=100.0*, *eps=0.001*, *C=1.0*, *nu=0.5*, *p=0.10000000000000001*, *shrinking=True*, *probability=False*)

decision_function(*T*)

Calculate the distance of the samples T to the separating hyperplane.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit(*X*, *y*, *sample_weight=[]*)

Fit the SVM model according to the given training data and parameters.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training vector, where n_samples is the number of samples and n_features is the number of features.

y : array, shape = [n_samples]

Target values. Array of floating-point numbers.

Returns **self** : object

Returns self.

predict(*X*)

This function does classification or regression on an array of test vectors X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the function value of X calculated is returned.

For an one-class model, +1 or -1 is returned.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

predict_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

score(*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.svm.NuSVR

class scikits.learn.svm.**NuSVR**(*nu=0.5*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *cache_size=100.0*, *eps=0.001*)

Nu Support Vector Regression.

Similar to NuSVC, for regression, uses a paramter nu to control the number of support vectors. However, unlike NuSVC, where nu replaces with C, here nu replaces with the parameter p of SVR.

Parameters **nu** : float, optional

An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken. Only available if impl='nu_svc'

C : float, optional (default=1.0)

penalty parameter C of the error term.

kernel : string, optional

Specifies the kernel type to be used in the algorithm. one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. If none is given 'rbf' will be used.

degree : int, optional

degree of kernel function is significant only in poly, rbf, sigmoid

gamma : float, optional

kernel coefficient for rbf and poly, by default 1/n_features will be taken.

eps: float, optional :

precision for stopping criteria

probability: boolean, optional (False by default) :

enable probability estimates. This must be enabled prior to calling prob_predict.

coef0 : float, optional

independent term in kernel function. It is only significant in poly/sigmoid.

shrinking: boolean, optional :

wether to use the shrinking heuristic.

cache_size: float, optional :

specify the size of the cache (in MB)

See Also:

NuSVR

Attributes

*support_*	array-like, shape = [n_SV]	Index of support vectors.
support_vectors	array-like, shape = [nSV, n_features]	Support vectors.
*dual_coef_*	array, shape = [n_classes-1, n_SV]	Coefficients of the support vector in the decision function.
*coef_*	array, shape = [n_classes-1, n_features]	Weights asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.
*intercept_*	array, shape = [n_class * (n_class-1) / 2]	Constants in decision function.

Methods

```
decision_function
fit
predict
predict_log_proba
predict_proba
score
```

__init__ (*nu=0.5*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *cache_size=100.0*, *eps=0.001*)

decision_function (*T*)

Calculate the distance of the samples T to the separating hyperplane.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit (*X*, *y*)

Fit the SVM model according to the given training data and parameters.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training vector, where n_samples is the number of samples and n_features is the number of features.

y : array, shape = [n_samples]

Target values. Array of floating-point numbers.

Returns **self** : object

Returns self.

predict (*X*)

This function does classification or regression on an array of test vectors X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the function value of X calculated is returned.

For an one-class model, +1 or -1 is returned.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

predict_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

score(*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.svm.OneClassSVM

class scikits.learn.svm.**OneClassSVM**(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *cache_size=100.0*, *eps=0.001*, *nu=0.5*, *shrinking=True*)

Unsupervised Outliers Detection.

Estimate the support of a high-dimensional distribution.

Parameters **kernel** : string, optional

Specifies the kernel type to be used in the algorithm. Can be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. If none is given 'rbf' will be used.

nu : float, optional

An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken.

degree : int, optional

Degree of kernel function. Significant only in poly, rbf, sigmoid.

gamma : float, optional

kernel coefficient for rbf and poly, by default 1/n_features will be taken.

coef0 : float, optional

Independent term in kernel function. It is only significant in poly/sigmoid.

eps: float, optional :

precision for stopping criteria

shrinking: boolean, optional :

wether to use the shrinking heuristic.

cache_size: float, optional :

specify the size of the cache (in MB)

Attributes

*support_*	array-like, shape = [n_SV]	Index of support vectors.
*support_vectors_*	array-like, shape = [nSV, n_features]	Support vectors.
*dual_coef_*	array, shape = [n_classes-1, n_SV]	Coefficient of the support vector in the decision function.
*coef_*	array, shape = [n_classes-1, n_features]	Weights asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.
*intercept_*	array, shape = [n_classes-1]	Constants in decision function.

Methods

```
decision_function
fit
predict
predict_log_proba
predict_proba
```

__init__(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *cache_size=100.0*, *eps=0.001*, *nu=0.5*, *shrinking=True*)

decision_function(*T*)

Calculate the distance of the samples T to the separating hyperplane.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit (*X*, *class_weight={}*, *sample_weight=[ ]*, ***params*)
Detects the soft boundary of the set of samples X.

Parameters **X** : array-like, shape = [n_samples, n_features]

Set of samples, where n_samples is the number of samples and n_features is the number of features.

Returns **self** : object

Returns self.

predict (*X*)
This function does classification or regression on an array of test vectors X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the function value of X calculated is returned.

For an one-class model, +1 or -1 is returned.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba (*T*)
This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

predict_proba (*T*)
This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

For sparse data

svm.sparse.SVC([kernel, degree, gamma, ...])	SVC for sparse matrices (csr).
svm.sparse.NuSVC([nu, kernel, degree, ...])	NuSVC for sparse matrices (csr).
svm.sparse.SVR([kernel, degree, gamma, ...])	SVR for sparse matrices (csr)
svm.sparse.NuSVR([nu, C, kernel, degree, ...])	NuSVR for sparse matrices (csr)
svm.sparse.OneClassSVM([kernel, degree, ...])	NuSVR for sparse matrices (csr)
svm.sparse.LinearSVC([penalty, loss, dual, ...])	Linear Support Vector Classification, Sparse Version

scikits.learn.svm.sparse.SVC

class scikits.learn.svm.sparse.**SVC**(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *cache_size=100.0*, *eps=0.001*, *C=1.0*, *shrinking=True*, *probability=False*)

SVC for sparse matrices (csr).

See scikits.learn.svm.SVC for a complete list of parameters

Notes

For best results, this accepts a matrix in csr format (scipy.sparse.csr), but should be able to convert from any array-like object (including other sparse representations).

Methods

decision_function
fit
predict
predict_log_proba
predict_proba
score

__init__(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *cache_size=100.0*, *eps=0.001*, *C=1.0*, *shrinking=True*, *probability=False*)

decision_function(*T*)

Calculate the distance of the samples T to the separating hyperplane.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit(*X*, *y*, *class_weight={}*, *sample_weight=[]*, ***params*)

Fit the SVM model according to the given training data and parameters.

Parameters **X** : sparse matrix, shape = [n_samples, n_features]

Training vectors, where n_samples is the number of samples and n_features is the number of features.

y : array-like, shape = [n_samples]

Target values (integers in classification, real numbers in regression)

class_weight : dict | 'auto', optional

Weights associated with classes in the form {class_label : weight}. If not given, all classes are supposed to have weight one.

The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

sample_weight : array-like, shape = [n_samples], optional

Weights applied to individual samples (1. for unweighted).

Returns **self** : object

Returns an instance of self.

Notes

For maximum effiency, use a sparse matrix in csr format (scipy.sparse.csr_matrix)

predict(*T*)

This function does classification or regression on an array of test vectors T.

For a classification model, the predicted class for each sample in T is returned. For a regression model, the function value of T calculated is returned.

For an one-class model, +1 or -1 is returned.

Parameters **T** : scipy.sparse.csr, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

predict_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

score(*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

scikits.learn.svm.sparse.NuSVC

class scikits.learn.svm.sparse.**NuSVC**(*nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *eps=0.001*, *cache_size=100.0*)

NuSVC for sparse matrices (csr).

See scikits.learn.svm.NuSVC for a complete list of parameters

Notes

For best results, this accepts a matrix in csr format (scipy.sparse.csr), but should be able to convert from any array-like object (including other sparse representations).

Methods

decision_function
fit
predict
predict_log_proba
predict_proba
score

__init__(*nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *eps=0.001*, *cache_size=100.0*)

decision_function(*T*)

Calculate the distance of the samples T to the separating hyperplane.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit(*X*, *y*, *class_weight={}*, *sample_weight=[]*, ***params*)

Fit the SVM model according to the given training data and parameters.

Parameters **X** : sparse matrix, shape = [n_samples, n_features]

Training vectors, where n_samples is the number of samples and n_features is the number of features.

y : array-like, shape = [n_samples]

Target values (integers in classification, real numbers in regression)

class_weight : dict | 'auto', optional

Weights associated with classes in the form {class_label : weight}. If not given, all classes are supposed to have weight one.

The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

sample_weight : array-like, shape = [n_samples], optional

Weights applied to individual samples (1. for unweighted).

Returns **self** : object

Returns an instance of self.

Notes

For maximum effiency, use a sparse matrix in csr format (scipy.sparse.csr_matrix)

predict(*T*)

This function does classification or regression on an array of test vectors T.

For a classification model, the predicted class for each sample in T is returned. For a regression model, the function value of T calculated is returned.

For an one-class model, +1 or -1 is returned.

Parameters **T** : scipy.sparse.csr, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

predict_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

score(*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

scikits.learn.svm.sparse.SVR

class scikits.learn.svm.sparse.**SVR**(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *cache_size=100.0*, *eps=0.001*, *C=1.0*, *nu=0.5*, *p=0.10000000000000001*, *shrinking=True*, *probability=False*)

SVR for sparse matrices (csr)

See scikits.learn.svm.SVR for a complete list of parameters

Notes

For best results, this accepts a matrix in csr format (scipy.sparse.csr), but should be able to convert from any array-like object (including other sparse representations).

Methods

decision_function
fit
predict
predict_log_proba
predict_proba
score

__init__(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *cache_size=100.0*, *eps=0.001*, *C=1.0*, *nu=0.5*, *p=0.10000000000000001*, *shrinking=True*, *probability=False*)

decision_function(*T*)

Calculate the distance of the samples T to the separating hyperplane.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit (*X*, *y*, *class_weight={}*, *sample_weight=[ ]*, ***params*)
Fit the SVM model according to the given training data and parameters.

Parameters **X** : sparse matrix, shape = [n_samples, n_features]

Training vectors, where n_samples is the number of samples and n_features is the number of features.

y : array-like, shape = [n_samples]

Target values (integers in classification, real numbers in regression)

class_weight : dict | 'auto', optional

Weights associated with classes in the form {class_label : weight}. If not given, all classes are supposed to have weight one.

The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

sample_weight : array-like, shape = [n_samples], optional

Weights applied to individual samples (1. for unweighted).

Returns **self** : object

Returns an instance of self.

Notes

For maximum effiency, use a sparse matrix in csr format (scipy.sparse.csr_matrix)

predict (*T*)
This function does classification or regression on an array of test vectors T.

For a classification model, the predicted class for each sample in T is returned. For a regression model, the function value of T calculated is returned.

For an one-class model, +1 or -1 is returned.

Parameters **T** : scipy.sparse.csr, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba (*T*)
This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

predict_proba(*T*)
This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

score(*X*, *y*)
Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.svm.sparse.NuSVR

class scikits.learn.svm.sparse.**NuSVR**(*nu=0.5*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *cache_size=100.0*, *eps=0.001*)
NuSVR for sparse matrices (csr)

See scikits.learn.svm.NuSVC for a complete list of parameters

Notes

For best results, this accepts a matrix in csr format (scipy.sparse.csr), but should be able to convert from any array-like object (including other sparse representations).

Methods

decision_function
fit
predict
predict_log_proba
predict_proba
score

__init__(*nu=0.5*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*, *cache_size=100.0*, *eps=0.001*)

decision_function(*T*)
Calculate the distance of the samples T to the separating hyperplane.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

fit (*X*, *y*, *class_weight={}*, *sample_weight=[]*, ***params*)

Fit the SVM model according to the given training data and parameters.

Parameters **X** : sparse matrix, shape = [n_samples, n_features]

Training vectors, where n_samples is the number of samples and n_features is the number of features.

y : array-like, shape = [n_samples]

Target values (integers in classification, real numbers in regression)

class_weight : dict | 'auto', optional

Weights associated with classes in the form {class_label : weight}. If not given, all classes are supposed to have weight one.

The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

sample_weight : array-like, shape = [n_samples], optional

Weights applied to individual samples (1. for unweighted).

Returns **self** : object

Returns an instance of self.

Notes

For maximum effiency, use a sparse matrix in csr format (scipy.sparse.csr_matrix)

predict (*T*)

This function does classification or regression on an array of test vectors T.

For a classification model, the predicted class for each sample in T is returned. For a regression model, the function value of T calculated is returned.

For an one-class model, +1 or -1 is returned.

Parameters **T** : scipy.sparse.csr, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba (*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

predict_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

score(*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.svm.sparse.OneClassSVM

class scikits.learn.svm.sparse.**OneClassSVM**(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *cache_size=100.0*, *eps=0.001*, *nu=0.5*, *shrinking=True*, *probability=False*)

NuSVR for sparse matrices (csr)

See scikits.learn.svm.NuSVC for a complete list of parameters

Notes

For best results, this accepts a matrix in csr format (scipy.sparse.csr), but should be able to convert from any array-like object (including other sparse representations).

Methods

decision_function
fit
predict
predict_log_proba
predict_proba

__init__(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *cache_size=100.0*, *eps=0.001*, *nu=0.5*, *shrinking=True*, *probability=False*)

decision_function(*T*)

Calculate the distance of the samples T to the separating hyperplane.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class * (n_class-1) / 2]

Returns the decision function of the sample for each class in the model.

predict(*T*)

This function does classification or regression on an array of test vectors T.

For a classification model, the predicted class for each sample in T is returned. For a regression model, the function value of T calculated is returned.

For an one-class model, +1 or -1 is returned.

Parameters **T** : scipy.sparse.csr, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

predict_proba(*T*)

This function does classification or regression on a test vector T given a model with probability information.

Parameters **T** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will meaningless results on very small datasets.

scikits.learn.svm.sparse.LinearSVC

class scikits.learn.svm.sparse.**LinearSVC**(*penalty='l2'*, *loss='l2'*, *dual=True*, *eps=0.0001*, *C=1.0*, *multi_class=False*, *fit_intercept=True*, *intercept_scaling=1*)

Linear Support Vector Classification, Sparse Version

Similar to SVC with parameter kernel='linear', but uses internally liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should be faster for huge datasets.

Parameters **loss** : string, 'l1' or 'l2' (default 'l2')

Specifies the loss function. With 'l1' it is the standard SVM loss (a.k.a. hinge Loss) while with 'l2' it is the squared loss. (a.k.a. squared hinge Loss)

penalty : string, 'l1' or 'l2' (default 'l2')

Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to `coef_` vectors that are sparse.

dual : bool, (default True)

Select the algorithm to either solve the dual or primal optimization problem.

intercept_scaling : float, default: 1

when self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased

See Also:

SVC

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller eps parameter.

References

LIBLINEAR – A Library for Large Linear Classification http://www.csie.ntu.edu.tw/~cjlin/liblinear/

Attributes

*coef_*	array, shape = [n_features] if n_classes == 2 else [n_classes, n_features]	Wiehgiths asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.
*intercept_*	array, shape = [1] if n_classes == 2 else [n_classes]	constants in decision function

Methods

decision_function
fit
predict
predict_proba
score
transform

__init__ (*penalty='l2'*, *loss='l2'*, *dual=True*, *eps=0.0001*, *C=1.0*, *multi_class=False*, *fit_intercept=True*, *intercept_scaling=1*)

decision_function (*X*)

Return the decision function of X according to the trained model.

Parameters **X** : sparse matrix, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class]

Returns the decision function of the sample for each class in the model.

fit (*X*, *y*, *class_weight={}*, ***params*)

Fit the model using X, y as training data.

Parameters **X** : sparse matrix, shape = [n_samples, n_features]

Training vector, where n_samples in the number of samples and n_features is the number of features.

y : array, shape = [n_samples]

Target vector relative to X

Returns **self** : object

Returns an instance of self.

predict (*X*)

Predict target values of X according to the fitted model.

Parameters **X** : sparse matrix, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

score (*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

1.6.2 Generalized Linear Models

linear_model.LinearRegression([fit_intercept])	Ordinary least squares Linear Regression.
linear_model.Ridge([alpha, fit_intercept])	Ridge regression.
linear_model.RidgeCV([alphas, ...])	Ridge regression with built-in cross-validation.
linear_model.Lasso([alpha, fit_intercept])	Linear Model trained with L1 prior as regularizer (aka the Lasso)
linear_model.LassoCV([eps, n_alphas, ...])	Lasso linear model with iterative fitting along a regularization path
linear_model.ElasticNet([alpha, rho, ...])	Linear Model trained with L1 and L2 prior as regularizer
linear_model.ElasticNetCV([rho, eps, ...])	Elastic Net model with iterative fitting along a regularization path
linear_model.LARS([fit_intercept, verbose])	Least Angle Regression model a.k.a. LAR
linear_model.LassoLARS([alpha, ...])	Lasso model fit with Least Angle Regression a.k.a. LARS
linear_model.LogisticRegression([penalty, ...])	Logistic Regression.
linear_model.SGDClassifier([loss, penalty, ...])	Linear model fitted by minimizing a regularized empirical loss with SGD.
linear_model.SGDRegressor([loss, penalty, ...])	Linear model fitted by minimizing a regularized empirical loss with SGD

scikits.learn.linear_model.LinearRegression

class scikits.learn.linear_model.**LinearRegression**(*fit_intercept=True*)

Ordinary least squares Linear Regression.

Notes

From the implementation point of view, this is just plain Ordinary Least Squares (numpy.linalg.lstsq) wrapped as a predictor object.

Attributes

*coef_*	array	Estimated coefficients for the linear regression problem.
*intercept_*	array	Independent term in the linear model.

Methods

fit
predict
score

__init__(*fit_intercept=True*)

fit(*X*, *y*, ***params*)

Fit linear model.

Parameters **X** : numpy array of shape [n_samples,n_features]

Training data

y : numpy array of shape [n_samples]

Target values

fit_intercept : boolean, optional

wether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

Returns **self** : returns an instance of self.

predict(*X*)

Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score(*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.Ridge

class scikits.learn.linear_model.**Ridge**(*alpha=1.0*, *fit_intercept=True*)

Ridge regression.

Parameters **alpha** : float

Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to (2*C)^-1 in other linear models such as LogisticRegression or LinearSVC.

fit_intercept : boolean

Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

Examples

```
>>> from scikits.learn.linear_model import Ridge
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> np.random.seed(0)
>>> y = np.random.randn(n_samples)
>>> X = np.random.randn(n_samples, n_features)
>>> clf = Ridge(alpha=1.0)
>>> clf.fit(X, y)
Ridge(alpha=1.0, fit_intercept=True)
```

Methods

fit
predict
score

__init__ (*alpha=1.0*, *fit_intercept=True*)

fit (*X*, *y*, *sample_weight=1.0*, *solver='default'*, ***params*)
Fit Ridge regression model

Parameters **X** : numpy array of shape [n_samples,n_features]

Training data

y : numpy array of shape [n_samples]

Target values

sample_weight : float or numpy array of shape [n_samples]

Sample weight

solver : 'default' | 'cg'

Solver to use in the computational routines. 'default' will use the standard scipy.linalg.solve function, 'cg' will use the a conjugate gradient solver as found in scipy.sparse.linalg.cg.

Returns **self** : returns an instance of self.

predict (*X*)
Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score (*X*, *y*)
Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.RidgeCV

class scikits.learn.linear_model.**RidgeCV** (*alphas=array([0.1, 1., 10.]), fit_intercept=True*, *score_func=None*, *loss_func=None*, *cv=None*)
Ridge regression with built-in cross-validation.

By default, it performs Generalized Cross-Validation, which is a form of efficient Leave-One-Out cross-validation. Currently, only the n_features > n_samples case is handled efficiently.

Parameters **alphas: numpy array of shape [n_alpha]** :

Array of alpha values to try. Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to (2*C)^-1 in other linear models such as LogisticRegression or LinearSVC.

fit_intercept : boolean

Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

loss_func: callable, optional :

function that takes 2 arguments and compares them in order to evaluate the performance of prediciton (small is good) if None is passed, the score of the estimator is maximized

score_func: callable, optional :

function that takes 2 arguments and compares them in order to evaluate the performance of prediciton (big is good) if None is passed, the score of the estimator is maximized

See Also:

`Ridge`

Methods

`fit`
`predict`
`score`

__init__ (*alphas=array([0.1, 1., 10.]), fit_intercept=True, score_func=None, loss_func=None, cv=None*)

fit (*X, y, sample_weight=1.0, **params*)

Fit Ridge regression model

Parameters **X** : numpy array of shape [n_samples, n_features]

Training data

y : numpy array of shape [n_samples] or [n_samples, n_responses]

Target values

sample_weight : float or numpy array of shape [n_samples]

Sample weight

cv : cross-validation generator, optional

If None, Generalized Cross-Validationn (efficient Leave-One-Out) will be used.

Returns **self** : Returns self.

predict (*X*)

Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score (*X, y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.Lasso

class scikits.learn.linear_model.**Lasso**(*alpha=1.0*, *fit_intercept=True*)

Linear Model trained with L1 prior as regularizer (aka the Lasso)

Technically the Lasso model is optimizing the same objective function as the Elastic Net with rho=1.0 (no L2 penalty).

Parameters **alpha** : float, optional

Constant that multiplies the L1 term. Defaults to 1.0

fit_intercept : boolean

whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

See Also:

LassoLARS

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a fortran contiguous numpy array.

Examples

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.Lasso(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
Lasso(alpha=0.1, fit_intercept=True)
>>> print clf.coef_
[ 0.85  0.  ]
>>> print clf.intercept_
0.15
```

Attributes

*coef_*	array, shape = [n_features]	parameter vector (w in the fomulation formula)
*intercept_*	float	independent term in decision function.

Methods

fit
predict
score

__init__ (*alpha=1.0*, *fit_intercept=True*)

fit (*X*, *y*, *precompute='auto'*, *Xy=None*, *max_iter=1000*, *tol=0.0001*, *coef_init=None*, ***params*)

Fit Elastic Net model with coordinate descent

Parameters **X: ndarray, (n_samples, n_features)** :

Data

y: ndarray, (n_samples) :

Target

precompute : True | False | 'auto' | array-like

Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

Xy : array-like, optional

Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomuted.

max_iter: int, optional :

The maximum number of iterations

tol: float, optional :

The tolerance for the optimization: if the updates are smaller than 'tol', the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a fortran contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

predict (*X*)

Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score (*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.LassoCV

class scikits.learn.linear_model.**LassoCV**(*eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*)

Lasso linear model with iterative fitting along a regularization path

The best model is selected by cross-validation.

Parameters **eps** : float, optional

Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3.

n_alphas : int, optional

Number of alphas along the regularization path

alphas : numpy array, optional

List of alphas where to compute the models. If None alphas are set automatically

Notes

See examples/linear_model/lasso_path_with_crossvalidation.py for an example.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a fortran contiguous numpy array.

Methods

fit
path
predict
score

__init__(*eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*)

fit(*X*, *y*, *cv=None*, ***fit_params*)

Fit linear model with coordinate descent along decreasing alphas using cross-validation

Parameters **X** : numpy array of shape [n_samples,n_features]

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

y : numpy array of shape [n_samples]

Target values

cv : cross-validation generator, optional

If None, KFold will be used.

fit_params : kwargs

keyword arguments passed to the Lasso fit method

static path(*X*, *y*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *verbose=False*, ***fit_params*)

Compute Lasso path with coordinate descent

Parameters **X** : numpy array of shape [n_samples,n_features]

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

y : numpy array of shape [n_samples]

Target values

eps : float, optional

Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3

n_alphas : int, optional

Number of alphas along the regularization path

alphas : numpy array, optional

List of alphas where to compute the models. If None alphas are set automatically

fit_params : kwargs

keyword arguments passed to the Lasso fit method

Returns **models** : a list of models along the regularization path

Notes

See examples/plot_lasso_coordinate_descent_path.py for an example.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a fortran contiguous numpy array.

predict(*X*)

Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score(*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.ElasticNet

class scikits.learn.linear_model.**ElasticNet**(*alpha=1.0*, *rho=0.5*, *fit_intercept=True*)

Linear Model trained with L1 and L2 prior as regularizer

rho=1 is the lasso penalty. Currently, rho <= 0.01 is not reliable, unless you supply your own sequence of alpha.

Parameters **alpha** : float

Constant that multiplies the L1 term. Defaults to 1.0

rho : float

The ElasticNet mixing parameter, with 0 < rho <= 1.

coef_: ndarray of shape n_features :

The initial coeffients to warm-start the optimization

fit_intercept: bool :

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.

Notes

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a fortran contiguous numpy array.

Methods

```
fit
predict
score
```

__init__ (*alpha=1.0*, *rho=0.5*, *fit_intercept=True*)

fit (*X*, *y*, *precompute='auto'*, *Xy=None*, *max_iter=1000*, *tol=0.0001*, *coef_init=None*, ***params*)

Fit Elastic Net model with coordinate descent

Parameters **X: ndarray, (n_samples, n_features)** :

Data

y: ndarray, (n_samples) :

Target

precompute : True | False | 'auto' | array-like

Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

Xy : array-like, optional

Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomuted.

max_iter: int, optional :

The maximum number of iterations

tol: float, optional :

The tolerance for the optimization: if the updates are smaller than 'tol', the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a fortran contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

predict(*X*)
Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score(*X*, *y*)
Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.ElasticNetCV

class scikits.learn.linear_model.**ElasticNetCV**(*rho=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*)

Elastic Net model with iterative fitting along a regularization path

The best model is selected by cross-validation.

Parameters **rho** : float, optional

float between 0 and 1 passed to ElasticNet (scaling between l1 and l2 penalties)

eps : float, optional

Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3.

n_alphas : int, optional

Number of alphas along the regularization path

alphas : numpy array, optional

List of alphas where to compute the models. If None alphas are set automatically

Notes

See examples/linear_model/lasso_path_with_crossvalidation.py for an example.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a fortran contiguous numpy array.

Methods

fit
path
predict
score

__init__(*rho=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*)

fit (*X*, *y*, *cv=None*, ***fit_params*)
Fit linear model with coordinate descent along decreasing alphas using cross-validation

Parameters **X** : numpy array of shape [n_samples,n_features]

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

y : numpy array of shape [n_samples]

Target values

cv : cross-validation generator, optional

If None, KFold will be used.

fit_params : kwargs

keyword arguments passed to the Lasso fit method

static path (*X*, *y*, *rho=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *verbose=False*, ***fit_params*)
Compute Elastic-Net path with coordinate descent

Parameters **X** : numpy array of shape [n_samples, n_features]

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

y : numpy array of shape [n_samples]

Target values

rho : float, optional

float between 0 and 1 passed to ElasticNet (scaling between l1 and l2 penalties). rho=1 corresponds to the Lasso

eps : float

Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3

n_alphas : int, optional

Number of alphas along the regularization path

alphas : numpy array, optional

List of alphas where to compute the models. If None alphas are set automatically

fit_params : kwargs

keyword arguments passed to the Lasso fit method

Returns **models** : a list of models along the regularization path

Notes

See examples/plot_lasso_coordinate_descent_path.py for an example.

predict (*X*)
Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score (*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.LARS

class scikits.learn.linear_model.**LARS** (*fit_intercept=True*, *verbose=False*)

Least Angle Regression model a.k.a. LAR

Parameters **n_features** : int, optional

Number of selected active features

fit_intercept : boolean

whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

See Also:

lars_path, LassoLARS

References

http://en.wikipedia.org/wiki/Least_angle_regression

Examples

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.LARS()
>>> clf.fit([[-1,1], [0, 0], [1, 1]], [-1, 0, -1], max_features=1)
LARS(verbose=False, fit_intercept=True)
>>> print clf.coef_
[ 0.         -0.81649658]
```

Attributes

*coef_*	array, shape = [n_features]	parameter vector (w in the fomulation formula)
*intercept_*	float	independent term in decision function.

Methods

fit
predict
score

__init__ (*fit_intercept=True*, *verbose=False*)

fit (*X*, *y*, *normalize=True*, *max_features=None*, *precompute='auto'*, *overwrite_X=False*, ***params*)
Fit the model using X, y as training data.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training data.

y : array-like, shape = [n_samples]

Target values.

precompute : True | False | 'auto' | array-like

Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

Returns **self** : object

returns an instance of self.

predict (*X*)
Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score (*X*, *y*)
Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.LassoLARS

class scikits.learn.linear_model.**LassoLARS** (*alpha=1.0*, *fit_intercept=True*, *verbose=False*)
Lasso model fit with Least Angle Regression a.k.a. LARS

It is a Linear Model trained with an L1 prior as regularizer. lasso).

Parameters **alpha** : float, optional

Constant that multiplies the L1 term. Defaults to 1.0

fit_intercept : boolean

whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

See Also:

lars_path, Lasso

References

http://en.wikipedia.org/wiki/Least_angle_regression

Examples

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.LassoLARS(alpha=0.01)
>>> clf.fit([[-1,1], [0, 0], [1, 1]], [-1, 0, -1])
LassoLARS(alpha=0.01, verbose=False, fit_intercept=True)
>>> print clf.coef_
[ 0.         -0.72649658]
```

Attributes

*coef_*	array, shape = [n_features]	parameter vector (w in the fomulation formula)
*intercept_*	float	independent term in decision function.

Methods

fit
predict
score

__init__ (*alpha=1.0*, *fit_intercept=True*, *verbose=False*)

fit (*X*, *y*, *normalize=True*, *max_features=None*, *precompute='auto'*, *overwrite_X=False*, ***params*)

Fit the model using X, y as training data.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training data.

y : array-like, shape = [n_samples]

Target values.

precompute : True | False | 'auto' | array-like

Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

Returns **self** : object

returns an instance of self.

predict (*X*)

Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score (*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.LogisticRegression

class scikits.learn.linear_model.**LogisticRegression**(*penalty='l2'*, *dual=False*, *eps=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*)

Logistic Regression.

Implements L1 and L2 regularized logistic regression.

Parameters **penalty** : string, 'l1' or 'l2'

Used to specify the norm used in the penalization

dual : boolean

Dual or primal formulation. Dual formulation is only implemented for l2 penalty.

C : float

Specifies the strength of the regularization. The smaller it is the bigger in the regularization.

fit_intercept : bool, default: True

Specifies if a constant (a.k.a. bias or intercept) should be added the decision function

intercept_scaling : float, default: 1

when self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased

See Also:

LinearSVC

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller eps parameter.

References

LIBLINEAR – A Library for Large Linear Classification http://www.csie.ntu.edu.tw/~cjlin/liblinear/

Attributes

*coef_*	array, shape = [n_classes-1, n_features]	Coefficient of the features in the decision function.
*intercept_*	array, shape = [n_classes-1]	intercept (a.k.a. bias) added to the decision function. It is available only when parameter intercept is set to True

Methods

decision_function
fit
predict
predict_log_proba
predict_proba
score
transform

__init__ (*penalty='l2'*, *dual=False*, *eps=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*)

decision_function (*X*)

Return the decision function of X according to the trained model.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_class]

Returns the decision function of the sample for each class in the model.

fit (*X*, *y*, *class_weight={}*, ***params*)

Fit the model according to the given training data and parameters.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training vector, where n_samples in the number of samples and n_features is the number of features.

y : array-like, shape = [n_samples]

Target vector relative to X

class_weight : dict , {class_label

Weights associated with classes. If not given, all classes are supposed to have weight one.

Returns **self** : object

Returns self.

predict (*X*)

Predict target values of X according to the fitted model.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

predict_log_proba (*X*)

Log of Probability estimates.

The returned estimates for all classes are ordered by the label of classes.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **X** : array-like, shape = [n_samples, n_classes]

Returns the log-probabilities of the sample for each class in the model, where classes are ordered by arithmetical order.

predict_proba (*X*)

Probability estimates.

The returned estimates for all classes are ordered by the label of classes.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = [n_samples, n_classes]

Returns the probability of the sample for each class in the model, where classes are ordered by arithmetical order.

score (*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

scikits.learn.linear_model.SGDClassifier

class scikits.learn.linear_model.**SGDClassifier**(*loss='hinge'*, *penalty='l2'*, *alpha=0.0001*, *rho=0.84999999999999998*, *fit_intercept=True*, *n_iter=5*, *shuffle=False*, *verbose=0*, *n_jobs=1*)

Linear model fitted by minimizing a regularized empirical loss with SGD.

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense numpy arrays of floating point values for the features.

Parameters **loss** : str, 'hinge' or 'log' or 'modified_huber'

The loss function to be used. Defaults to 'hinge'. The hinge loss is a margin loss used by standard linear SVM models. The 'log' loss is the loss of logistic regression models and can be used for probability estimation in binary classifiers. 'modified_huber' is another smooth loss that brings tolerance to outliers.

penalty : str, 'l2' or 'l1' or 'elasticnet'

The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' migh bring sparsity to the model (feature selection) not achievable with 'l2'.

alpha : float

Constant that multiplies the regularization term. Defaults to 0.0001

rho : float

The Elastic Net mixing parameter, with 0 < rho <= 1. Defaults to 0.85.

fit_intercept: bool :

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

n_iter: int :

The number of passes over the training data (aka epochs). Defaults to 5.

shuffle: bool :

Whether or not the training data should be shuffled after each epoch. Defaults to False.

verbose: integer, optional :

The verbosity level

n_jobs: integer, optional :

The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. -1 means 'all CPUs'. Defaults to 1.

See Also:

`LinearSVC`, `LogisticRegression`

Examples

```
>>> import numpy as np
>>> from scikits.learn import linear_model
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> Y = np.array([1, 1, 2, 2])
>>> clf = linear_model.SGDClassifier()
>>> clf.fit(X, Y)
SGDClassifier(loss='hinge', n_jobs=1, shuffle=False, verbose=0, n_iter=5,
       fit_intercept=True, penalty='l2', rho=1.0, alpha=0.0001)
>>> print clf.predict([[-0.8, -1]])
[ 1.]
```

Attributes

*coef_*	array, shape = [n_features] if n_classes == 2 else [n_classes, n_features]	Weights asigned to the features.
*intercept_*	array, shape = [1] if n_classes == 2 else [n_classes]	Constants in decision function.

Methods

`decision_function`
`fit`
`predict`
`predict_proba`
`score`

__init__ (*loss='hinge', penalty='l2', alpha=0.0001, rho=0.84999999999999998, fit_intercept=True, n_iter=5, shuffle=False, verbose=0, n_jobs=1*)

decision_function (*X*)

Predict signed 'distance' to the hyperplane (aka confidence score)

Parameters **X** : array, shape [n_samples, n_features]

Returns **array, shape = [n_samples] if n_classes == 2 else [n_samples, n_classes]** :

The signed 'distances' to the hyperplane(s).

fit (*X*, *y*, *coef_init=None*, *intercept_init=None*, *class_weight={}*, ***params*)
Fit linear model with Stochastic Gradient Descent.

Parameters **X** : numpy array of shape [n_samples,n_features]

Training data

y : numpy array of shape [n_samples]

Target values

coef_init : array, shape = [n_features] if n_classes == 2 else [n_classes,

n_features] :

The initial coeffients to warm-start the optimization.

intercept_init : array, shape = [1] if n_classes == 2 else [n_classes]

The initial intercept to warm-start the optimization.

class_weight : dict, {class_label

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

Returns **self** : returns an instance of self.

predict (*X*)
Predict using the linear model

Parameters **X** : array or scipy.sparse matrix of shape [n_samples, n_features]

Whether the numpy.array or scipy.sparse matrix is accepted dependes on the actual implementation

Returns **array, shape = [n_samples]** :

Array containing the predicted class labels.

predict_proba (*X*)
Predict class membership probability

Parameters **X** : array or scipy.sparse matrix of shape [n_samples, n_features]

Returns **array, shape = [n_samples] if n_classes == 2 else [n_samples,** :

n_classes] :

Contains the membership probabilities of the positive class.

score (*X*, *y*)
Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

scikits.learn.linear_model.SGDRegressor

class scikits.learn.linear_model.**SGDRegressor**(*loss='squared_loss'*, *penalty='l2'*, *alpha=0.0001*, *rho=0.84999999999999998*, *fit_intercept=True*, *n_iter=5*, *shuffle=False*, *verbose=0*, *p=0.10000000000000001*)

Linear model fitted by minimizing a regularized empirical loss with SGD

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense numpy arrays of floating point values for the features.

Parameters **loss** : str, 'squared_loss' or 'huber'

The loss function to be used. Defaults to 'squared_loss' which refers to the ordinary least squares fit. 'huber' is an epsilon insensitive loss function for robust regression.

penalty : str, 'l2' or 'l1' or 'elasticnet'

The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' migh bring sparsity to the model (feature selection) not achievable with 'l2'.

alpha : float

Constant that multiplies the regularization term. Defaults to 0.0001

rho : float

The Elastic Net mixing parameter, with 0 < rho <= 1. Defaults to 0.85.

fit_intercept: bool :

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

n_iter: int :

The number of passes over the training data (aka epochs). Defaults to 5.

shuffle: bool :

Whether or not the training data should be shuffled after each epoch. Defaults to False.

verbose: integer, optional :

The verbosity level.

p : float

Epsilon in the epsilon-insensitive huber loss function; only if *loss=='huber'*.

See Also:

Ridge, ElasticNet, Lasso, SVR

Examples

```
>>> import numpy as np
>>> from scikits.learn import linear_model
>>> n_samples, n_features = 10, 5
>>> np.random.seed(0)
>>> y = np.random.randn(n_samples)
>>> X = np.random.randn(n_samples, n_features)
>>> clf = linear_model.SGDRegressor()
>>> clf.fit(X, y)
SGDRegressor(loss='squared_loss', shuffle=False, verbose=0, n_iter=5,
       fit_intercept=True, penalty='l2', p=0.1, rho=1.0, alpha=0.0001)
```

Attributes

*coef_*	array, shape = [n_features]	Weights asigned to the features.
*intercept_*	array, shape = [1]	The intercept term.

Methods

fit
predict
score

__init__(*loss='squared_loss', penalty='l2', alpha=0.0001, rho=0.84999999999999998, fit_intercept=True, n_iter=5, shuffle=False, verbose=0, p=0.10000000000000001*)

fit(*X, y, coef_init=None, intercept_init=None, **params*)

Fit linear model with Stochastic Gradient Descent.

Parameters **X** : numpy array of shape [n_samples,n_features]

Training data

y : numpy array of shape [n_samples]

Target values

coef_init : array, shape = [n_features]

The initial coeffients to warm-start the optimization.

intercept_init : array, shape = [1]

The initial intercept to warm-start the optimization.

Returns **self** : returns an instance of self.

predict(*X*)

Predict using the linear model

Parameters **X** : array or scipy.sparse matrix of shape [n_samples, n_features]

Whether the numpy.array or scipy.sparse matrix is accepted dependes on the actual implementation

Returns **array, shape = [n_samples]** :

Array containing the predicted class labels.

score(*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

linear_model.lasso_path(X, y, **fit_params)	Compute Lasso path with coordinate descent
linear_model.lars_path(X, y[, Xy, Gram, ...])	Compute Least Angle Regression and LASSO path

scikits.learn.linear_model.lasso_path

scikits.learn.linear_model.**lasso_path**(*X*, *y*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *fit_intercept=True*, *verbose=False*, ***fit_params*)

Compute Lasso path with coordinate descent

Parameters **X** : numpy array of shape [n_samples,n_features]

Training data. Pass directly as fortran contiguous data to avoid unnecessary memory duplication

y : numpy array of shape [n_samples]

Target values

eps : float, optional

Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3

n_alphas : int, optional

Number of alphas along the regularization path

alphas : numpy array, optional

List of alphas where to compute the models. If None alphas are set automatically

fit_params : kwargs

keyword arguments passed to the Lasso fit method

Returns **models** : a list of models along the regularization path

Notes

See examples/plot_lasso_coordinate_descent_path.py for an example.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a fortran contiguous numpy array.

scikits.learn.linear_model.lars_path

scikits.learn.linear_model.**lars_path**(*X*, *y*, *Xy=None*, *Gram=None*, *max_features=None*, *alpha_min=0*, *method='lar'*, *overwrite_X=False*, *overwrite_Gram=False*, *verbose=False*)

Compute Least Angle Regression and LASSO path

Parameters **X: array, shape: (n_samples, n_features)** :

Input data

y: array, shape: (n_samples) :

Input targets

max_features: integer, optional :

Maximum number of selected features.

Gram: array, shape: (n_features, n_features), optional :

Precomputed Gram matrix (X' * X)

alpha_min: float, optional :

Minimum correlation along the path. It corresponds to the regularization parameter alpha parameter in the Lasso.

method: 'lar' | 'lasso' :

Specifies the returned model. Select 'lar' for Least Angle Regression, 'lasso' for the Lasso.

Returns **alphas: array, shape: (max_features + 1,)** :

Maximum of covariances (in absolute value) at each iteration.

active: array, shape (max_features,) :

Indices of active variables at the end of the path.

coefs: array, shape (n_features, max_features+1) :

Coefficients along the path

See Also:

LassoLARS, *LARS*

Notes

- http://en.wikipedia.org/wiki/Least-angle_regression
- http://en.wikipedia.org/wiki/Lasso_(statistics)#LASSO_method

For sparse data

linear_model.sparse.Lasso([alpha, fit_intercept])	Linear Model trained with L1 prior as regularizer
linear_model.sparse.ElasticNet([alpha, rho, ...])	Linear Model trained with L1 and L2 prior as regularizer
linear_model.sparse.SGDClassifier([loss, ...])	Linear model fitted by minimizing a regularized empirical loss with SGD
linear_model.sparse.SGDRegressor([loss, ...])	Linear model fitted by minimizing a regularized empirical loss with SGD

scikits.learn.linear_model.sparse.Lasso

class scikits.learn.linear_model.sparse.**Lasso**(*alpha=1.0*, *fit_intercept=False*)

Linear Model trained with L1 prior as regularizer

This implementation works on scipy.sparse X and dense **coef_**. Technically this is the same as Elastic Net with the L2 penalty set to zero.

Parameters **alpha** : float

Constant that multiplies the L1 term. Defaults to 1.0

coef_ : ndarray of shape n_features

The initial coeffients to warm-start the optimization

fit_intercept: bool :

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.

Methods

fit
predict
score

__init__(*alpha=1.0*, *fit_intercept=False*)

fit(*X*, *y*, *max_iter=1000*, *tol=0.0001*, ***params*)

Fit current model with coordinate descent

X is expected to be a sparse matrix. For maximum efficiency, use a sparse matrix in CSC format (scipy.sparse.csc_matrix)

predict(*X*)

Predict using the linear model

Parameters **X** : scipy.sparse matrix of shape [n_samples, n_features]

Returns **array, shape = [n_samples] with the predicted real values** :

score(*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.sparse.ElasticNet

class scikits.learn.linear_model.sparse.**ElasticNet**(*alpha=1.0*, *rho=0.5*, *fit_intercept=False*)

Linear Model trained with L1 and L2 prior as regularizer

This implementation works on scipy.sparse X and dense **coef_**.

rho=1 is the lasso penalty. Currently, rho <= 0.01 is not reliable, unless you supply your own sequence of alpha.

Parameters **alpha** : float

Constant that multiplies the L1 term. Defaults to 1.0

rho : float

The ElasticNet mixing parameter, with 0 < rho <= 1.

coef_ : ndarray of shape n_features

The initial coeffients to warm-start the optimization

fit_intercept: bool :

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.

TODO: fit_intercept=True is not yet implemented

Methods

fit
predict
score

__init__(*alpha=1.0*, *rho=0.5*, *fit_intercept=False*)

fit(*X*, *y*, *max_iter=1000*, *tol=0.0001*, ***params*)

Fit current model with coordinate descent

X is expected to be a sparse matrix. For maximum efficiency, use a sparse matrix in CSC format (scipy.sparse.csc_matrix)

predict(*X*)

Predict using the linear model

Parameters **X** : scipy.sparse matrix of shape [n_samples, n_features]

Returns **array, shape = [n_samples] with the predicted real values** :

score(*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.sparse.SGDClassifier

class scikits.learn.linear_model.sparse.**SGDClassifier**(*loss='hinge'*, *penalty='l2'*, *alpha=0.0001*, *rho=0.84999999999999998*, *fit_intercept=True*, *n_iter=5*, *shuffle=False*, *verbose=0*, *n_jobs=1*)

Linear model fitted by minimizing a regularized empirical loss with SGD

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works on scipy.sparse X and dense **coef_**.

Parameters **loss** : str, 'hinge' or 'log' or 'modified_huber'

The loss function to be used. Defaults to 'hinge'. The hinge loss is a margin loss used by standard linear SVM models. The 'log' loss is the loss of logistic regression models and can be used for probability estimation in binary classifiers. 'modified_huber' is another smooth loss that brings tolerance to outliers.

penalty : str, 'l2' or 'l1' or 'elasticnet'

The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' migh bring sparsity to the model (feature selection) not achievable with 'l2'.

alpha : float

Constant that multiplies the regularization term. Defaults to 0.0001

rho : float

The Elastic Net mixing parameter, with 0 < rho <= 1. Defaults to 0.85.

fit_intercept: bool :

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

n_iter: int :

The number of passes over the training data (aka epochs). Defaults to 5.

shuffle: bool :

Whether or not the training data should be shuffled after each epoch. Defaults to False.

verbose: integer, optional :

The verbosity level

n_jobs: integer, optional :

The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. -1 means 'all CPUs'. Defaults to 1.

See Also:

`LinearSVC`, `LogisticRegression`

Examples

```
>>> import numpy as np
>>> from scikits.learn import linear_model
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> clf = linear_model.sparse.SGDClassifier()
```

```
>>> clf.fit(X, y)
SGDClassifier(loss='hinge', n_jobs=1, shuffle=False, verbose=0, n_iter=5,
       fit_intercept=True, penalty='l2', rho=1.0, alpha=0.0001)
>>> print clf.predict([[-0.8, -1]])
[ 1.]
```

Attributes

*coef_*	array, shape = [n_features] if n_classes == 2 else [n_classes, n_features]	Weights asigned to the features.
*intercept_*	array, shape = [1] if n_classes == 2 else [n_classes]	Constants in decision function.

Methods

decision_function
fit
predict
predict_proba
score

__init__ (*loss='hinge'*, *penalty='l2'*, *alpha=0.0001*, *rho=0.84999999999999998*, *fit_intercept=True*, *n_iter=5*, *shuffle=False*, *verbose=0*, *n_jobs=1*)

decision_function (*X*)

Predict signed 'distance' to the hyperplane (aka confidence score).

Parameters **X** : scipy.sparse matrix of shape [n_samples, n_features]

Returns **array, shape = [n_samples] if n_classes == 2 else [n_samples, n_classes]** :

The signed 'distances' to the hyperplane(s).

fit (*X*, *y*, *coef_init=None*, *intercept_init=None*, *class_weight={}*, ***params*)

Fit linear model with Stochastic Gradient Descent

X is expected to be a sparse matrix. For maximum efficiency, use a sparse matrix in CSR format (scipy.sparse.csr_matrix)

Parameters **X** : scipy sparse matrix of shape [n_samples,n_features]

Training data

y : numpy array of shape [n_samples]

Target values

coef_init : array, shape = [n_features] if n_classes == 2 else

[n_classes,n_features] :

The initial coeffients to warm-start the optimization.

intercept_init : array, shape = [1] if n_classes == 2 else [n_classes]

The initial intercept to warm-start the optimization.

class_weight : dict, {class_label

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

Returns **self** : returns an instance of self.

predict(*X*)

Predict using the linear model

Parameters **X** : array or scipy.sparse matrix of shape [n_samples, n_features]

Whether the numpy.array or scipy.sparse matrix is accepted dependes on the actual implementation

Returns **array, shape = [n_samples]** :

Array containing the predicted class labels.

predict_proba(*X*)

Predict class membership probability

Parameters **X** : array or scipy.sparse matrix of shape [n_samples, n_features]

Returns **array, shape = [n_samples] if n_classes == 2 else [n_samples,** :

n_classes] :

Contains the membership probabilities of the positive class.

score(*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

scikits.learn.linear_model.sparse.SGDRegressor

class scikits.learn.linear_model.sparse.**SGDRegressor**(*loss='squared_loss'*, *penalty='l2'*, *alpha=0.0001*, *rho=0.84999999999999998*, *fit_intercept=True*, *n_iter=5*, *shuffle=False*, *verbose=0*, *p=0.10000000000000001*)

Linear model fitted by minimizing a regularized empirical loss with SGD

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense numpy arrays of floating point values for the features.

Parameters **loss** : str, 'squared_loss' or 'huber'

The loss function to be used. Defaults to 'squared_loss' which refers to the ordinary least squares fit. 'huber' is an epsilon insensitive loss function for robust regression.

penalty : str, 'l2' or 'l1' or 'elasticnet'

The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' migh bring sparsity to the model (feature selection) not achievable with 'l2'.

alpha : float

Constant that multiplies the regularization term. Defaults to 0.0001

rho : float

The Elastic Net mixing parameter, with 0 < rho <= 1. Defaults to 0.85.

fit_intercept: bool :

Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

n_iter: int :

The number of passes over the training data (aka epochs). Defaults to 5.

shuffle: bool :

Whether or not the training data should be shuffled after each epoch. Defaults to False.

verbose: integer, optional :

The verbosity level

p : float

Epsilon in the epsilon insensitive huber loss function; only if *loss=='huber'*.

See Also:

`RidgeRegression`, `ElasticNet`, `Lasso`, `SVR`

Examples

```
>>> import numpy as np
>>> from scikits.learn import linear_model
>>> n_samples, n_features = 10, 5
>>> np.random.seed(0)
>>> y = np.random.randn(n_samples)
>>> X = np.random.randn(n_samples, n_features)
>>> clf = linear_model.sparse.SGDRegressor()
>>> clf.fit(X, y)
SGDRegressor(loss='squared_loss', shuffle=False, verbose=0, n_iter=5,
       fit_intercept=True, penalty='l2', p=0.1, rho=1.0, alpha=0.0001)
```

Attributes

*coef_*	array, shape = [n_features]	Weights asigned to the features.
*intercept_*	array, shape = [1]	The intercept term.

Methods

fit
predict
score

__init__ (*loss='squared_loss'*, *penalty='l2'*, *alpha=0.0001*, *rho=0.84999999999999998*, *fit_intercept=True*, *n_iter=5*, *shuffle=False*, *verbose=0*, *p=0.10000000000000001*)

fit (*X*, *y*, *coef_init=None*, *intercept_init=None*, ***params*)

Fit linear model with Stochastic Gradient Descent

X is expected to be a sparse matrix. For maximum efficiency, use a sparse matrix in CSR format (scipy.sparse.csr_matrix)

Parameters **X** : scipy sparse matrix of shape [n_samples,n_features]

Training data

y : numpy array of shape [n_samples]

Target values

coef_init : array, shape = [n_features]

The initial coeffients to warm-start the optimization.

intercept_init : array, shape = [1]

Returns **self** : returns an instance of self.

predict (*X*)

Predict using the linear model

Parameters **X** : array or scipy.sparse matrix of shape [n_samples, n_features]

Whether the numpy.array or scipy.sparse matrix is accepted dependes on the actual implementation

Returns **array, shape = [n_samples]** :

Array containing the predicted class labels.

score (*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

1.6.3 Bayesian Regression

linear_model.BayesianRidge([n_iter, eps, ...])	Bayesian ridge regression
linear_model.ARDRegression([n_iter, eps, ...])	Bayesian ARD regression.

scikits.learn.linear_model.BayesianRidge

class scikits.learn.linear_model.**BayesianRidge**(*n_iter=300*, *eps=0.001*, *alpha_1=9.9999999999999995e-07*, *alpha_2=9.9999999999999995e-07*, *lambda_1=9.9999999999999995e-07*, *lambda_2=9.9999999999999995e-07*, *compute_score=False*, *fit_intercept=True*, *verbose=False*)

Bayesian ridge regression

Fit a Bayesian ridge model and optimize the regularization parameters lambda (precision of the weights) and alpha (precision of the noise).

Parameters **X** : array, shape = (n_samples, n_features)

Training vectors.

y : array, shape = (length)

Target values for training vectors

n_iter : int, optional

Maximum number of interations. Default is 300.

eps : float, optional

Stop the algorithm if w has converged. Default is 1.e-3.

alpha_1 : float, optional

Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter. Default is 1.e-6

alpha_2 : float, optional

Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

lambda_1 : float, optional

Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

lambda_2 : float, optional

Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter. Default is 1.e-6

compute_score : boolean, optional

If True, compute the objective function at each step of the model. Default is False

fit_intercept : boolean, optional

wether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered). Default is True.

Notes

See examples/linear_model/plot_bayesian_ridge.py for an example.

Examples

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.BayesianRidge()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
BayesianRidge(n_iter=300, verbose=False, lambda_1=1e-06, lambda_2=1e-06,
       fit_intercept=True, eps=0.001, alpha_2=1e-06, alpha_1=1e-06,
       compute_score=False)
>>> clf.predict([[1, 1]])
array([ 1.])
```

Attributes

*coef_*	array, shape = (n_features)	Coefficients of the regression model (mean of distribution)
*alpha_*	float	estimated precision of the noise.
*lambda_*	array, shape = (n_features)	estimated precisions of the weights.
*scores_*	float	if computed, value of the objective function (to be maximized)

Methods

fit(X, y)	self	Fit the model
predict(X)	array	Predict using the model.

__init__ (*n_iter=300*, *eps=0.001*, *alpha_1=9.9999999999999995e-07*, *alpha_2=9.9999999999999995e-07*, *lambda_1=9.9999999999999995e-07*, *lambda_2=9.9999999999999995e-07*, *compute_score=False*, *fit_intercept=True*, *verbose=False*)

fit (*X*, *y*, ***params*)

Fit the model

Parameters **X** : numpy array of shape [n_samples,n_features]

Training data

y : numpy array of shape [n_samples]

Target values

Returns **self** : returns an instance of self.

predict (*X*)

Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score (*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

scikits.learn.linear_model.ARDRegression

class scikits.learn.linear_model.**ARDRegression**(*n_iter=300*, *eps=0.001*, *alpha_1=9.9999999999999995e-07*, *alpha_2=9.9999999999999995e-07*, *lambda_1=9.9999999999999995e-07*, *lambda_2=9.9999999999999995e-07*, *compute_score=False*, *threshold_lambda=10000.0*, *fit_intercept=True*, *verbose=False*)

Bayesian ARD regression.

Fit the weights of a regression model, using an ARD prior. The weights of the regression model are assumed to be in Gaussian distributions. Also estimate the parameters lambda (precisions of the distributions of the weights) and alpha (precision of the distribution of the noise). The estimation is done by an iterative procedures (Evidence Maximization)

Parameters **X** : array, shape = (n_samples, n_features)

Training vectors.

y : array, shape = (n_samples)

Target values for training vectors

n_iter : int, optional

Maximum number of interations. Default is 300

eps : float, optional

Stop the algorithm if w has converged. Default is 1.e-3.

alpha_1 : float, optional

Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

alpha_2 : float, optional

Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

lambda_1 : float, optional

Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

lambda_2 : float, optional

Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

compute_score : boolean, optional

If True, compute the objective function at each step of the model. Default is False.

threshold_lambda : float, optional

threshold for removing (pruning) weights with high precision from the computation. Default is 1.e+4.

fit_intercept : boolean, optional

wether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered). Default is True.

verbose : boolean, optional

Verbose mode when fitting the model. Default is False.

Notes

See examples/linear_model/plot_ard.py for an example.

Examples

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.ARDRegression()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
ARDRegression(n_iter=300, verbose=False, lambda_1=1e-06, lambda_2=1e-06,
       fit_intercept=True, eps=0.001, threshold_lambda=10000.0,
       alpha_2=1e-06, alpha_1=1e-06, compute_score=False)
>>> clf.predict([[1, 1]])
array([ 1.])
```

Attributes

*coef_*	array, shape = (n_features)	Coefficients of the regression model (mean of distribution)
*alpha_*	float	estimated precision of the noise.
*lambda_*	array, shape = (n_features)	estimated precisions of the weights.
*sigma_*	array, shape = (n_features, n_features)	estimated variance-covariance matrix of the weights
*scores_*	float	if computed, value of the objective function (to be maximized)

Methods

fit(X, y)	self	Fit the model
predict(X)	array	Predict using the model.

__init__ (*n_iter=300*, *eps=0.001*, *alpha_1=9.9999999999999995e-07*, *alpha_2=9.9999999999999995e-07*, *lambda_1=9.9999999999999995e-07*, *lambda_2=9.9999999999999995e-07*, *compute_score=False*, *threshold_lambda=10000.0*, *fit_intercept=True*, *verbose=False*)

fit (*X*, *y*, ***params*)

Fit the ARDRegression model according to the given training data and parameters.

Iterative procedure to maximize the evidence

Parameters **X** : array-like, shape = [n_samples, n_features]

Training vector, where n_samples in the number of samples and n_features is the number of features.

y : array, shape = [n_samples]

Target values (integers)

Returns **self** : returns an instance of self.

predict(*X*)

Predict using the linear model

Parameters **X** : numpy array of shape [n_samples, n_features]

Returns **C** : array, shape = [n_samples]

Returns predicted values.

score(*X*, *y*)

Returns the coefficient of determination of the prediction

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Returns **z** : float

1.6.4 Naive Bayes

naive_bayes.GNB()	Gaussian Naive Bayes (GNB)

scikits.learn.naive_bayes.GNB

class scikits.learn.naive_bayes.**GNB**

Gaussian Naive Bayes (GNB)

Parameters **X** : array-like, shape = [n_samples, n_features]

Training vector, where n_samples in the number of samples and n_features is the number of features.

y : array, shape = [n_samples]

Target vector relative to X

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> Y = np.array([1, 1, 1, 2, 2, 2])
>>> from scikits.learn.naive_bayes import GNB
>>> clf = GNB()
>>> clf.fit(X, Y)
GNB()
>>> print clf.predict([[-0.8, -1]])
[1]
```

Attributes

proba_y	array, shape = nb of classes	probability of each class.
theta	array of shape nb_class*nb_features	mean of each feature for the different class
sigma	array of shape nb_class*nb_features	variance of each feature for the different class

Methods

fit(X, y)	self	Fit the model
predict(X)	array	Predict using the model.
predict_proba(X)	array	Predict the probability of each class using the model.

__init__()

score(*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

1.6.5 Nearest Neighbors

neighbors.NeighborsClassifier([n_neighbors, ...])	Classifier implementing k-Nearest Neighbor Algorithm.
neighbors.NeighborsRegressor([n_neighbors, ...])	Regression based on k-Nearest Neighbor Algorithm.
ball_tree.BallTree	Ball Tree for fast nearest-neighbor searches :

scikits.learn.neighbors.NeighborsClassifier

class scikits.learn.neighbors.**NeighborsClassifier**(*n_neighbors=5*, *algorithm='auto'*, *window_size=1*)

Classifier implementing k-Nearest Neighbor Algorithm.

Parameters **n_neighbors** : int, optional

Default number of neighbors. Defaults to 5.

window_size : int, optional

Window size passed to BallTree

algorithm : {'auto', 'ball_tree', 'brute', 'brute_inplace'}, optional

Algorithm used to compute the nearest neighbors. 'ball_tree' will construct a BallTree, 'brute' and 'brute_inplace' will perform brute-force search.'auto' will guess the most appropriate based on current dataset.

See Also:

BallTree

References

http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> samples = [[0, 0, 1], [1, 0, 0]]
>>> labels = [0, 1]
>>> from scikits.learn.neighbors import NeighborsClassifier
>>> neigh = NeighborsClassifier(n_neighbors=1)
>>> neigh.fit(samples, labels)
NeighborsClassifier(n_neighbors=1, window_size=1, algorithm='auto')
>>> print neigh.predict([[0,0,0]])
[1]
```

Methods

fit
kneighbors
predict
score

__init__ (*n_neighbors=5*, *algorithm='auto'*, *window_size=1*)

fit (*X*, *Y*, ***params*)

Fit the model using X, y as training data.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training data.

y : array-like, shape = [n_samples]

Target values, array of integer values.

params : list of keyword, optional

Overwrite keywords from __init__

kneighbors (*data*, *return_distance=True*, ***params*)

Finds the K-neighbors of a point.

Returns distance

Parameters **point** : array-like

The new point.

n_neighbors : int

Number of neighbors to get (default is the value passed to the constructor).

return_distance : boolean, optional. Defaults to True.

If False, distances will not be returned

Returns **dist** : array

Array representing the lengths to point, only present if return_distance=True

ind : array

Indices of the nearest points in the population matrix.

Examples

In the following example, we construnct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> labels = [0, 0, 1]
>>> from scikits.learn.neighbors import NeighborsClassifier
>>> neigh = NeighborsClassifier(n_neighbors=1)
>>> neigh.fit(samples, labels)
NeighborsClassifier(n_neighbors=1, window_size=1, algorithm='auto')
>>> print neigh.kneighbors([1., 1., 1.])
(array([ 0.5]), array([2]))
```

As you can see, it returns [0.5], and [2], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]])
```

predict (*X*, ***params*)

Predict the class labels for the provided data.

Parameters **X: array** :

A 2-D array representing the test point.

n_neighbors : int

Number of neighbors to get (default is the value passed to the constructor).

Returns **labels: array** :

List of class labels (one for each data sample).

score (*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

scikits.learn.neighbors.NeighborsRegressor

class scikits.learn.neighbors.**NeighborsRegressor**(*n_neighbors=5*, *mode='mean'*, *algorithm='auto'*, *window_size=1*)

Regression based on k-Nearest Neighbor Algorithm.

The target is predicted by local interpolation of the targets associated of the k-Nearest Neighbors in the training set.

Different modes for estimating the result can be set via parameter mode. 'barycenter' will apply the weights that best reconstruct the point from its neighbors while 'mean' will apply constant weights to each point.

Parameters **n_neighbors** : int, optional

Default number of neighbors. Defaults to 5.

window_size : int, optional

Window size passed to BallTree

mode : {'mean', 'barycenter'}, optional

Weights to apply to labels.

algorithm : {'auto', 'ball_tree', 'brute', 'brute_inplace'}, optional

Algorithm used to compute the nearest neighbors. 'ball_tree' will construct a BallTree, 'brute' and 'brute_inplace' will perform brute-force search.'auto' will guess the most appropriate based on current dataset.

Notes

http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from scikits.learn.neighbors import NeighborsRegressor
>>> neigh = NeighborsRegressor(n_neighbors=2)
>>> neigh.fit(X, y)
NeighborsRegressor(n_neighbors=2, window_size=1, mode='mean',
          algorithm='auto')
>>> print neigh.predict([[1.5]])
[ 0.5]
```

Methods

fit
kneighbors
predict
score

__init__ (*n_neighbors=5*, *mode='mean'*, *algorithm='auto'*, *window_size=1*)

fit (*X*, *Y*, ***params*)

Fit the model using X, y as training data.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training data.

y : array-like, shape = [n_samples]

Target values, array of integer values.

params : list of keyword, optional

Overwrite keywords from __init__

kneighbors (*data*, *return_distance=True*, ***params*)

Finds the K-neighbors of a point.

Returns distance

Parameters **point** : array-like

The new point.

n_neighbors : int

Number of neighbors to get (default is the value passed to the constructor).

return_distance : boolean, optional. Defaults to True.

If False, distances will not be returned

Returns **dist** : array

Array representing the lengths to point, only present if return_distance=True

ind : array

Indices of the nearest points in the population matrix.

Examples

In the following example, we construnct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> labels = [0, 0, 1]
>>> from scikits.learn.neighbors import NeighborsClassifier
>>> neigh = NeighborsClassifier(n_neighbors=1)
>>> neigh.fit(samples, labels)
NeighborsClassifier(n_neighbors=1, window_size=1, algorithm='auto')
>>> print neigh.kneighbors([1., 1., 1.])
(array([ 0.5]), array([2]))
```

As you can see, it returns [0.5], and [2], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]])
```

predict (*X*, ***params*)

Predict the target for the provided data.

Parameters **X** : array

A 2-D array representing the test data.

n_neighbors : int, optional

Number of neighbors to get (default is the value passed to the constructor).

Returns **y: array** :

List of target values (one for each data sample).

score (*X*, *y*)

Returns the mean error rate on the given test data and labels.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training set.

y : array-like, shape = [n_samples]

Labels for X.

Returns **z** : float

scikits.learn.ball_tree.BallTree

class scikits.learn.ball_tree.**BallTree**

Ball Tree for fast nearest-neighbor searches :

BallTree(M, leafsize=20)

Parameters **M** : array-like, shape = [N,D]

N is the number of points in the data set, and D is the dimension of the parameter space.

Note: if M is an aligned array of doubles (not necessarily contiguous) then data will not be copied. Otherwise, an internal copy will be made.

leafsize [positive integer (default = 20)] number of points at which to switch to brute-force

Methods

query
query_ball

__init__ ()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

data

View of data making up the Ball Tree

dim

Dimension of the Ball Tree

query (*x*, *k=1*, *return_distance=True*)

query the Ball Tree for the k nearest neighbors

Parameters **x** : array-like, last dimension self.dim

An array of points to query

k : integer (default = 1)

The number of nearest neighbors to return

return_distance : boolean (default = True)

if True, return a tuple (d,i) if False, return array i

Returns **i** : if return_distance == False

(d,i) : if return_distance == True

d : array of doubles - shape: x.shape[:-1] + (k,)

each entry gives the list of distances to the neighbors of the corresponding point (note that distances are not sorted)

i : array of integers - shape: x.shape[:-1] + (k,)

each entry gives the list of indices of neighbors of the corresponding point (note that neighbors are not sorted)

query_ball (*x*, *r*, *count_only = False*)

query the Ball Tree for the k nearest neighbors

Parameters **x** : array-like, last dimension self.dim

An array of points to query

r : floating-point value

Radius around each point within which all neighbors are returned

count_only : boolean (default = False)

if True, return count of neighbors for each point

if False, return full list of neighbors for each point

Returns **i** : array of integers, shape: x.shape[:-1]

if count_only is False each entry gives the list of neighbors of the corresponding point (note that neighbors are not sorted). Otherwise return only the number of neighbors.

size

Number of points in the Ball Tree

neighbors.kneighbors_graph(X, n_neighbors[, ...])	Computes the (weighted) graph of k-Neighbors for points in X
ball_tree.knn_brute(x, pt[, k])	Brute-Force k-nearest neighbor search.

scikits.learn.neighbors.kneighbors_graph

scikits.learn.neighbors.**kneighbors_graph** (*X*, *n_neighbors*, *mode='connectivity'*)

Computes the (weighted) graph of k-Neighbors for points in X

Parameters **X** : array-like, shape = [n_samples, n_features]

Coordinates of samples. One sample per row.

n_neighbors : int

Number of neighbors for each sample.

mode : {'connectivity', 'distance', 'barycenter'}

Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are euclidian distance between points. In 'barycenter' they are barycenter weights estimated by solving a linear system for each point.

Returns **A** : CSR sparse matrix, shape = [n_samples, n_samples]

A[i,j] is assigned the weight of edge that connects i to j.

Examples

```
>>> X = [[0], [3], [1]]
>>> from scikits.learn.neighbors import kneighbors_graph
>>> A = kneighbors_graph(X, 2)
>>> A.todense()
matrix([[ 1.,  0.,  1.],
        [ 0.,  1.,  1.],
        [ 1.,  0.,  1.]])
```

scikits.learn.ball_tree.knn_brute

scikits.learn.ball_tree.**knn_brute**(*x*, *pt*, *k=1*)

Brute-Force k-nearest neighbor search.

Parameters **x** : array of shape [N,D]

representing N points in D dimensions

pt : array-like, last dimension D

An array of points to query

k : a positive integer, giving the number of nearest

neighbors to query

Returns **nbrs** : array of integers - shape: pt.shape[:-1] + (k,)

each entry gives the list of indices of neighbors of the corresponding point

1.6.6 Gaussian Mixture Models

mixture.GMM([n_states, cvtype])	Gaussian Mixture Model

scikits.learn.mixture.GMM

class scikits.learn.mixture.**GMM**(*n_states=1*, *cvtype='diag'*)

Gaussian Mixture Model

Representation of a Gaussian mixture model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a GMM distribution.

Initializes parameters such that every mixture component has zero mean and identity covariance.

Parameters **n_states** : int, optional

Number of mixture components. Defaults to 1.

cvtype : string (read-only), optional

String describing the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'. Defaults to 'diag'.

Examples

```
>>> import numpy as np
>>> from scikits.learn import mixture
>>> g = mixture.GMM(n_states=2)

>>> # Generate random observations with two modes centered on 0
>>> # and 10 to use for training.
>>> np.random.seed(0)
>>> obs = np.concatenate((np.random.randn(100, 1),
...                       10 + np.random.randn(300, 1)))
>>> g.fit(obs)
GMM(cvtype='diag', n_states=2)
>>> g.weights
array([ 0.25,  0.75])
>>> g.means
array([[ 0.05980802],
       [ 9.94199467]])
>>> g.covars
[array([[ 1.01682662]]), array([[ 0.96080513]])]
>>> np.round(g.weights, 2)
array([ 0.25,  0.75])
>>> np.round(g.means, 2)
array([[ 0.06],
       [ 9.94]])
>>> np.round(g.covars, 2)
...
array([[[ 1.02]],
       [[ 0.96]]])
>>> g.predict([[0], [2], [9], [10]])
array([0, 0, 1, 1])
>>> np.round(g.score([[0], [2], [9], [10]]), 2)
array([-2.32, -4.16, -1.65, -1.19])

>>> # Refit the model on new data (initial parameters remain the
>>> # same), this time with an even split between the two modes.
>>> g.fit(20 * [[0]] +  20 * [[10]])
GMM(cvtype='diag', n_states=2)
>>> np.round(g.weights, 2)
array([ 0.5,  0.5])
```

Attributes

cvtype
n_states
weights
means
covars

n_features	int	Dimensionality of the Gaussians.

Methods

decode(X)	Find most likely mixture components for each point in *X*.
eval(X)	Compute the log likelihood of *X* under the model and the posterior distribution over mixture components.
fit(X)	Estimate model parameters from *X* using the EM algorithm.
predict(X)	Like decode, find most likely mixtures components for each observation in *X*.
rvs(n=1)	Generate *n* samples from the model.
score(X)	Compute the log likelihood of *X* under the model.

__init__ (*n_states=1*, *cvtype='diag'*)

covars
Return covars as a full matrix.

cvtype
Covariance type of the model.

Must be one of 'spherical', 'tied', 'diag', 'full'.

decode (*obs*)
Find most likely mixture components for each point in *obs*.

Parameters **obs** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns **logprobs** : array_like, shape (n_samples,)

Log probability of each point in *obs* under the model.

components : array_like, shape (n_samples,)

Index of the most likelihod mixture components for each observation

eval (*obs*)
Evaluate the model on data

Compute the log probability of *obs* under the model and return the posterior distribution (responsibilities) of each mixture component for each element of *obs*.

Parameters **obs** : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns **logprob** : array_like, shape (n_samples,)

Log probabilities of each data point in *obs*

posteriors: array_like, shape (n_samples, n_states) :

Posterior probabilities of each mixture component for each observation

fit (*X*, *n_iter=10*, *min_covar=0.001*, *thresh=0.01*, *params='wmc'*, *init_params='wmc'*)
Estimate model parameters with the expectation-maximization algorithm.

A initialization step is performed before entering the em algorithm. If you want to avoid this step, set the keyword argument init_params to the empty string ''. Likewise, if you would like just to do an initialization, call this method with n_iter=0.

Parameters **X** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

n_iter : int, optional

Number of EM iterations to perform.

min_covar : float, optional

Floor on the diagonal of the covariance matrix to prevent overfitting. Defaults to 1e-3.

thresh : float, optional

Convergence threshold.

params : string, optional

Controls which parameters are updated in the training process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars. Defaults to 'wmc'.

init_params : string, optional

Controls which parameters are updated in the initialization process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars. Defaults to 'wmc'.

means

Mean parameters for each mixture component.

n_states

Number of mixture components in the model.

predict(*X*)

Predict label for data.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **C** : array, shape = (n_samples,)

predict_proba(*X*)

Predict posterior probability of data under each Gaussian in the model.

Parameters **X** : array-like, shape = [n_samples, n_features]

Returns **T** : array-like, shape = (n_samples, n_states)

Returns the probability of the sample for each Gaussian (state) in the model.

rvs(*n_samples=1*)

Generate random samples from the model.

Parameters **n_samples** : int, optional

Number of samples to generate. Defaults to 1.

Returns **obs** : array_like, shape (n_samples, n_features)

List of samples

score(*obs*)

Compute the log probability under the model.

Parameters **obs** : array_like, shape (n_samples, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns **logprob** : array_like, shape (n_samples,)

Log probabilities of each data point in *obs*

weights

Mixing weights for each mixture component.

1.6.7 Hidden Markov Models

hmm.GaussianHMM([n_states, cvtype, ...])	Hidden Markov Model with Gaussian emissions
hmm.MultinomialHMM([n_states, startprob, ...])	Hidden Markov Model with multinomial (discrete) emissions
hmm.GMMHMM([n_states, n_mix, startprob, ...])	Hidden Markov Model with Gaussin mixture emissions

scikits.learn.hmm.GaussianHMM

class scikits.learn.hmm.**GaussianHMM**(*n_states=1*, *cvtype='diag'*, *startprob=None*, *transmat=None*, *startprob_prior=None*, *transmat_prior=None*, *means_prior=None*, *means_weight=0*, *covars_prior=0.01*, *covars_weight=1*)

Hidden Markov Model with Gaussian emissions

Representation of a hidden Markov model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a HMM.

See Also:

GMM Gaussian mixture model

Examples

```
>>> from scikits.learn.hmm import GaussianHMM
>>> GaussianHMM(n_states=2)
GaussianHMM(cvtype='diag', n_states=2, means_weight=0, startprob_prior=1.0,
      startprob=array([ 0.5,  0.5]),
      transmat=array([[ 0.5,  0.5],
       [ 0.5,  0.5]]),
      transmat_prior=1.0, means_prior=None, covars_weight=1,
      covars_prior=0.01)
```

Attributes

cvtype
n_states
transmat
startprob
means
covars

n_features	int (read-only)	Dimensionality of the Gaussian emissions.

Methods

eval(X)	Compute the log likelihood of *X* under the HMM.
decode(X)	Find most likely state sequence for each point in *X* using the Viterbi algorithm.
rvs(n=1)	Generate *n* samples from the HMM.
init(X)	Initialize HMM parameters from *X*.
fit(X)	Estimate HMM parameters from *X* using the Baum-Welch algorithm.
predict(X)	Like decode, find most likely state sequence corresponding to *X*.
score(X)	Compute the log likelihood of *X* under the model.

__init__(*n_states=1*, *cvtype='diag'*, *startprob=None*, *transmat=None*, *startprob_prior=None*, *transmat_prior=None*, *means_prior=None*, *means_weight=0*, *covars_prior=0.01*, *covars_weight=1*)

Create a hidden Markov model with Gaussian emissions.

Initializes parameters such that every state has zero mean and identity covariance.

Parameters **n_states** : int

Number of states.

cvtype : string

String describing the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'. Defaults to 'diag'.

covars

Return covars as a full matrix.

cvtype

Covariance type of the model.

Must be one of 'spherical', 'tied', 'diag', 'full'.

decode(*obs*, *maxrank=None*, *beamlogprob=-inf*)

Find most likely state sequence corresponding to *obs*.

Uses the Viterbi algorithm.

Parameters **obs** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **viterbi_logprob** : float

Log probability of the maximum likelihood path through the HMM

states : array_like, shape (n,)

Index of the most likely states for each observation

See Also:

eval Compute the log probability under the model and posteriors

score Compute the log probability under the model

eval(*obs*, *maxrank=None*, *beamlogprob=-inf*)
Compute the log probability under the model and compute posteriors

Implements rank and beam pruning in the forward-backward algorithm to speed up inference in large models.

Parameters **obs** : array_like, shape (n, n_features)

Sequence of n_features-dimensional data points. Each row corresponds to a single point in the sequence.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **logprob** : array_like, shape (n,)

Log probabilities of the sequence *obs*

posteriors: array_like, shape (n, n_states) :

Posterior probabilities of each state for each observation

See Also:

score Compute the log probability under the model

decode Find most likely state sequence corresponding to a *obs*

fit(*obs*, *n_iter=10*, *thresh=0.01*, *params='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'*, *init_params='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'*, *maxrank=None*, *beamlogprob=-inf*, ***kwargs*)
Estimate model parameters.

An initialization step is performed before entering the EM algorithm. If you want to avoid this step, set the keyword argument init_params to the empty string ''. Likewise, if you would like just to do an initialization, call this method with n_iter=0.

Parameters **obs** : list

List of array-like observation sequences (shape (n_i, n_features)).

n_iter : int, optional

Number of iterations to perform.

thresh : float, optional

Convergence threshold.

params : string, optional

Controls which parameters are updated in the training process. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

init_params : string, optional

Controls which parameters are initialized prior to training. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

maxrank : int, optional

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See "The HTK Book" for more details.

beamlogprob : float, optional

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See "The HTK Book" for more details.

Notes

In general, *logprob* should be non-decreasing unless aggressive pruning is used. Decreasing *logprob* is generally a sign of overfitting (e.g. a covariance parameter getting too small). You can fix this by getting more training data, or decreasing *covars_prior*.

means
Mean parameters for each state.

n_states
Number of states in the model.

predict(*obs*, ***kwargs*)
Find most likely state sequence corresponding to *obs*.

Parameters **obs** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **states** : array_like, shape (n,)

Index of the most likely states for each observation

predict_proba(*obs*, ***kwargs*)
Compute the posterior probability for each state in the model

Parameters **obs** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

See eval() for a list of accepted keyword arguments. :

Returns **T** : array-like, shape (n, n_states)

Returns the probability of the sample for each state in the model.

rvs(*n=1*)
Generate random samples from the model.

Parameters **n** : int

Number of samples to generate.

Returns **obs** : array_like, length *n*

List of samples

score(*obs*, *maxrank=None*, *beamlogprob=-inf*)
Compute the log probability under the model.

Parameters **obs** : array_like, shape (n, n_features)

Sequence of n_features-dimensional data points. Each row corresponds to a single data point.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **logprob** : array_like, shape (n,)

Log probabilities of each data point in *obs*

See Also:

eval Compute the log probability under the model and posteriors

decode Find most likely state sequence corresponding to a *obs*

startprob
Mixing startprob for each state.

transmat
Matrix of transition probabilities.

scikits.learn.hmm.MultinomialHMM

class scikits.learn.hmm.**MultinomialHMM**(*n_states=1*, *startprob=None*, *transmat=None*, *startprob_prior=None*, *transmat_prior=None*)
Hidden Markov Model with multinomial (discrete) emissions

See Also:

GaussianHMM HMM with Gaussian emissions

Examples

```
>>> from scikits.learn.hmm import MultinomialHMM
>>> MultinomialHMM(n_states=2)
...
MultinomialHMM(transmat=array([[ 0.5,  0.5],
       [ 0.5,  0.5]]),
```

```
 startprob_prior=1.0, n_states=2, startprob=array([ 0.5,  0.5]),
transmat_prior=1.0)
```

Attributes

n_states
transmat
startprob

n_symbols emissionprob: array, shape (‘n_states‘, ‘n_symbols‘)	int	Number of possible symbols emitted by the model (in the observations). Probability of emitting a given symbol when in each state.

Methods

eval(X)	Compute the log likelihood of *X* under the HMM.
decode(X)	Find most likely state sequence for each point in *X* using the Viterbi algorithm.
rvs(n=1)	Generate *n* samples from the HMM.
init(X)	Initialize HMM parameters from *X*.
fit(X)	Estimate HMM parameters from *X* using the Baum-Welch algorithm.
predict(X)	Like decode, find most likely state sequence corresponding to *X*.
score(X)	Compute the log likelihood of *X* under the model.

__init__(*n_states=1*, *startprob=None*, *transmat=None*, *startprob_prior=None*, *transmat_prior=None*)

Create a hidden Markov model with multinomial emissions.

Parameters **n_states** : int

Number of states.

decode(*obs*, *maxrank=None*, *beamlogprob=-inf*)

Find most likely state sequence corresponding to *obs*.

Uses the Viterbi algorithm.

Parameters **obs** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **viterbi_logprob** : float

Log probability of the maximum likelihood path through the HMM

states : array_like, shape (n,)

Index of the most likely states for each observation

See Also:

eval Compute the log probability under the model and posteriors

score Compute the log probability under the model

emissionprob
Emission probability distribution for each state.

eval(*obs*, *maxrank=None*, *beamlogprob=-inf*)
Compute the log probability under the model and compute posteriors

Implements rank and beam pruning in the forward-backward algorithm to speed up inference in large models.

Parameters **obs** : array_like, shape (n, n_features)

Sequence of n_features-dimensional data points. Each row corresponds to a single point in the sequence.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **logprob** : array_like, shape (n,)

Log probabilities of the sequence *obs*

posteriors: array_like, shape (n, n_states) :

Posterior probabilities of each state for each observation

See Also:

score Compute the log probability under the model

decode Find most likely state sequence corresponding to a *obs*

fit(*obs*, *n_iter=10*, *thresh=0.01*, *params='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'*, *init_params='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'*, *maxrank=None*, *beamlogprob=-inf*, ***kwargs*)
Estimate model parameters.

An initialization step is performed before entering the EM algorithm. If you want to avoid this step, set the keyword argument init_params to the empty string ''. Likewise, if you would like just to do an initialization, call this method with n_iter=0.

Parameters **obs** : list

List of array-like observation sequences (shape (n_i, n_features)).

n_iter : int, optional

Number of iterations to perform.

thresh : float, optional

Convergence threshold.

params : string, optional

Controls which parameters are updated in the training process. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

init_params : string, optional

Controls which parameters are initialized prior to training. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

maxrank : int, optional

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See "The HTK Book" for more details.

beamlogprob : float, optional

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See "The HTK Book" for more details.

Notes

In general, *logprob* should be non-decreasing unless aggressive pruning is used. Decreasing *logprob* is generally a sign of overfitting (e.g. a covariance parameter getting too small). You can fix this by getting more training data, or decreasing *covars_prior*.

n_states
Number of states in the model.

predict(*obs*, ***kwargs*)
Find most likely state sequence corresponding to *obs*.

Parameters **obs** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **states** : array_like, shape (n,)

Index of the most likely states for each observation

predict_proba(*obs*, ***kwargs*)
Compute the posterior probability for each state in the model

Parameters **obs** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

See eval() for a list of accepted keyword arguments. :

Returns **T** : array-like, shape (n, n_states)

Returns the probability of the sample for each state in the model.

rvs (*n=1*)
Generate random samples from the model.

Parameters **n** : int

Number of samples to generate.

Returns **obs** : array_like, length *n*

List of samples

score (*obs*, *maxrank=None*, *beamlogprob=-inf*)
Compute the log probability under the model.

Parameters **obs** : array_like, shape (n, n_features)

Sequence of n_features-dimensional data points. Each row corresponds to a single data point.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **logprob** : array_like, shape (n,)

Log probabilities of each data point in *obs*

See Also:

eval Compute the log probability under the model and posteriors

decode Find most likely state sequence corresponding to a *obs*

startprob
Mixing startprob for each state.

transmat
Matrix of transition probabilities.

scikits.learn.hmm.GMMHMM

class scikits.learn.hmm.**GMMHMM**(*n_states=1*, *n_mix=1*, *startprob=None*, *transmat=None*, *startprob_prior=None*, *transmat_prior=None*, *gmms=None*, *cvtype=None*)
Hidden Markov Model with Gaussin mixture emissions

See Also:

GaussianHMM HMM with Gaussian emissions

Examples

```
>>> from scikits.learn.hmm import GMMHMM
>>> GMMHMM(n_states=2, n_mix=10, cvtype='diag')
...
GMMHMM(n_mix=10, cvtype='diag', n_states=2, startprob_prior=1.0,
    startprob=array([ 0.5,  0.5]),
    transmat=array([[ 0.5,  0.5],
       [ 0.5,  0.5]]),
    transmat_prior=1.0,
    gmms=[GMM(cvtype='diag', n_states=10), GMM(cvtype='diag',
          n_states=10)])
```

Attributes

n_states
transmat
startprob

gmms: array of GMM objects, length 'n_states'	GMM emission distributions for each state

Methods

eval(X)	Compute the log likelihood of *X* under the HMM.
decode(X)	Find most likely state sequence for each point in *X* using the Viterbi algorithm.
rvs(n=1)	Generate *n* samples from the HMM.
init(X)	Initialize HMM parameters from *X*.
fit(X)	Estimate HMM parameters from *X* using the Baum-Welch algorithm.
predict(X)	Like decode, find most likely state sequence corresponding to *X*.
score(X)	Compute the log likelihood of *X* under the model.

__init__(*n_states=1*, *n_mix=1*, *startprob=None*, *transmat=None*, *startprob_prior=None*, *transmat_prior=None*, *gmms=None*, *cvtype=None*)

Create a hidden Markov model with GMM emissions.

Parameters **n_states** : int

Number of states.

decode(*obs*, *maxrank=None*, *beamlogprob=-inf*)

Find most likely state sequence corresponding to *obs*.

Uses the Viterbi algorithm.

Parameters **obs** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **viterbi_logprob** : float

Log probability of the maximum likelihood path through the HMM

states : array_like, shape (n,)

Index of the most likely states for each observation

See Also:

eval Compute the log probability under the model and posteriors

score Compute the log probability under the model

eval (*obs*, *maxrank=None*, *beamlogprob=-inf*)

Compute the log probability under the model and compute posteriors

Implements rank and beam pruning in the forward-backward algorithm to speed up inference in large models.

Parameters **obs** : array_like, shape (n, n_features)

Sequence of n_features-dimensional data points. Each row corresponds to a single point in the sequence.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **logprob** : array_like, shape (n,)

Log probabilities of the sequence *obs*

posteriors: array_like, shape (n, n_states) :

Posterior probabilities of each state for each observation

See Also:

score Compute the log probability under the model

decode Find most likely state sequence corresponding to a *obs*

fit (*obs*, *n_iter=10*, *thresh=0.01*, *params='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'*, *init_params='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'*, *maxrank=None*, *beamlogprob=-inf*, ***kwargs*)

Estimate model parameters.

An initialization step is performed before entering the EM algorithm. If you want to avoid this step, set the keyword argument init_params to the empty string ''. Likewise, if you would like just to do an initialization, call this method with n_iter=0.

Parameters **obs** : list

List of array-like observation sequences (shape (n_i, n_features)).

n_iter : int, optional

Number of iterations to perform.

thresh : float, optional

Convergence threshold.

params : string, optional

Controls which parameters are updated in the training process. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

init_params : string, optional

Controls which parameters are initialized prior to training. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

maxrank : int, optional

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See "The HTK Book" for more details.

beamlogprob : float, optional

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See "The HTK Book" for more details.

Notes

In general, *logprob* should be non-decreasing unless aggressive pruning is used. Decreasing *logprob* is generally a sign of overfitting (e.g. a covariance parameter getting too small). You can fix this by getting more training data, or decreasing *covars_prior*.

n_states
Number of states in the model.

predict(*obs*, ***kwargs*)
Find most likely state sequence corresponding to *obs*.

Parameters **obs** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **states** : array_like, shape (n,)

Index of the most likely states for each observation

predict_proba(*obs*, ***kwargs*)
Compute the posterior probability for each state in the model

Parameters **obs** : array_like, shape (n, n_features)

List of n_features-dimensional data points. Each row corresponds to a single data point.

See eval() for a list of accepted keyword arguments. :

Returns **T** : array-like, shape (n, n_states)

Returns the probability of the sample for each state in the model.

rvs (*n=1*)

Generate random samples from the model.

Parameters **n** : int

Number of samples to generate.

Returns **obs** : array_like, length *n*

List of samples

score (*obs*, *maxrank=None*, *beamlogprob=-inf*)

Compute the log probability under the model.

Parameters **obs** : array_like, shape (n, n_features)

Sequence of n_features-dimensional data points. Each row corresponds to a single data point.

maxrank : int

Maximum rank to evaluate for rank pruning. If not None, only consider the top *maxrank* states in the inner sum of the forward algorithm recursion. Defaults to None (no rank pruning). See The HTK Book for more details.

beamlogprob : float

Width of the beam-pruning beam in log-probability units. Defaults to -numpy.Inf (no beam pruning). See The HTK Book for more details.

Returns **logprob** : array_like, shape (n,)

Log probabilities of each data point in *obs*

See Also:

eval Compute the log probability under the model and posteriors

decode Find most likely state sequence corresponding to a *obs*

startprob

Mixing startprob for each state.

transmat

Matrix of transition probabilities.

1.6.8 Clustering

cluster.KMeans([k, init, n_init, max_iter, ...])	K-Means clustering
cluster.MeanShift([bandwidth])	MeanShift clustering
cluster.SpectralClustering([k, mode])	Spectral clustering: apply k-means to a projection of the graph laplacian, finds normalized graph cuts.
cluster.AffinityPropagation([damping, ...])	Perform Affinity Propagation Clustering of data

scikits.learn.cluster.KMeans

class scikits.learn.cluster.**KMeans**(*k=8*, *init='random'*, *n_init=10*, *max_iter=300*, *tol=0.0001*, *verbose=0*, *rng=None*, *copy_x=True*)

K-Means clustering

Parameters **data** : ndarray

A M by N array of M observations in N dimensions or a length M array of M one-dimensional observations.

k : int or ndarray

The number of clusters to form as well as the number of centroids to generate. If init initialization string is 'matrix', or if a ndarray is given instead, it is interpreted as initial cluster to use instead.

max_iter : int

Maximum number of iterations of the k-means algorithm for a single run.

n_init: int, optional, default: 10 :

Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

init : {'k-means++', 'random', 'points', 'matrix'}

Method for initialization, defaults to 'random':

'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

'random': generate k centroids from a Gaussian with mean and variance estimated from the data.

'points': choose k observations (rows) at random from data for the initial centroids.

'matrix': interpret the k parameter as a k by M (or length k array for one-dimensional data) array of initial centroids.

tol: float, optional default: 1e-4 :

Relative tolerance w.r.t. inertia to declare convergence

Notes

The k-means problem is solved using the Lloyd algorithm.

The average complexity is given by O(k n T), were n is the number of samples and T is the number of iteration.

The worst case complexity is given by O(n^(k+2/p)) with n = n_samples, p = n_features. (D. Arthur and S. Vassilvitskii, 'How slow is the k-means method?' SoCG2006)

In practice, the K-means algorithm is very fast (one of the fastest clustering algorithms available), but it falls in local minima. That's why it can be useful to restart it several times.

Attributes

cluster_centers_: array, [n_clusters, n_features]	Coordinates of cluster centers
labels_:	Labels of each point
inertia_: float	The value of the inertia criterion associated with the chosen partition.

Methods

fit(X):	Compute K-Means clustering

__init__ (*k=8*, *init='random'*, *n_init=10*, *max_iter=300*, *tol=0.0001*, *verbose=0*, *rng=None*, *copy_x=True*)

fit (*X*, ***params*)
Compute k-means

scikits.learn.cluster.MeanShift

class scikits.learn.cluster.**MeanShift** (*bandwidth=None*)
MeanShift clustering

Parameters **bandwidth: float, optional** :

Bandwith used in the RBF kernel If not set, the bandwidth is estimated. See clustering.estimate_bandwidth

Notes

Reference:

K. Funkunaga and L.D. Hosteler, "The Estimation of the Gradient of a Density Function, with Applications in Pattern Recognition"

The algorithmic complexity of the mean shift algorithm is O(T n^2) with n the number of samples and T the number of iterations. It is not adviced for a large number of samples.

Attributes

cluster_centers_: array, [n_clusters, n_features]	Coordinates of cluster centers
labels_:	Labels of each point

Methods

fit(X):	Compute MeanShift clustering

__init__ (*bandwidth=None*)

fit (*X*, ***params*)
Compute MeanShift

Parameters **X** : array [n_samples, n_features]

Input points

scikits.learn.cluster.SpectralClustering

class scikits.learn.cluster.**SpectralClustering** (*k=8*, *mode=None*)
Spectral clustering: apply k-means to a projection of the graph laplacian, finds normalized graph cuts.

Parameters **k: integer, optional** :

The dimension of the projection subspace.

mode: {None, 'arpack' or 'amg'} :

The eigenvalue decomposition strategy to use. AMG (Algebraic MultiGrid) is much faster, but requires pyamg to be installed.

Attributes

labels_:	Labels of each point

Methods

fit(X):	Compute spectral clustering

__init__ (*k=8*, *mode=None*)

fit (*X*, ***params*)
Compute the spectral clustering from the adjacency matrix of the graph.

Parameters **X: array-like or sparse matrix, shape: (p, p)** :

The adjacency matrix of the graph to embed. X is an adjacency matrix of a similarity graph: its entries must be positive or zero. Zero means that elements have nothing in common, whereas high values mean that elements are strongly similar.

Notes

If you have an affinity matrix, such as a distance matrix, for which 0 means identical elements, and high values means very dissimilar elements, it can be transformed in a similarity matrix that is well suited for the algorithm by applying the gaussian (heat) kernel:

```
np.exp(- X**2/2. * delta**2)
```

If the pyamg package is installed, it is used. This greatly speeds up computation.

scikits.learn.cluster.AffinityPropagation

class scikits.learn.cluster.**AffinityPropagation**(*damping=0.5*, *max_iter=200*, *convit=30*, *copy=True*)

Perform Affinity Propagation Clustering of data

Parameters **damping** : float, optional

Damping factor

max_iter : int, optional

Maximum number of iterations

convit : int, optional

Number of iterations with no change in the number of estimated clusters that stops the convergence.

copy: boolean, optional :

Make a copy of input data. True by default.

Notes

See examples/plot_affinity_propagation.py for an example.

Reference:

Brendan J. Frey and Delbert Dueck, "Clustering by Passing Messages Between Data Points", Science Feb. 2007

The algorithmic complexity of affinity propagation is quadratic in the number of points.

Attributes

cluster_centers_indices_	array, [n_clusters]	Indices of cluster centers
labels_	array, [n_samples]	Labels of each point

Methods

fit:	Compute the clustering

__init__(*damping=0.5*, *max_iter=200*, *convit=30*, *copy=True*)

fit(*S*, *p=None*, ***params*)

compute MeanShift

Parameters **S: array [n_points, n_points]** :

Matrix of similarities between points

p: array [n_points,] or float, optional :

Preferences for each point

damping : float, optional

Damping factor

copy: boolean, optional :

If copy is False, the affinity matrix is modified inplace by the algorithm, for memory efficiency

1.6.9 Metrics

metrics.euclidean_distances(X, Y[, ...])	Considering the rows of X (and Y=X) as vectors, compute the
metrics.unique_labels	
metrics.confusion_matrix(y_true, y_pred[, ...])	Compute confusion matrix to evaluate the accuracy of a classification
metrics.roc_curve(y, probas_)	compute Receiver operating characteristic (ROC)
metrics.auc(x, y)	Compute Area Under the Curve (AUC) using the trapezoidal rule
metrics.precision_score(y_true, y_pred[, ...])	Compute the precision
metrics.recall_score(y_true, y_pred[, pos_label])	Compute the recall
metrics.fbeta_score(y_true, y_pred, beta[, ...])	Compute fbeta score
metrics.f1_score(y_true, y_pred[, pos_label])	Compute f1 score
metrics.precision_recall_fscore_support(y, ...)	Compute precisions, recalls, f-measures and support for each class
metrics.classification_report(y_true, y_pred)	Build a text report showing the main classification metrics
metrics.precision_recall_curve(y_true, ...)	Compute precision-recall pairs for different probability thresholds
metrics.r2_score(y_true, y_pred)	R^2 (coefficient of determination) regression score function
metrics.zero_one_score(y_true, y_pred)	Zero-One classification score
metrics.zero_one(y_true, y_pred)	Zero-One classification loss
metrics.mean_square_error(y_true, y_pred)	Mean square error regression loss

scikits.learn.metrics.euclidean_distances

scikits.learn.metrics.**euclidean_distances**(*X*, *Y*, *Y_norm_squared=None*, *squared=False*)

Considering the rows of X (and Y=X) as vectors, compute the distance matrix between each pair of vectors.

Parameters **X: array of shape (n_samples_1, n_features)** :

Y: array of shape (n_samples_2, n_features) :

Y_norm_squared: array [n_samples_2], optional :

pre-computed (Y**2).sum(axis=1)

squared: boolean, optional :

This routine will return squared Euclidean distances instead.

Returns **distances: array of shape (n_samples_1, n_samples_2)** :

Examples

```
>>> from scikits.learn.metrics.pairwise import euclidean_distances
>>> X = [[0, 1], [1, 1]]
>>> # distrance between rows of X
>>> euclidean_distances(X, X)
array([[ 0.,  1.],
       [ 1.,  0.]])
>>> # get distance to origin
>>> euclidean_distances(X, [[0, 0]])
array([[ 1.        ],
       [ 1.41421356]])
```

scikits.learn.metrics.confusion_matrix

scikits.learn.metrics.**confusion_matrix**(*y_true*, *y_pred*, *labels=None*)

Compute confusion matrix to evaluate the accuracy of a classification

By definition a confusion matrix cm is such that cm[i, j] is equal to the number of observations known to be in group i but predicted to be in group j

Parameters **y_true** : array, shape = [n_samples]

true targets

y_pred : array, shape = [n_samples]

estimated targets

Returns **CM** : array, shape = [n_classes, n_classes]

confusion matrix

References

http://en.wikipedia.org/wiki/Confusion_matrix

scikits.learn.metrics.roc_curve

scikits.learn.metrics.**roc_curve**(*y*, *probas_*)

compute Receiver operating characteristic (ROC)

Parameters **y** : array, shape = [n_samples]

true targets

probas_ : array, shape = [n_samples]

estimated probabilities

Returns **fpr** : array, shape = [n]

False Positive Rates

tpr : array, shape = [n]

True Positive Rates

thresholds : array, shape = [n]

Thresholds on **proba_** used to compute fpr and tpr

References

http://en.wikipedia.org/wiki/Receiver_operating_characteristic

scikits.learn.metrics.auc

scikits.learn.metrics.**auc**(*x*, *y*)

Compute Area Under the Curve (AUC) using the trapezoidal rule

Parameters **x** : array, shape = [n]

x coordinates

y : array, shape = [n]

y coordinates

Returns **auc** : float

scikits.learn.metrics.precision_score

scikits.learn.metrics.**precision_score**(*y_true*, *y_pred*, *pos_label=1*)

Compute the precision

The precision is the ratio $tp/(tp + fp)$ where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The best value is 1 and the worst value is 0.

Parameters **y_true** : array, shape = [n_samples]

true targets

y_pred : array, shape = [n_samples]

predicted targets

pos_label : int

in the binary classification case, give the label of the positive class (default is 1)

Returns **precision** : float

precision of the positive class in binary classification or weighted avergage of the precision of each class for the multiclass task

scikits.learn.metrics.recall_score

scikits.learn.metrics.**recall_score**(*y_true*, *y_pred*, *pos_label=1*)

Compute the recall

The recall is the ratio $tp/(tp+fn)$ where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The best value is 1 and the worst value is 0.

Parameters **y_true** : array, shape = [n_samples]

true targets

y_pred : array, shape = [n_samples]

predicted targets

pos_label : int

in the binary classification case, give the label of the positive class (default is 1)

Returns **recall** : float

recall of the positive class in binary classification or weighted avergage of the recall of each class for the multiclass task

scikits.learn.metrics.fbeta_score

scikits.learn.metrics.**fbeta_score**(*y_true*, *y_pred*, *beta*, *pos_label=1*)

Compute fbeta score

The F_beta score can be interpreted as a weighted average of the precision and recall, where an F_beta score reaches its best value at 1 and worst score at 0.

F_1 weights recall beta as much as precision.

See: http://en.wikipedia.org/wiki/F1_score

Parameters **y_true** : array, shape = [n_samples]

true targets

y_pred : array, shape = [n_samples]

predicted targets

beta: float :

pos_label : int

in the binary classification case, give the label of the positive class (default is 1)

Returns **fbeta_score** : float

fbeta_score of the positive class in binary classification or weighted avergage of the fbeta_score of each class for the multiclass task

scikits.learn.metrics.f1_score

scikits.learn.metrics.**f1_score**(*y_true*, *y_pred*, *pos_label=1*)

Compute f1 score

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the f1 score are equal.

F_1 = 2 * (precision * recall) / (precision + recall)

See: http://en.wikipedia.org/wiki/F1_score

In the multi-class case, this is the weighted average of the f1-score of each class.

Parameters **y_true** : array, shape = [n_samples]

true targets

y_pred : array, shape = [n_samples]

predicted targets

pos_label : int

in the binary classification case, give the label of the positive class (default is 1)

Returns **f1_score** : float

f1_score of the positive class in binary classification or weighted avergage of the f1_scores of each class for the multiclass task

References

http://en.wikipedia.org/wiki/F1_score

scikits.learn.metrics.precision_recall_fscore_support

scikits.learn.metrics.**precision_recall_fscore_support**(*y_true*, *y_pred*, *beta=1.0*, *labels=None*)

Compute precisions, recalls, f-measures and support for each class

The precision is the ratio $tp/(tp + fp)$ where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio $tp/(tp+fn)$ where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The F_beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F_beta score reaches its best value at 1 and worst score at 0.

The F_beta score weights recall beta as much as precision. beta = 1.0 means recall and precsion are as important.

The support is the number of occurrences of each class in y_true.

Parameters **y_true** : array, shape = [n_samples]

true targets

y_pred : array, shape = [n_samples]

predicted targets

beta : float, 1.0 by default

the strength of recall versus precision in the f-score

Returns **precision: array, shape = [n_unique_labels], dtype = np.double** :

recall: array, shape = [n_unique_labels], dtype = np.double :

f1_score: array, shape = [n_unique_labels], dtype = np.double :

support: array, shape = [n_unique_labels], dtype = np.long :

References

http://en.wikipedia.org/wiki/Precision_and_recall

scikits.learn.metrics.classification_report

scikits.learn.metrics.**classification_report**(*y_true*, *y_pred*, *labels=None*, *class_names=None*)

Build a text report showing the main classification metrics

Parameters **y_true** : array, shape = [n_samples]

true targets

y_pred : array, shape = [n_samples]

estimated targets

labels : array, shape = [n_labels]

optional list of label indices to include in the report

class_names : list of strings

optional display names matching the labels (same order)

Returns **report** : string

Text summary of the precision, recall, f1-score for each class

scikits.learn.metrics.precision_recall_curve

scikits.learn.metrics.**precision_recall_curve**(*y_true*, *probas_pred*)

Compute precision-recall pairs for different probability thresholds

Note: this implementation is restricted to the binary classification task.

The precision is the ratio $tp/(tp + fp)$ where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio $tp/(tp+fn)$ where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

Parameters **y_true** : array, shape = [n_samples]

true targets of binary classification in range {-1, 1} or {0, 1}

probas_pred : array, shape = [n_samples]

estimated probabilities

Returns **precision** : array, shape = [n]

Precision values

recall : array, shape = [n]

Recall values

thresholds : array, shape = [n]

Thresholds on **proba_** used to compute precision and recall

scikits.learn.metrics.r2_score

scikits.learn.metrics.**r2_score**(*y_true*, *y_pred*)

R^2 (coefficient of determination) regression score function

Best possible score is 1.0, lower values are worse.

Note: not a symmetric function.

return the R^2 score

Parameters **y_true** : array-like

y_pred : array-like

scikits.learn.metrics.zero_one_score

scikits.learn.metrics.**zero_one_score**(*y_true*, *y_pred*)

Zero-One classification score

Positive integer (number of good classifications). The best performance is 1.

Return the percentage of good predictions.

Parameters **y_true** : array-like

y_pred : array-like

Returns **score** : integer

scikits.learn.metrics.zero_one

scikits.learn.metrics.**zero_one**(*y_true*, *y_pred*)

Zero-One classification loss

Positive integer (number of misclassifications). The best performance is 0.

Return the number of errors

Parameters **y_true** : array-like

y_pred : array-like

Returns **loss** : integer

scikits.learn.metrics.mean_square_error

scikits.learn.metrics.**mean_square_error**(*y_true*, *y_pred*)

Mean square error regression loss

Positive floating point value: the best value is 0.0.

return the mean square error

Parameters **y_trye** : array-like

y_pred : array-like

Returns **loss** : float

1.6.10 Covariance Estimators

covariance.Covariance([store_covariance])	Basic covariance estimator
covariance.ShrunkCovariance([...])	Covariance estimator with shrinkage
covariance.LedoitWolf([store_covariance])	LedoitWolf Estimator

scikits.learn.covariance.Covariance

class scikits.learn.covariance.**Covariance**(*store_covariance=True*)
Basic covariance estimator

Parameters **store_covariance** : bool

Specify if the estimated covariance is stored

Attributes

*covariance_*	2D ndarray, shape (n_features, n_features)	Estimated covariance matrix (stored only is store_covariance is True)
*precision_*	2D ndarray, shape (n_features, n_features)	Estimated precision matrix

Methods

fit
log_likelihood
score

__init__(*store_covariance=True*)

scikits.learn.covariance.ShrunkCovariance

class scikits.learn.covariance.**ShrunkCovariance**(*store_covariance=True*, *shrinkage=None*)
Covariance estimator with shrinkage

Parameters **store_covariance** : bool

Specify if the estimated covariance is stored

shrinkage : float

Shrinkage (in [0, 1])

Notes

The regularized covariance is given by

(1 - shrinkage)*cov

- shrinkage*mu*np.identity(n_features)

where mu = trace(cov) / n_features

Attributes

*covariance_*	2D ndarray, shape (n_features, n_features)	Estimated covariance matrix (stored only is store_covariance is True)
*precision_*	2D ndarray, shape (n_features, n_features)	Estimated precision matrix

Methods

fit
log_likelihood
score

__init__(*store_covariance=True*, *shrinkage=None*)

scikits.learn.covariance.LedoitWolf

class scikits.learn.covariance.**LedoitWolf**(*store_covariance=True*)

LedoitWolf Estimator

Parameters **store_covariance** : bool

Specify if the estimated covariance is stored

Notes

The regularised covariance is:

```
(1 - shrinkage)*cov
        + shrinkage*mu*np.identity(n_features)
```

where mu = trace(cov) / n_features

Reference : "A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices", Ledoit and Wolf, Journal of Multivariate Analysis, Volume 88, Issue 2, February 2004, pages 365-411.

Attributes

*covariance_*	2D ndarray, shape (n_features, n_features)	Estimated covariance matrix (stored only is store_covariance is True)
*precision_*	2D ndarray, shape (n_features, n_features)	Estimated precision matrix
*shrinkage_*	float	Scalar used to regularize the precision matrix estimation

Methods

fit
log_likelihood
score

__init__(*store_covariance=True*)

covariance.ledoit_wolf(X[, return_shrinkage])	Estimates the shrunk Ledoit-Wolf covariance matrix.

scikits.learn.covariance.ledoit_wolf

scikits.learn.covariance.**ledoit_wolf**(*X*, *return_shrinkage=False*)

Estimates the shrunk Ledoit-Wolf covariance matrix.

Parameters **X: 2D ndarray, shape (n, p)** :

The data matrix, with p features and n samples.

return_shrinkage: boolean, optional :

If return_shrinkage is True, the regularisation_factor is returned.

Returns **regularised_cov: 2D ndarray** :

Regularized covariance

shrinkage: float :

Regularisation factor

Notes

The regularised covariance is:

```
(1 - shrinkage)*cov
        + shrinkage * mu * np.identity(n_features)
```

where mu = trace(cov) / n_features

1.6.11 Signal Decomposition

pca.PCA([n_components, copy, whiten])	Principal component analysis (PCA)
pca.ProbabilisticPCA([n_components, copy, ...])	
pca.RandomizedPCA(n_components[, copy, ...])	Principal component analysis (PCA) using randomized SVD
fastica.FastICA([n_components, algorithm, ...])	FastICA; a fast algorithm for Independent Component Analysis

scikits.learn.pca.PCA

class scikits.learn.pca.**PCA**(*n_components=None*, *copy=True*, *whiten=False*)

Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data and keeping only the most significant singular vectors to project the data to a lower dimensional space.

This implementation uses the scipy.linalg implementation of the singular value decomposition. It only works for dense arrays and is not scalable to large dimensional data.

The time complexity of this implementation is O(n ** 3) assuming n ~ n_samples ~ n_features.

Parameters **n_components: int, none or string** :

Number of components to keep. if n_components is not set all components are kept:

n_components == min(n_samples, n_features)

if n_components == 'mle', Minka's MLE is used to guess the dimension

if 0 < n_components < 1, select the number of components such that the explained variance ratio is greater than n_components

copy: bool :

If False, data passed to fit are overwritten

whiten: bool, optional :

When True (False by default) the **components_** vectors are divided by n_samples times singular values to ensure uncorrelated outputs with unit component-wise variances.

Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making there data respect some hard-wired assumptions.

See Also:

ProbabilisticPCA, RandomizedPCA

Notes

For n_components='mle', this class uses the method of Thomas P. Minka: Automatic Choice of Dimensionality for PCA. NIPS 2000: 598-604

Examples

```
>>> import numpy as np
>>> from scikits.learn.pca import PCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> pca = PCA(n_components=2)
>>> pca.fit(X)
PCA(copy=True, n_components=2, whiten=False)
>>> print pca.explained_variance_ratio_
[ 0.99244289  0.00755711]
```

Attributes

Methods

fit
inverse_transform
transform

__init__ (*n_components=None*, *copy=True*, *whiten=False*)

fit (*X*, ***params*)
Fit the model from data in X.

Parameters **X: array-like, shape (n_samples, n_features)** :

Training vector, where n_samples in the number of samples and n_features is the number of features.

Returns **self** : object

Returns the instance itself.

inverse_transform (*X*)
Return an input X_original whose transform would be X

Note: if whitening is enabled, inverse_transform does not compute the exact inverse operation as transform.

transform(*X*)
Apply the dimension reduction learned on the train data.

scikits.learn.pca.ProbabilisticPCA

class scikits.learn.pca.**ProbabilisticPCA**(*n_components=None*, *copy=True*, *whiten=False*)

__init__(*n_components=None*, *copy=True*, *whiten=False*)

fit(*X*, *homoscedastic=True*)
Additionally to PCA.fit, learns a covariance model

Parameters **X: array of shape(n_samples, n_dim)** :

The data to fit

homoscedastic: bool, optional, :

If True, average variance across remaining dimensions

inverse_transform(*X*)
Return an input X_original whose transform would be X

Note: if whitening is enabled, inverse_transform does not compute the exact inverse operation as transform.

score(*X*)
Return a score associated to new data

Parameters **X: array of shape(n_samples, n_dim)** :

The data to test

Returns **ll: array of shape (n_samples),** :

log-likelihood of each row of X under the current model

transform(*X*)
Apply the dimension reduction learned on the train data.

scikits.learn.pca.RandomizedPCA

class scikits.learn.pca.**RandomizedPCA**(*n_components*, *copy=True*, *iterated_power=3*, *whiten=False*)
Principal component analysis (PCA) using randomized SVD

Linear dimensionality reduction using approximated Singular Value Decomposition of the data and keeping only the most significant singular vectors to project the data to a lower dimensional space.

This implementation uses a randomized SVD implementation and can handle both scipy.sparse and numpy dense arrays as input.

Parameters **n_components: int** :

Maximum number of components to keep: default is 50.

copy: bool :

If False, data passed to fit are overwritten

iterated_power: int, optional :

Number of iteration for the power method. 3 by default.

whiten: bool, optional :

When True (False by default) the **components_** vectors are divided by the singular values to ensure uncorrelated outputs with unit component-wise variances.

Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making there data respect some hard-wired assumptions.

See Also:

`PCA`, `ProbabilisticPCA`

Notes

References:

•Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 (arXiv:909)

•A randomized algorithm for the decomposition of matrices Per-Gunnar Martinsson, Vladimir Rokhlin and Mark Tygert

Examples

```
>>> import numpy as np
>>> from scikits.learn.pca import RandomizedPCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> pca = RandomizedPCA(n_components=2)
>>> pca.fit(X)
RandomizedPCA(copy=True, n_components=2, iterated_power=3, whiten=False)
>>> print pca.explained_variance_ratio_
[ 0.99244289  0.00755711]
```

Attributes

Methods

fit
inverse_transform
transform

__init__ (*n_components*, *copy=True*, *iterated_power=3*, *whiten=False*)

fit (*X*, ***params*)

Fit the model to the data X.

Parameters **X: array-like or scipy.sparse matrix, shape (n_samples, n_features)** :

Training vector, where n_samples in the number of samples and n_features is the number of features.

Returns **self** : object

Returns the instance itself.

inverse_transform(*X*)
Return an reconstructed input whose transform would be X

transform(*X*)
Apply the dimension reduction learned on the training data.

scikits.learn.fastica.FastICA

class scikits.learn.fastica.**FastICA**(*n_components=None*, *algorithm='parallel'*, *whiten=True*, *fun='logcosh'*, *fun_prime=''*, *fun_args={}*, *max_iter=200*, *tol=0.0001*, *w_init=None*)
FastICA; a fast algorithm for Independent Component Analysis

Parameters **n_components** : int, optional

Number of components to use. If none is passed, all are used.

algorithm: {'parallel', 'deflation'} :

Apply parallel or deflational algorithm for FastICA

whiten: boolean, optional :

If whiten is false, the data is already considered to be whitened, and no whitening is performed.

fun: {'logcosh', 'exp', or 'cube'}, or a callable :

The non-linear function used in the FastICA loop to approximate negentropy. If a function is passed, it derivative should be passed as the 'fun_prime' argument.

fun_prime: None or a callable :

The derivative of the non-linearity used.

max_iter : int, optional

Maximum number of iterations during fit

tol : float, optional

Tolerance on update at each iteration

w_init: None of an (n_components, n_components) ndarray :

The mixing matrix to be used to initialize the algorithm.

Notes

Implementation based on : A. Hyvarinen and E. Oja, Independent Component Analysis: Algorithms and Applications, Neural Networks, 13(4-5), 2000, pp. 411-430

Attributes

unmixing_matrix_	2D array, [n_components, n_samples]	

Methods

get_mixing_matrix() :	Returns an estimate of the mixing matrix

__init__(*n_components=None*, *algorithm='parallel'*, *whiten=True*, *fun='logcosh'*, *fun_prime=''*, *fun_args={}*, *max_iter=200*, *tol=0.0001*, *w_init=None*)

get_mixing_matrix()
Compute the mixing matrix

transform(*X*)
Apply un-mixing matrix "W" to X to recover the sources

S = W * X

fastica.fastica(X[, n_components, ...])	Perform Fast Independent Component Analysis.

scikits.learn.fastica.fastica

scikits.learn.fastica.**fastica**(*X*, *n_components=None*, *algorithm='parallel'*, *whiten=True*, *fun='logcosh'*, *fun_prime=''*, *fun_args={}*, *max_iter=200*, *tol=0.0001*, *w_init=None*)

Perform Fast Independent Component Analysis.

Parameters **X** : (n, p) array of shape = [n_samples, n_features]

Training vector, where n_samples is the number of samples and n_features is the number of features.

n_components : int, optional

Number of components to extract. If None no dimension reduction is performed.

algorithm : {'parallel','deflation'}

Apply an parallel or deflational FASTICA algorithm.

whiten: boolean, optional :

If true perform an initial whitening of the data. Do not set to false unless the data is already white, as you will get incorrect results. If whiten is true, the data is assumed to have already been preprocessed: it should be centered, normed and white.

fun : String or Function

The functional form of the G function used in the approximation to neg-entropy. Could be either 'logcosh', 'exp', or 'cube'. You can also provide your own function but in this case, its derivative should be provided via argument fun_prime

fun_prime : Empty string ('') or Function

See fun.

fun_args : Optional dictionnary

If empty and if fun='logcosh', fun_args will take value {'alpha' : 1.0}

max_iter : int

Maximum number of iterations to perform

tol : float

A positive scalar giving the tolerance at which the un-mixing matrix is considered to have converged

w_init : (n_components,n_components) array

Initial un-mixing array of dimension (n.comp,n.comp). If None (default) then an array of normal r.v.'s is used

source_only: if True, only the sources matrix is returned :

Notes

The data matrix X is considered to be a linear combination of non-Gaussian (independent) components i.e. X = AS where columns of S contain the independent components and A is a linear mixing matrix. In short ICA attempts to 'un-mix' the data by estimating an un-mixing matrix W where S = W K X.

Implemented using FastICA:

A. Hyvarinen and E. Oja, Independent Component Analysis: Algorithms and Applications, Neural Networks, 13(4-5), 2000, pp. 411-430

1.6.12 Cross Validation

`cross_val.LeaveOneOut`(n[, indices])	Leave-One-Out cross validation iterator
`cross_val.LeavePOut`(n, p[, indices])	Leave-P-Out cross validation iterator
`cross_val.KFold`(n, k[, indices])	K-Folds cross validation iterator
`cross_val.StratifiedKFold`(y, k[, indices])	Stratified K-Folds cross validation iterator
`cross_val.LeaveOneLabelOut`(labels[, indices])	Leave-One-Label_Out cross-validation iterator
`cross_val.LeavePLabelOut`(labels, p[, indices])	Leave-P-Label_Out cross-validation iterator

scikits.learn.cross_val.LeaveOneOut

class `scikits.learn.cross_val.`**LeaveOneOut**(*n*, *indices=False*)

Leave-One-Out cross validation iterator

Provides train/test indices to split data in train test sets

__init__(*n*, *indices=False*)

Leave-One-Out cross validation iterator

Provides train/test indices to split data in train test sets

Parameters **n: int** :

Total number of elements

indices: boolean, optional (default False) :

Return train/test split with integer indices or boolean mask. Integer indices are useful when dealing with sparse matrices that cannot be indexed by boolean masks.

Examples

```
>>> from scikits.learn import cross_val
>>> X = np.array([[1, 2], [3, 4]])
>>> y = np.array([1, 2])
>>> loo = cross_val.LeaveOneOut(2)
>>> len(loo)
```

```
2
>>> print loo
scikits.learn.cross_val.LeaveOneOut(n=2)
>>> for train_index, test_index in loo:
...    print "TRAIN:", train_index, "TEST:", test_index
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
...    print X_train, X_test, y_train, y_test
TRAIN: [False  True] TEST: [ True False]
[[3 4]] [[1 2]] [2] [1]
TRAIN: [ True False] TEST: [False  True]
[[1 2]] [[3 4]] [1] [2]
```

scikits.learn.cross_val.LeavePOut

class scikits.learn.cross_val.**LeavePOut**(*n*, *p*, *indices=False*)

Leave-P-Out cross validation iterator

Provides train/test indices to split data in train test sets

__init__(*n*, *p*, *indices=False*)

Leave-P-Out cross validation iterator

Provides train/test indices to split data in train test sets

Parameters **n: int** :

Total number of elements

p: int :

Size test sets

indices: boolean, optional (default False) :

Return train/test split with integer indices or boolean mask. Integer indices are useful when dealing with sparse matrices that cannot be indexed by boolean masks.

Examples

```
>>> from scikits.learn import cross_val
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 3, 4])
>>> lpo = cross_val.LeavePOut(4, 2)
>>> len(lpo)
6
>>> print lpo
scikits.learn.cross_val.LeavePOut(n=4, p=2)
>>> for train_index, test_index in lpo:
...    print "TRAIN:", train_index, "TEST:", test_index
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
TRAIN: [False False  True  True] TEST: [ True  True False False]
TRAIN: [False  True False  True] TEST: [ True False  True False]
TRAIN: [False  True  True False] TEST: [ True False False  True]
TRAIN: [ True False False  True] TEST: [False  True  True False]
TRAIN: [ True False  True False] TEST: [False  True False  True]
TRAIN: [ True  True False False] TEST: [False False  True  True]
```

scikits.learn.cross_val.KFold

class scikits.learn.cross_val.**KFold**(*n*, *k*, *indices=False*)
K-Folds cross validation iterator

Provides train/test indices to split data in train test sets

__init__(*n*, *k*, *indices=False*)
K-Folds cross validation iterator

Provides train/test indices to split data in train test sets

Parameters **n: int** :

Total number of elements

k: int :

number of folds

indices: boolean, optional (default False) :

Return train/test split with integer indices or boolean mask. Integer indices are useful when dealing with sparse matrices that cannot be indexed by boolean masks.

Notes

All the folds have size trunc(n/k), the last one has the complementary

Examples

```
>>> from scikits.learn import cross_val
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4])
>>> kf = cross_val.KFold(4, k=2)
>>> len(kf)
2
>>> print kf
scikits.learn.cross_val.KFold(n=4, k=2)
>>> for train_index, test_index in kf:
...    print "TRAIN:", train_index, "TEST:", test_index
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
TRAIN: [False False  True  True] TEST: [ True  True False False]
TRAIN: [ True  True False False] TEST: [False False  True  True]
```

scikits.learn.cross_val.StratifiedKFold

class scikits.learn.cross_val.**StratifiedKFold**(*y*, *k*, *indices=False*)
Stratified K-Folds cross validation iterator

Provides train/test indices to split data in train test sets

This cross-validation object is a variation of KFold, which returns stratified folds. The folds are made by preserving the percentage of samples for each class.

__init__ (*y*, *k*, *indices=False*)
K-Folds cross validation iterator

Provides train/test indices to split data in train test sets

Parameters **y: array, [n_samples]** :

Samples to split in K folds

k: int :

number of folds

indices: boolean, optional (default False) :

Return train/test split with integer indices or boolean mask. Integer indices are useful when dealing with sparse matrices that cannot be indexed by boolean masks.

Notes

All the folds have size trunc(n/k), the last one has the complementary

Examples

```
>>> from scikits.learn import cross_val
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> skf = cross_val.StratifiedKFold(y, k=2)
>>> len(skf)
2
>>> print skf
scikits.learn.cross_val.StratifiedKFold(labels=[0 0 1 1], k=2)
>>> for train_index, test_index in skf:
...    print "TRAIN:", train_index, "TEST:", test_index
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
TRAIN: [False  True False  True] TEST: [ True False  True False]
TRAIN: [ True False  True False] TEST: [False  True False  True]
```

scikits.learn.cross_val.LeaveOneLabelOut

class scikits.learn.cross_val.**LeaveOneLabelOut** (*labels*, *indices=False*)
Leave-One-Label_Out cross-validation iterator

Provides train/test indices to split data in train test sets

__init__ (*labels*, *indices=False*)
Leave-One-Label_Out cross validation

Provides train/test indices to split data in train test sets

Parameters **labels** : list

List of labels

indices: boolean, optional (default False) :

Return train/test split with integer indices or boolean mask. Integer indices are useful when dealing with sparse matrices that cannot be indexed by boolean masks.

Examples

```
>>> from scikits.learn import cross_val
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 1, 2])
>>> labels = np.array([1, 1, 2, 2])
>>> lol = cross_val.LeaveOneLabelOut(labels)
>>> len(lol)
2
>>> print lol
scikits.learn.cross_val.LeaveOneLabelOut(labels=[1 1 2 2])
>>> for train_index, test_index in lol:
...    print "TRAIN:", train_index, "TEST:", test_index
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
...    print X_train, X_test, y_train, y_test
TRAIN: [False False  True  True] TEST: [ True  True False False]
[[5 6]
 [7 8]] [[1 2]
 [3 4]] [1 2] [1 2]
TRAIN: [ True  True False False] TEST: [False False  True  True]
[[1 2]
 [3 4]] [[5 6]
 [7 8]] [1 2] [1 2]
```

scikits.learn.cross_val.LeavePLabelOut

class scikits.learn.cross_val.**LeavePLabelOut**(*labels*, *p*, *indices=False*)

Leave-P-Label_Out cross-validation iterator

Provides train/test indices to split data in train test sets

__init__(*labels*, *p*, *indices=False*)

Leave-P-Label_Out cross validation

Provides train/test indices to split data in train test sets

Parameters **labels** : list

List of labels

indices: boolean, optional (default False) :

Return train/test split with integer indices or boolean mask. Integer indices are useful when dealing with sparse matrices that cannot be indexed by boolean masks.

Examples

```
>>> from scikits.learn import cross_val
>>> X = np.array([[1, 2], [3, 4], [5, 6]])
>>> y = np.array([1, 2, 1])
>>> labels = np.array([1, 2, 3])
>>> lpl = cross_val.LeavePLabelOut(labels, p=2)
>>> len(lpl)
3
>>> print lpl
scikits.learn.cross_val.LeavePLabelOut(labels=[1 2 3], p=2)
```

```
>>> for train_index, test_index in lpl:
...     print "TRAIN:", train_index, "TEST:", test_index
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print X_train, X_test, y_train, y_test
TRAIN: [False False  True] TEST: [ True  True False]
[[5 6]] [[1 2]
 [3 4]] [1] [1 2]
TRAIN: [False  True False] TEST: [ True False  True]
[[3 4]] [[1 2]
 [5 6]] [2] [1 1]
TRAIN: [ True False False] TEST: [False  True  True]
[[1 2]] [[3 4]
 [5 6]] [1] [2 1]
```

1.6.13 Grid Search

grid_search.GridSearchCV(estimator, param_grid)	Grid search on the parameters of a classifier

scikits.learn.grid_search.GridSearchCV

class scikits.learn.grid_search.**GridSearchCV**(*estimator*, *param_grid*, *loss_func=None*, *score_func=None*, *fit_params={}*, *n_jobs=1*, *iid=True*, *refit=True*, *cv=None*)

Grid search on the parameters of a classifier

Important members are fit, predict.

GridSearchCV implements a "fit" method and a "predict" method like any classifier except that the parameters of the classifier used to predict is optimized by cross-validation

Parameters **estimator: object type that implements the "fit" and "predict" methods** :

A object of that type is instanciated for each grid point

param_grid: dict :

a dictionary of parameters that are used the generate the grid

loss_func: callable, optional :

function that takes 2 arguments and compares them in order to evaluate the performance of prediciton (small is good) if None is passed, the score of the estimator is maximized

score_func: callable, optional :

function that takes 2 arguments and compares them in order to evaluate the performance of prediciton (big is good) if None is passed, the score of the estimator is maximized

fit_params : dict, optional

parameters to pass to the fit method

n_jobs: int, optional :

number of jobs to run in parallel (default 1)

iid: boolean, optional :

If True, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample, and not the mean loss across the folds.

cv : crossvalidation generator

see scikits.learn.cross_val module

refit: boolean :

refit the best estimator with the entire dataset

Notes

The parameters selected are those that maximize the score of the left out data, unless an explicit score_func is passed in which case it is used instead. If a loss function loss_func is passed, it overrides the score functions and is minimized.

Examples

```
>>> from scikits.learn import svm, grid_search, datasets
>>> iris = datasets.load_iris()
>>> parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
>>> svr = svm.SVR()
>>> clf = grid_search.GridSearchCV(svr, parameters)
>>> clf.fit(iris.data, iris.target)
GridSearchCV(n_jobs=1, fit_params={}, loss_func=None, refit=True, cv=None,
       iid=True,
       estimator=SVR(kernel='rbf', C=1.0, probability=False, ...
       ...
```

Methods

fit
score

__init__ (*estimator*, *param_grid*, *loss_func=None*, *score_func=None*, *fit_params={}*, *n_jobs=1*, *iid=True*, *refit=True*, *cv=None*)

fit (*X*, *y=None*, ***params*)

Run fit with all sets of parameters

Returns the best classifier

Parameters **X: array, [n_samples, n_features]** :

Training vector, where n_samples in the number of samples and n_features is the number of features.

y: array, [n_samples] or None :

Target vector relative to X, None for unsupervised problems

1.6.14 Feature Selection

feature_selection.rfe.RFE([estimator, ...])	Feature ranking with Recursive feature elimination
feature_selection.rfe.RFECV([estimator, ...])	Feature ranking with Recursive feature elimination and cross validation

scikits.learn.feature_selection.rfe.RFE

class scikits.learn.feature_selection.rfe.**RFE**(*estimator=None*, *n_features=None*, *percentage=0.10000000000000001*)

Feature ranking with Recursive feature elimination

Parameters **estimator** : object

A supervised learning estimator with a fit method that updates a **coef_** attributes that holds the fitted parameters. The first dimension of the **coef_** array must be equal n_features an important features must yield high absolute values in the **coef_** array.

For instance this is the case for most supervised learning algorithms such as Support Vector Classifiers and Generalized Linear Models from the svm and linear_model package.

n_features : int

Number of features to select

percentage : float

The percentage of features to remove at each iteration Should be between (0, 1]. By default 0.1 will be taken.

References

Guyon, I., Weston, J., Barnhill, S., & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. Mach. Learn., 46(1-3), 389–422.

Examples

```
>>> # TODO!
```

Attributes

*support_*	array-like, shape = [n_features]	Mask of estimated support
*ranking_*	array-like, shape = [n_features]	Mask of the ranking of features

Methods

fit(X, y)	self	Fit the model
transform(X)	array	Reduce X to support

__init__(*estimator=None*, *n_features=None*, *percentage=0.10000000000000001*)

fit(*X*, *y*)

Fit the RFE model

Parameters **X** : array-like, shape = [n_samples, n_features]

Training vector, where n_samples in the number of samples and n_features is the number of features.

y : array, shape = [n_samples]

Target values (integers in classification, real numbers in regression)

transform(*X*, *copy=True*)

Reduce X to the features selected during the fit

Parameters **X** : array-like, shape = [n_samples, n_features]

Vector, where n_samples in the number of samples and n_features is the number of features.

scikits.learn.feature_selection.rfe.RFECV

class scikits.learn.feature_selection.rfe.**RFECV**(*estimator=None*, *n_features=None*, *percentage=0.10000000000000001*, *loss_func=None*)

Feature ranking with Recursive feature elimination and cross validation

Parameters **estimator** : object

A supervised learning estimator with a fit method that updates a **coef_** attributes that holds the fitted parameters. The first dimension of the **coef_** array must be equal n_features an important features must yield high absolute values in the **coef_** array.

For instance this is the case for most supervised learning algorithms such as Support Vector Classifiers and Generalized Linear Models from the svm and linear_model package.

n_features : int

Number of features to select

percentage : float

The percentage of features to remove at each iteration Should be between (0, 1]. By default 0.1 will be taken.

References

Guyon, I., Weston, J., Barnhill, S., & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. Mach. Learn., 46(1-3), 389–422.

Examples

```
>>> # TODO!
```

Attributes

*support_*	array-like, shape = [n_features]	Mask of estimated support
*ranking_*	array-like, shape = [n_features]	Mask of the ranking of features

Methods

fit(X, y)	self	Fit the model
transform(X)	array	Reduce X to support

__init__ (*estimator=None*, *n_features=None*, *percentage=0.10000000000000001*, *loss_func=None*)

fit (*X*, *y*, *cv=None*)

Fit the RFE model with cross-validation

The final size of the support is tuned by cross validation.

Parameters **X** : array-like, shape = [n_samples, n_features]

Training vector, where n_samples in the number of samples and n_features is the number of features.

y : array, shape = [n_samples]

Target values (integers in classification, real numbers in regression)

cv : cross-validation instance

transform (*X*, *copy=True*)

Reduce X to the features selected during the fit

Parameters **X** : array-like, shape = [n_samples, n_features]

Vector, where n_samples in the number of samples and n_features is the number of features.

1.6.15 Feature Extraction

feature_extraction.image.img_to_graph(img[, ...])	Graph of the pixel-to-pixel gradient connections

scikits.learn.feature_extraction.image.img_to_graph

scikits.learn.feature_extraction.image.**img_to_graph** (*img*, *mask=None*, *return_as=<class 'scipy.sparse.coo.coo_matrix'>*, *dtype=None*)

Graph of the pixel-to-pixel gradient connections

Edges are weighted with the gradient values.

Parameters **img: ndarray, 2D or 3D** :

2D or 3D image

mask : ndarray of booleans, optional

An optional mask of the image, to consider only part of the pixels.

return_as: np.ndarray or a sparse matrix class, optional :

The class to use to build the returned adjacency matrix.

dtype: None or dtype, optional :

The data of the returned sparse matrix. By default it is the dtype of img

feature_extraction.text.RomanPreprocessor()	Fast preprocessor suitable for roman languages ..
feature_extraction.text.WordNGramAnalyzer([...])	Simple analyzer: transform a text document into a sequence of word tokens
feature_extraction.text.CharNGramAnalyzer([...])	Compute character n-grams features of a text document
feature_extraction.text.CountVectorizer([...])	Convert a collection of raw documents to a matrix of token counts
feature_extraction.text.TfidfTransformer([...])	Transform a count matrix to a TF or TF-IDF representation
feature_extraction.text.Vectorizer([...])	Convert a collection of raw documents to a matrix

scikits.learn.feature_extraction.text.RomanPreprocessor

class scikits.learn.feature_extraction.text.**RomanPreprocessor**

Fast preprocessor suitable for roman languages

Methods

preprocess

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

scikits.learn.feature_extraction.text.WordNGramAnalyzer

class scikits.learn.feature_extraction.text.**WordNGramAnalyzer**(*charset='utf-8'*, *min_n=1*, *max_n=1*, *preprocessor=RomanPreprocessor()*, *stop_words=set([*'all', 'six', 'less', 'being', 'indeed', 'over', 'move', 'anyway', 'four', 'not', 'own', 'through', 'yourselves', 'fify', 'where', 'mill', 'only', 'find', 'before', 'one', 'whose', 'system', 'how', 'somewhere', 'with', 'thick', 'show', 'had', 'enough', 'should', 'to', 'must', 'whom', 'seeming', 'under', 'ours', 'two', 'has', 'might', 'thereafter', 'latterly', 'do', 'them', 'his', 'around', 'than', 'get', 'very', 'de', 'none', 'cannot', 'every', 'whether', 'they', 'front', 'during', 'thus', 'now', 'him', 'nor', 'name', 'several', 'hereafter', 'always', 'who', 'cry', 'whither', 'this', 'someone', 'either', 'each', 'become', 'thereupon', 'sometime', 'side', 'towards', 'therein', 'twelve', 'because', 'often', 'ten', 'our', 'eg', 'some', 'back', 'up', 'go', 'namely', 'computer', 'are', 'further', 'beyond', 'ourselves', 'yet', 'out', 'even', 'will', 'what', 'still', 'for', 'bottom', 'mine', 'since', 'please', 'forty', 'per', 'its', 'everything', 'behind', 'un', 'above', 'between', 'it', 'neither', 'seemed', 'ever', 'across', 'she',

This simple implementation does:

- lower case conversion
- unicode accents removal
- token extraction using unicode regexp word bounderies for token of minimum size of 2 symbols (by default)
- output token n-grams (unigram only by default)

Methods

`analyze`

__init__(*charset='utf-8'*, *min_n=1*, *max_n=1*, *preprocessor=RomanPreprocessor()*, *stop_words=set([*, *'all'*, *'six'*, *'less'*, *'being'*, *'indeed'*, *'over'*, *'move'*, *'anyway'*, *'four'*, *'not'*, *'own'*, *'through'*, *'yourselves'*, *'fifty'*, *'where'*, *'mill'*, *'only'*, *'find'*, *'before'*, *'one'*, *'whose'*, *'system'*, *'how'*, *'somewhere'*, *'with'*, *'thick'*, *'show'*, *'had'*, *'enough'*, *'should'*, *'to'*, *'must'*, *'whom'*, *'seeming'*, *'under'*, *'ours'*, *'two'*, *'has'*, *'might'*, *'thereafter'*, *'latterly'*, *'do'*, *'them'*, *'his'*, *'around'*, *'than'*, *'get'*, *'very'*, *'de'*, *'none'*, *'cannot'*, *'every'*, *'whether'*, *'they'*, *'front'*, *'during'*, *'thus'*, *'now'*, *'him'*, *'nor'*, *'name'*, *'several'*, *'hereafter'*, *'always'*, *'who'*, *'cry'*, *'whither'*, *'this'*, *'someone'*, *'either'*, *'each'*, *'become'*, *'thereupon'*, *'sometime'*, *'side'*, *'towards'*, *'therein'*, *'twelve'*, *'because'*, *'often'*, *'ten'*, *'our'*, *'eg'*, *'some'*, *'back'*, *'up'*, *'go'*, *'namely'*, *'computer'*, *'are'*, *'further'*, *'beyond'*, *'ourselves'*, *'yet'*, *'out'*, *'even'*, *'will'*, *'what'*, *'still'*, *'for'*, *'bottom'*, *'mine'*, *'since'*, *'please'*, *'forty'*, *'per'*, *'its'*, *'everything'*, *'behind'*, *'un'*, *'above'*, *'between'*, *'it'*, *'neither'*, *'seemed'*, *'ever'*, *'across'*, *'she'*, *'somehow'*, *'be'*, *'we'*, *'full'*, *'never'*, *'sixty'*, *'however'*, *'here'*, *'otherwise'*, *'were'*, *'whereupon'*, *'nowhere'*, *'although'*, *'found'*, *'alone'*, *'re'*, *'along'*, *'fifteen'*, *'by'*, *'both'*, *'about'*, *'last'*, *'would'*, *'anything'*, *'via'*, *'many'*, *'could'*, *'thence'*, *'put'*, *'against'*, *'keep'*, *'etc'*, *'amount'*, *'became'*, *'ltd'*, *'hence'*, *'onto'*, *'or'*, *'con'*, *'among'*, *'already'*, *'co'*, *'afterwards'*, *'formerly'*, *'within'*, *'seems'*, *'into'*, *'others'*, *'while'*, *'whatever'*, *'except'*, *'down'*, *'hers'*, *'everyone'*, *'done'*, *'least'*, *'another'*, *'whoever'*, *'moreover'*, *'couldnt'*, *'throughout'*, *'anyhow'*, *'yourself'*, *'three'*, *'from'*, *'her'*, *'few'*, *'together'*, *'top'*, *'there'*, *'due'*, *'been'*, *'next'*, *'anyone'*, *'eleven'*, *'much'*, *'call'*, *'therefore'*, *'interest'*, *'then'*, *'thru'*, *'themselves'*, *'hundred'*, *'was'*, *'sincere'*, *'empty'*, *'more'*, *'himself'*, *'elsewhere'*, *'mostly'*, *'on'*, *'fire'*, *'am'*, *'becoming'*, *'hereby'*, *'amongst'*, *'else'*, *'part'*, *'everywhere'*, *'too'*, *'herself'*, *'former'*, *'those'*, *'he'*, *'me'*, *'myself'*, *'made'*, *'twenty'*, *'these'*, *'bill'*, *'cant'*, *'us'*, *'until'*, *'besides'*, *'nevertheless'*, *'below'*, *'anywhere'*, *'nine'*, *'can'*, *'of'*, *'your'*, *'toward'*, *'my'*, *'something'*, *'and'*, *'whereafter'*, *'whenever'*, *'give'*, *'almost'*, *'wherever'*, *'is'*, *'describe'*, *'beforehand'*, *'herein'*, *'an'*, *'as'*, *'itself'*, *'at'*, *'have'*, *'in'*, *'seem'*, *'whence'*, *'ie'*, *'any'*, *'fill'*, *'again'*, *'hasnt'*, *'inc'*, *'thereby'*, *'thin'*, *'no'*, *'perhaps'*, *'latter'*, *'meanwhile'*, *'when'*, *'detail'*, *'same'*, *'wherein'*, *'beside'*, *'also'*, *'that'*, *'other'*, *'take'*, *'which'*, *'becomes'*, *'you'*, *'if'*, *'nobody'*, *'see'*, *'though'*, *'may'*, *'after'*, *'upon'*, *'most'*, *'hereupon'*, *'eight'*, *'but'*, *'serious'*, *'nothing'*, *'such'*, *'why'*, *'a'*, *'off'*, *'whereby'*, *'third'*, *'i'*, *'whole'*, *'noone'*, *'sometimes'*, *'well'*, *'amoungst'*, *'yours'*, *'their'*, *'rather'*, *'without'*, *'so'*, *'five'*, *'the'*, *'first'*, *'whereas'*, *'once'* *])*, *token_pattern='\b\w\w+\b'*)

scikits.learn.feature_extraction.text.CharNGramAnalyzer

class scikits.learn.feature_extraction.text.**CharNGramAnalyzer**(*charset='utf-8'*, *preprocessor=RomanPreprocessor()*, *min_n=3*, *max_n=6*)

Compute character n-grams features of a text document

This analyzer is interesting since it is language agnostic and will work well even for language where word segmentation is not as trivial as English such as Chinese and German for instance.

Because of this, it can be considered a basic morphological analyzer.

Methods

analyze

__init__(*charset='utf-8'*, *preprocessor=RomanPreprocessor()*, *min_n=3*, *max_n=6*)

scikits.learn.feature_extraction.text.CountVectorizer

CountVectorizer(analyzer=WordNGramAnalyzer(stop_words=set(['all', 'six', 'less', 'being',
max_n=1, token_pattern='\b\w\w+\b', charset='utf-8', min_n=1,
preprocessor=RomanPreprocessor()), vocabulary={}, max_df=1.0, max_features=None, dtype=<ty

Convert a collection of raw documents to a matrix of token counts

This implementation produces a dense representation of the counts using a numpy array.

If you do not provide an a-priori dictionary and you do not use an analyzer that does some kind of feature selection then the number of features (the vocabulary size found by analysing the data) might be very large and the count vectors might not fit in memory.

For this case it is either recommended to use the sparse.CountVectorizer variant of this class or a HashingVectorizer that will reduce the dimensionality to an arbitrary number by using random projection.

Parameters **analyzer: WordNGramAnalyzer or CharNGramAnalyzer, optional** :

vocabulary: dict, optional :

A dictionary where keys are tokens and values are indices in the matrix. This is useful in order to fix the vocabulary in advance.

dtype: type, optional :

Type of the matrix returned by fit_transform() or transform().

Methods

fit
fit_transform
transform

CountVectorizer.**fit**(*raw_documents*, *y=None*)

Learn a vocabulary dictionary of all tokens in the raw documents

Parameters **raw_documents: iterable** :

an iterable which yields either str, unicode or file objects

Returns **self** :

CountVectorizer.**fit_transform**(*raw_documents*, *y=None*)

Learn the vocabulary dictionary and return the count vectors

This is more efficient than calling fit followed by transform.

Parameters **raw_documents: iterable** :

an iterable which yields either str, unicode or file objects

Returns **vectors: array, [n_samples, n_features]** :

`CountVectorizer.`**`transform`**(*raw_documents*)

Extract token counts out of raw text documents

Parameters **raw_documents: iterable** :

an iterable which yields either str, unicode or file objects

Returns **vectors: array, [n_samples, n_features]** :

scikits.learn.feature_extraction.text.TfidfTransformer

class `scikits.learn.feature_extraction.text.`**`TfidfTransformer`**(*use_tf=True*, *use_idf=True*)

Transform a count matrix to a TF or TF-IDF representation

TF means term-frequency while TF-IDF means term-frequency times inverse document-frequency:

http://en.wikipedia.org/wiki/TF-IDF

The goal of using TF-IDF instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than feature that occur in a small fraction of the training corpus.

TF-IDF can be seen as a smooth alternative to the stop words filtering.

Parameters **use_tf: boolean** :

enable term-frequency normalization

use_idf: boolean :

enable inverse-document-frequency reweighting

Methods

`fit`
`transform`

`__init__`(*use_tf=True*, *use_idf=True*)

`fit`(*X*, *y=None*)

Learn the IDF vector (global term weights)

Parameters **X: array, [n_samples, n_features]** :

a matrix of term/token counts

`transform`(*X*, *copy=True*)

Transform a count matrix to a TF or TF-IDF representation

Parameters **X: array, [n_samples, n_features]** :

a matrix of term/token counts

Returns **vectors: array, [n_samples, n_features]** :

scikits.learn.feature_extraction.text.Vectorizer

Vectorizer(analyzer=WordNGramAnalyzer(stop_words=set(['all', 'six', 'less', 'being', 'indee
max_n=1, token_pattern='\b\w\w+\b', charset='utf-8', min_n=1,
preprocessor=RomanPreprocessor()), max_df=1.0, max_features=None, use_tf=True, use_idf=True

Convert a collection of raw documents to a matrix

Equivalent to CountVectorizer followed by TfidfTransformer.

Methods

fit
fit_transform
transform

Vectorizer.**fit_transform**(*raw_documents*)

Learn the representation and return the vectors.

Parameters **raw_documents: iterable** :

an iterable which yields either str, unicode or file objects

Returns **vectors: array, [n_samples, n_features]** :

Vectorizer.**transform**(*raw_documents*, *copy=True*)

Return the vectors.

Parameters **raw_documents: iterable** :

an iterable which yields either str, unicode or file objects

Returns **vectors: array, [n_samples, n_features]** :

For sparse data

feature_extraction.text.sparse.TfidfTransformer([...])	
feature_extraction.text.sparse.CountVectorizer([...])	Convert a collection of raw documents to a matrix of token counts
feature_extraction.text.sparse.Vectorizer([...])	Convert a collection of raw documents to a sparse matrix

scikits.learn.feature_extraction.text.sparse.TfidfTransformer

class scikits.learn.feature_extraction.text.sparse.**TfidfTransformer**(*use_tf=True*, *use_idf=True*)

__init__(*use_tf=True*, *use_idf=True*)

fit(*X*, *y=None*)

Learn the IDF vector (global term weights)

Parameters **X: sparse matrix, [n_samples, n_features]** :

a matrix of term/token counts

transform(*X*, *copy=True*)

Transform a count matrix to a TF or TF-IDF representation

Parameters **X: sparse matrix, [n_samples, n_features]** :

a matrix of term/token counts

Returns **vectors: sparse matrix, [n_samples, n_features]** :

scikits.learn.feature_extraction.text.sparse.CountVectorizer

```
CountVectorizer(analyzer=WordNGramAnalyzer(stop_words=set(['all', 'six', 'less', 'being', '
max_n=1, token_pattern='\b\w\w+\b', charset='utf-8', min_n=1,
preprocessor=RomanPreprocessor()), vocabulary={}, max_df=1.0, max_features=None, dtype=<typ
```

Convert a collection of raw documents to a matrix of token counts

This implementation produces a sparse representation of the counts using scipy.sparse.coo_matrix.

Parameters **analyzer: WordNGramAnalyzer or CharNGramAnalyzer, optional** :

vocabulary: dict, optional :

A dictionary where keys are tokens and values are indices in the matrix. This is useful in order to fix the vocabulary in advance.

dtype: type, optional :

Type of the matrix returned by fit_transform() or transform().

Methods

fit
fit_transform
transform

CountVectorizer.**fit**(*raw_documents*, *y=None*)

Learn a vocabulary dictionary of all tokens in the raw documents

Parameters **raw_documents: iterable** :

an iterable which yields either str, unicode or file objects

Returns **self** :

CountVectorizer.**fit_transform**(*raw_documents*, *y=None*)

Learn the vocabulary dictionary and return the count vectors

This is more efficient than calling fit followed by transform.

Parameters **raw_documents: iterable** :

an iterable which yields either str, unicode or file objects

Returns **vectors: array, [n_samples, n_features]** :

CountVectorizer.**transform**(*raw_documents*)

Extract token counts out of raw text documents

Parameters **raw_documents: iterable** :

an iterable which yields either str, unicode or file objects

Returns **vectors: array, [n_samples, n_features]** :

scikits.learn.feature_extraction.text.sparse.Vectorizer

Vectorizer(analyzer=WordNGramAnalyzer(stop_words=set(['all', 'six', 'less', 'being', 'indee
max_n=1, token_pattern='\b\w\w+\b', charset='utf-8', min_n=1,
preprocessor=RomanPreprocessor()), max_df=1.0, max_features=None, use_tf=True, use_idf=True

Convert a collection of raw documents to a sparse matrix

Equivalent to CountVectorizer followed by TfidfTransformer.

Methods

fit
fit_transform
transform

Vectorizer.**fit_transform**(*raw_documents*)

Learn the representation and return the vectors.

Parameters **raw_documents: iterable** :

an iterable which yields either str, unicode or file objects

Returns **vectors: array, [n_samples, n_features]** :

Vectorizer.**transform**(*raw_documents*, *copy=True*)

Return the vectors.

Parameters **raw_documents: iterable** :

an iterable which yields either str, unicode or file objects

Returns **vectors: array, [n_samples, n_features]** :

1.6.16 Pipeline

pipeline.Pipeline(steps)	Pipeline of transforms with a final estimator

scikits.learn.pipeline.Pipeline

class scikits.learn.pipeline.**Pipeline**(*steps*)

Pipeline of transforms with a final estimator

Sequentialy apply a list of transforms and a final estimator Intermediate steps of the pipeline must be 'transforms', that is that they must implements fit & transform methods The final estimator need only implements fit.

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables to setting parameters of the various steps using their names and the parameter name separated by a '__', as in the example below.

Attributes

Methods

fit:	Fit all the transforms one after the other and transform the data, then fit the transformed data using the final estimator
fit_transform:	Fit all the transforms one after the other and transform the data, then use fit_transform on transformed data using the final estimator. Valid only if the final estimator implements fit_transform.
predict:	Applies transforms to the data, and the predict method of the final estimator. Valid only if the final estimator implements predict.
transform:	Applies transforms to the data, and the transform method of the final estimator. Valid only if the final estimator implements transform.
score:	Applies transforms to the data, and the score method of the final estimator. Valid only if the final estimator implements score.

__init__ (*steps*)

Parameters **steps: list** :

List of (name, transform) object (implementing fit/transform) that are chained, in the order in which they are chained, with the last object an estimator.

CHAPTER

TWO

EXAMPLE GALLERY

2.1 Examples

2.1.1 General examples

General-purpose and introductory examples for the scikit.

Classification of text documents using sparse features

This is an example showing how the scikit-learn can be used to classify documents by topics using a bag-of-words approach. This example uses a scipy.sparse matrix to store the features instead of standard numpy arrays.

The dataset used in this example is the 20 newsgroups dataset which will be automatically downloaded and then cached.

You can adjust the number of categories by giving there name to the dataset loader or setting them to None to get the 20 of them.

This example demos various linear classifiers with different training strategies.

To run this example use:

```
% python examples/document_classification_20newsgroups.py [options]
```

Options are:

--report	Print a detailed classification report.
--confusion-matrix	Print the confusion matrix.

Python source code: `document_classification_20newsgroups.py`

```
print __doc__

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#         Olivier Grisel <olivier.grisel@ensta.org>
#         Mathieu Blondel <mathieu@mblondel.org>
# License: Simplified BSD

from time import time
import os
import sys

from scikits.learn.datasets import load_files
```

```
from scikits.learn.feature_extraction.text.sparse import Vectorizer
from scikits.learn.linear_model import RidgeClassifier
from scikits.learn.svm.sparse import LinearSVC
from scikits.learn.linear_model.sparse import SGDClassifier
from scikits.learn import metrics

# parse commandline arguments
argv = sys.argv[1:]
if "--report" in argv:
    print_report = True
else:
    print_report = False
if "--confusion-matrix" in argv:
    print_cm = True
else:
    print_cm = False

###############################################################################
# Download the data, if not already on disk
url = "http://people.csail.mit.edu/jrennie/20Newsgroups/20news-18828.tar.gz"
archive_name = "20news-18828.tar.gz"

if not os.path.exists(archive_name[:-7]):
    if not os.path.exists(archive_name):
        import urllib
        print "Downloading data, please Wait (14MB)..."
        print url
        opener = urllib.urlopen(url)
        open(archive_name, 'wb').write(opener.read())
        print

    import tarfile
    print "Decompressiong the archive: " + archive_name
    tarfile.open(archive_name, "r:gz").extractall()
    print


###############################################################################
# Load some categories from the training set
categories = [
    'alt.atheism',
    'talk.religion.misc',
    'comp.graphics',
    'sci.space',
]
# Uncomment the following to do the analysis on all the categories
#categories = None

print "Loading 20 newsgroups dataset for categories:"
print categories

data = load_files('20news-18828', categories=categories, shuffle=True, rng=42)
print "%d documents" % len(data.filenames)
print "%d categories" % len(data.target_names)
print

# split a training set and a test set
filenames = data.filenames
```

```
y = data.target

n = filenames.shape[0]
filenames_train, filenames_test = filenames[:-n/2], filenames[-n/2:]
y_train, y_test = y[:-n/2], y[-n/2:]

print "Extracting features from the training dataset using a sparse vectorizer"
t0 = time()
vectorizer = Vectorizer()
X_train = vectorizer.fit_transform((open(f).read() for f in filenames_train))
print "done in %fs" % (time() - t0)
print "n_samples: %d, n_features: %d" % X_train.shape
print

print "Extracting features from the test dataset using the same vectorizer"
t0 = time()
X_test = vectorizer.transform((open(f).read() for f in filenames_test))
print "done in %fs" % (time() - t0)
print "n_samples: %d, n_features: %d" % X_test.shape
print


###############################################################################
# Benchmark classifiers
def benchmark(clf):
    print 80 * '_'
    print "Training: "
    print clf
    t0 = time()
    clf.fit(X_train, y_train)
    train_time = time() - t0
    print "train time: %0.3fs" % train_time

    t0 = time()
    pred = clf.predict(X_test)
    test_time = time() - t0
    print "test time:  %0.3fs" % test_time

    score = metrics.f1_score(y_test, pred)
    print "f1-score:   %0.3f" % score

    nnz = clf.coef_.nonzero()[0].shape[0]
    print "non-zero coef: %d" % nnz
    print

    if print_report:
        print "classification report:"
        print metrics.classification_report(y_test, pred,
                                            class_names=categories)

    if print_cm:
        print "confusion matrix:"
        print metrics.confusion_matrix(y_test, pred)

    print
    return score, train_time, test_time

for clf, name in ((RidgeClassifier(), "Ridge Classifier"),):
```

```
    print 80*'='
    print name
    results = benchmark(clf)

for penalty in ["l2", "l1"]:
    print 80*'='
    print "%s penalty" % penalty.upper()
    # Train Liblinear model
    liblinear_results = benchmark(LinearSVC(loss='l2', penalty=penalty, C=1000,
                                            dual=False, eps=1e-3))

    # Train SGD model
    sgd_results = benchmark(SGDClassifier(alpha=.0001, n_iter=50,
                                          penalty=penalty))

# Train SGD with Elastic Net penalty
print 80*'='
print "Elastic-Net penalty"
sgd_results = benchmark(SGDClassifier(alpha=.0001, n_iter=50,
                                      penalty="elasticnet"))
```

Pipeline Anova SVM

Simple usage of Pipeline that runs successively a univariate feature selection with anova and then a C-SVM of the selected features.

Python source code: `feature_selection_pipeline.py`

```
print __doc__

from scikits.learn import svm
from scikits.learn.datasets import samples_generator
from scikits.learn.feature_selection import SelectKBest, f_regression
from scikits.learn.pipeline import Pipeline

# import some data to play with
X, y = samples_generator.test_dataset_classif(k=5)

# ANOVA SVM-C
# 1) anova filter, take 5 best ranked features
anova_filter = SelectKBest(f_regression, k=5)
# 2) svm
clf = svm.SVC(kernel='linear')

anova_svm = Pipeline([('anova', anova_filter), ('svm', clf)])
anova_svm.fit(X, y)
anova_svm.predict(X)
```

Parameter estimation using grid search with a nested cross-validation

The classifier is optimized by "nested" cross-validation using the GridSearchCV object.

The performance of the selected parameters is evaluated using cross-validation (different than the nested cross-validation that is used to select the best classifier).

Python source code: `grid_search_digits.py`

```
print __doc__

from pprint import pprint
import numpy as np

from scikits.learn import datasets
from scikits.learn.cross_val import StratifiedKFold
from scikits.learn.grid_search import GridSearchCV
from scikits.learn.metrics import classification_report
from scikits.learn.metrics import precision_score
from scikits.learn.metrics import recall_score
from scikits.learn.svm import SVC

###############################################################################
# Loading the Digits dataset
digits = datasets.load_digits()

# To apply an classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
X = digits.images.reshape((n_samples, -1))
y = digits.target

# split the dataset in two equal part respecting label proportions
train, test = iter(StratifiedKFold(y, 2)).next()

###############################################################################
# Set the parameters by cross-validation
tuned_parameters = [{'kernel': ['rbf'], 'gamma': [1e-3, 1e-4],
                     'C': [1, 10, 100, 1000]},
                    {'kernel': ['linear'], 'C': [1, 10, 100, 1000]}]

scores = [
    ('precision', precision_score),
    ('recall', recall_score),
]

for score_name, score_func in scores:
    clf = GridSearchCV(SVC(C=1), tuned_parameters, n_jobs=2,
                       score_func=score_func)
    clf.fit(X[train], y[train], cv=StratifiedKFold(y[train], 5))
    y_true, y_pred = y[test], clf.predict(X[test])

    print "Classification report for the best estimator: "
    print clf.best_estimator
    print "Tuned for '%s' with optimal value: %0.3f" % (
        score_name, score_func(y_true, y_pred))
    print classification_report(y_true, y_pred)
    print "Grid scores:"
    pprint(clf.grid_points_scores_)
    print

# Note the problem is too easy: the hyperparameter plateau is too flat and the
# output model is the same for precision and recall with ties in quality
```

Sample pipeline for text feature extraction and evaluation

The dataset used in this example is the 20 newsgroups dataset which will be automatically downloaded and then cached and reused for the document classification example.

You can adjust the number of categories by giving there name to the dataset loader or setting them to None to get the 20 of them.

Here is a sample output of a run on a quad-core machine:

```
Loading 20 newsgroups dataset for categories:
['alt.atheism', 'talk.religion.misc']
1427 documents
2 categories

Performing grid search...
pipeline: ['vect', 'tfidf', 'clf']
parameters:
{'clf__alpha': (1.0000000000000001e-05, 9.9999999999999995e-07),
 'clf__n_iter': (10, 50, 80),
 'clf__penalty': ('l2', 'elasticnet'),
 'tfidf__use_idf': (True, False),
 'vect__analyzer__max_n': (1, 2),
 'vect__max_df': (0.5, 0.75, 1.0),
 'vect__max_features': (None, 5000, 10000, 50000)}
done in 1737.030s

Best score: 0.940
Best parameters set:
    clf__alpha: 9.9999999999999995e-07
    clf__n_iter: 50
    clf__penalty: 'elasticnet'
    tfidf__use_idf: True
    vect__analyzer__max_n: 2
    vect__max_df: 0.75
    vect__max_features: 50000
```

Python source code: `grid_search_text_feature_extraction.py`

```
print __doc__

# Author: Olivier Grisel <olivier.grisel@ensta.org>
#         Peter Prettenhofer <peter.prettenhofer@gmail.com>
#         Mathieu Blondel <mathieu@mblondel.org>
# License: Simplified BSD

from pprint import pprint
from time import time
import os

from scikits.learn.datasets import load_files
from scikits.learn.feature_extraction.text.sparse import CountVectorizer
from scikits.learn.feature_extraction.text.sparse import TfidfTransformer
from scikits.learn.linear_model.sparse import SGDClassifier
from scikits.learn.grid_search import GridSearchCV
from scikits.learn.pipeline import Pipeline

###############################################################################
# Download the data, if not already on disk
```

```
url = "http://people.csail.mit.edu/jrennie/20Newsgroups/20news-18828.tar.gz"
archive_name = "20news-18828.tar.gz"

if not os.path.exists(archive_name[:-7]):
    if not os.path.exists(archive_name):
        import urllib
        print "Downloading data, please Wait (14MB)..."
        print url
        opener = urllib.urlopen(url)
        open(archive_name, 'wb').write(opener.read())
        print

    import tarfile
    print "Decompressiong the archive: " + archive_name
    tarfile.open(archive_name, "r:gz").extractall()
    print


################################################################################
# Load some categories from the training set
categories = [
    'alt.atheism',
    'talk.religion.misc',
]
# Uncomment the following to do the analysis on all the categories
#categories = None

print "Loading 20 newsgroups dataset for categories:"
print categories

data = load_files('20news-18828', categories=categories)
print "%d documents" % len(data.filenames)
print "%d categories" % len(data.target_names)
print

################################################################################
# define a pipeline combining a text feature extractor with a simple
# classifier
pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', SGDClassifier()),
])

parameters = {
# uncommenting more parameters will give better exploring power but will
# increase processing time in a combinatorial way
    'vect__max_df': (0.5, 0.75, 1.0),
#    'vect__max_features': (None, 5000, 10000, 50000),
    'vect__analyzer__max_n': (1, 2), # words or bigrams
#    'tfidf__use_idf': (True, False),
    'clf__alpha': (0.00001, 0.000001),
    'clf__penalty': ('l2', 'elasticnet'),
#    'clf__n_iter': (10, 50, 80),
}

# find the best parameters for both the feature extraction and the
# classifier
```

```
grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1)

# cross-validation doesn't work if the length of the data is not known,
# hence use lists instead of iterators
text_docs = [file(f).read() for f in data.filenames]

print "Performing grid search..."
print "pipeline:", [name for name, _ in pipeline.steps]
print "parameters:"
pprint(parameters)
t0 = time()
grid_search.fit(text_docs, data.target)
print "done in %0.3fs" % (time() - t0)
print

print "Best score: %0.3f" % grid_search.best_score
print "Best parameters set:"
best_parameters = grid_search.best_estimator._get_params()
for param_name in sorted(parameters.keys()):
    print "\t%s: %r" % (param_name, best_parameters[param_name])
```

Logistic Regression

with l1 and l2 penalty

Python source code: `logistic_l1_l2_coef.py`

```
print __doc__

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD Style.

import numpy as np

from scikits.learn.linear_model import LogisticRegression
from scikits.learn import datasets

iris = datasets.load_iris()
X = iris.data
y = iris.target

# Set regularization parameter
C = 0.1

classifier_l1_LR = LogisticRegression(C=C, penalty='l1')
classifier_l2_LR = LogisticRegression(C=C, penalty='l2')
classifier_l1_LR.fit(X, y)
classifier_l2_LR.fit(X, y)

hyperplane_coefficients_l1_LR = classifier_l1_LR.coef_[:]
hyperplane_coefficients_l2_LR = classifier_l2_LR.coef_[:]

# hyperplane_coefficients_l1_LR contains zeros due to the
# L1 sparsity inducing norm

pct_non_zeros_l1_LR = np.mean(hyperplane_coefficients_l1_LR != 0) * 100
pct_non_zeros_l2_LR = np.mean(hyperplane_coefficients_l2_LR != 0) * 100
```

```
print "Percentage of non zeros coefficients (L1) : %f" % pct_non_zeros_l1_LR
print "Percentage of non zeros coefficients (L2) : %f" % pct_non_zeros_l2_LR
```

Classification of text documents: using a MLComp dataset

This is an example showing how the scikit-learn can be used to classify documents by topics using a bag-of-words approach. This example uses a scipy.sparse matrix to store the features instead of standard numpy arrays.

The dataset used in this example is the 20 newsgroups dataset and should be downloaded from the http://mlcomp.org (free registration required):

> http://mlcomp.org/datasets/379

Once downloaded unzip the arhive somewhere on your filesystem. For instance in:

```
% mkdir -p ~/data/mlcomp
% cd  ~/data/mlcomp
% unzip /path/to/dataset-379-20news-18828_XXXXX.zip
```

You should get a folder `~/data/mlcomp/379` with a file named `metadata` and subfolders `raw`, `train` and `test` holding the text documents organized by newsgroups.

Then set the `MLCOMP_DATASETS_HOME` environment variable pointing to the root folder holding the uncompressed archive:

```
% export MLCOMP_DATASETS_HOME="~/data/mlcomp"
```

Then you are ready to run this example using your favorite python shell:

```
% ipython examples/mlcomp_sparse_document_classification.py
```

Python source code: `mlcomp_sparse_document_classification.py`

```
print __doc__

# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: Simplified BSD

from time import time
import sys
import os
import numpy as np
import scipy.sparse as sp
import pylab as pl

from scikits.learn.datasets import load_mlcomp
from scikits.learn.feature_extraction.text.sparse import Vectorizer
from scikits.learn.linear_model.sparse import SGDClassifier
from scikits.learn.metrics import confusion_matrix
from scikits.learn.metrics import classification_report

if 'MLCOMP_DATASETS_HOME' not in os.environ:
    print "Please follow those instructions to get started:"
    sys.exit(0)

# Load the training set
print "Loading 20 newsgroups training set... "
news_train = load_mlcomp('20news-18828', 'train')
```

```
print news_train.DESCR
print "%d documents" % len(news_train.filenames)
print "%d categories" % len(news_train.target_names)

print "Extracting features from the dataset using a sparse vectorizer"
t0 = time()
vectorizer = Vectorizer()
X_train = vectorizer.fit_transform((open(f).read()
                                    for f in news_train.filenames))
print "done in %fs" % (time() - t0)
print "n_samples: %d, n_features: %d" % X_train.shape
assert sp.issparse(X_train)
y_train = news_train.target

print "Training a linear classifier..."
parameters = {
    'loss': 'hinge',
    'penalty': 'l2',
    'n_iter': 50,
    'alpha': 0.00001,
    'fit_intercept': True,
}
print "parameters:", parameters
t0 = time()
clf = SGDClassifier(**parameters).fit(X_train, y_train)
print "done in %fs" % (time() - t0)
print "Percentage of non zeros coef: %f" % (np.mean(clf.coef_ != 0) * 100)

print "Loading 20 newsgroups test set... "
news_test = load_mlcomp('20news-18828', 'test')
t0 = time()
print "done in %fs" % (time() - t0)

print "Predicting the labels of the test set..."
print "%d documents" % len(news_test.filenames)
print "%d categories" % len(news_test.target_names)

print "Extracting features from the dataset using the same vectorizer"
t0 = time()
X_test = vectorizer.transform((open(f).read() for f in news_test.filenames))
y_test = news_test.target
print "done in %fs" % (time() - t0)
print "n_samples: %d, n_features: %d" % X_test.shape

print "Predicting the outcomes of the testing set"
t0 = time()
pred = clf.predict(X_test)
print "done in %fs" % (time() - t0)

print "Classification report on test set for classifier:"
print clf
print
print classification_report(y_test, pred, class_names=news_test.target_names)

cm = confusion_matrix(y_test, pred)
print "Confusion matrix:"
print cm
```

```
# Show confusion matrix
pl.matshow(cm)
pl.title('Confusion matrix')
pl.colorbar()
pl.show()
```

Gaussian Naive Bayes

A classification example using Gaussian Naive Bayes (GNB).

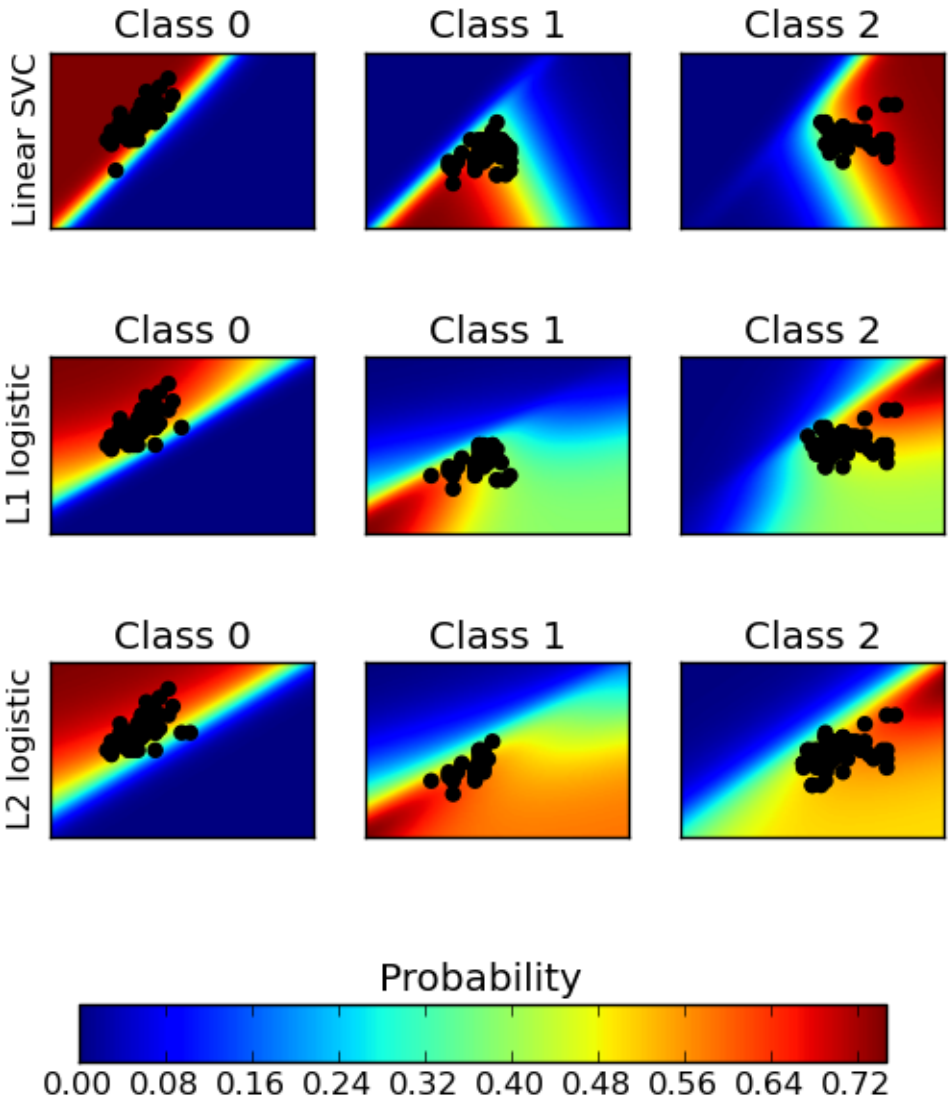
Python source code: `naive_bayes.py`

```
##############################################################################
# import some data to play with

# The IRIS dataset
from scikits.learn import datasets
iris = datasets.load_iris()

X = iris.data
y = iris.target

##############################################################################
# GNB
from scikits.learn.naive_bayes import GNB
gnb = GNB()

y_pred = gnb.fit(X, y).predict(X)

print "Number of mislabeled points : %d" % (y != y_pred).sum()
```

Plot classification probability

Plot the classification probability for different classifiers. We use a 3 class dataset, and we classify it with a Support Vector classifier, as well as L1 and L2 penalized logistic regression.

The logistic regression is not a multiclass classifier out of the box. As a result it can identify only the first class.



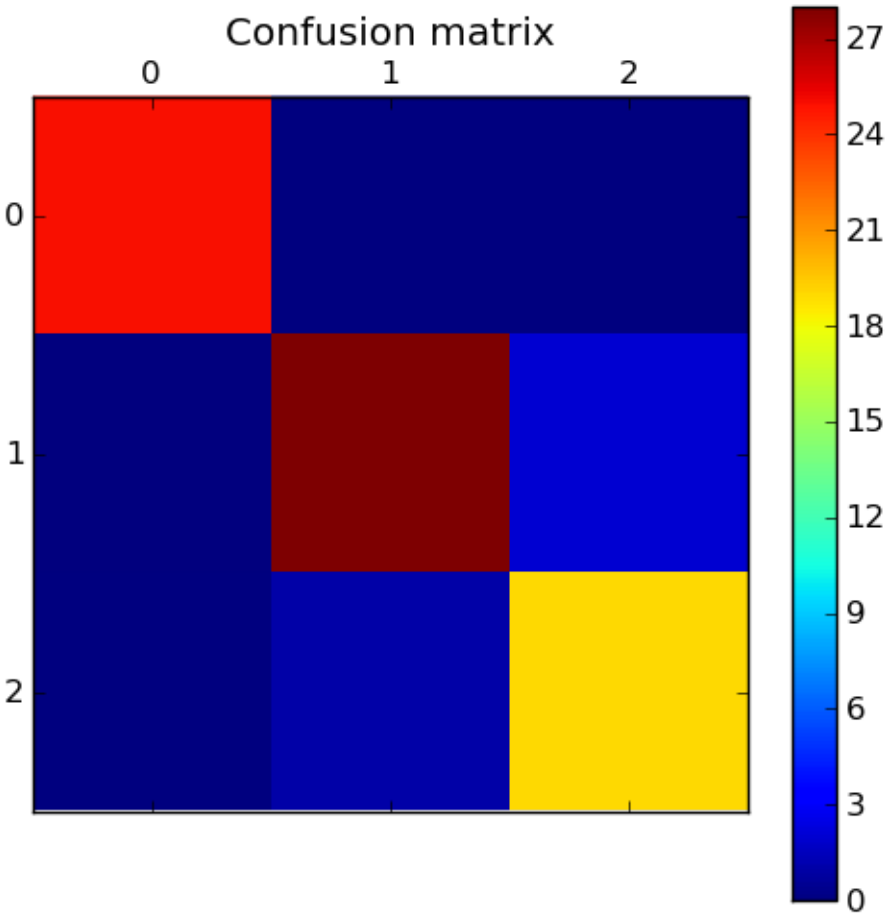
Python source code: `plot_classification_probability.py`

```
# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD Style.

# $Id$

import pylab as pl
import numpy as np

from scikits.learn.linear_model import LogisticRegression
from scikits.learn.svm import SVC
from scikits.learn import datasets

iris = datasets.load_iris()
X = iris.data[:, :2] # we only take the first two features for visualization
y = iris.target

n_features = X.shape[1]
```

```
C = 1.0

# Create different classifiers. The logistic regression cannot do
# multiclass out of the box.
classifiers = {
                'L1 logistic': LogisticRegression(C=C, penalty='l1'),
                'L2 logistic': LogisticRegression(C=C, penalty='l2'),
                'Linear SVC': SVC(kernel='linear', C=C, probability=True),
              }

n_classifiers = len(classifiers)

pl.figure(figsize=(3*2, n_classifiers*2))
pl.subplots_adjust(bottom=.2, top=.95)

for index, (name, classifier) in enumerate(classifiers.iteritems()):
    classifier.fit(X, y)

    y_pred = classifier.predict(X)
    classif_rate = np.mean(y_pred.ravel() == y.ravel()) * 100
    print  "classif_rate for %s : %f " % (name, classif_rate)

    # View probabilities=
    xx = np.linspace(3,9,100)
    yy = np.linspace(1,5,100).T
    xx, yy = np.meshgrid(xx, yy)
    Xfull = np.c_[xx.ravel(),yy.ravel()]
    probas = classifier.predict_proba(Xfull)
    n_classes = np.unique(y_pred).size
    for k in range(n_classes):
        pl.subplot(n_classifiers, n_classes, index*n_classes + k + 1)
        pl.title("Class %d" % k)
        if k == 0:
            pl.ylabel(name)
        imshow_handle = pl.imshow(probas[:, k].reshape((100, 100)),
                                  extent=(3, 9, 1, 5), origin='lower')
        pl.xticks(())
        pl.yticks(())
        idx = (y_pred == k)
        if idx.any():
            pl.scatter(X[idx, 0], X[idx, 1], marker='o', c='k')

ax = pl.axes([0.15, 0.04, 0.7, 0.05])
pl.title("Probability")
pl.colorbar(imshow_handle, cax=ax, orientation='horizontal')

pl.show()
```

Confusion matrix

Example of confusion matrix usage to evaluate the quality of the output of a classifier.



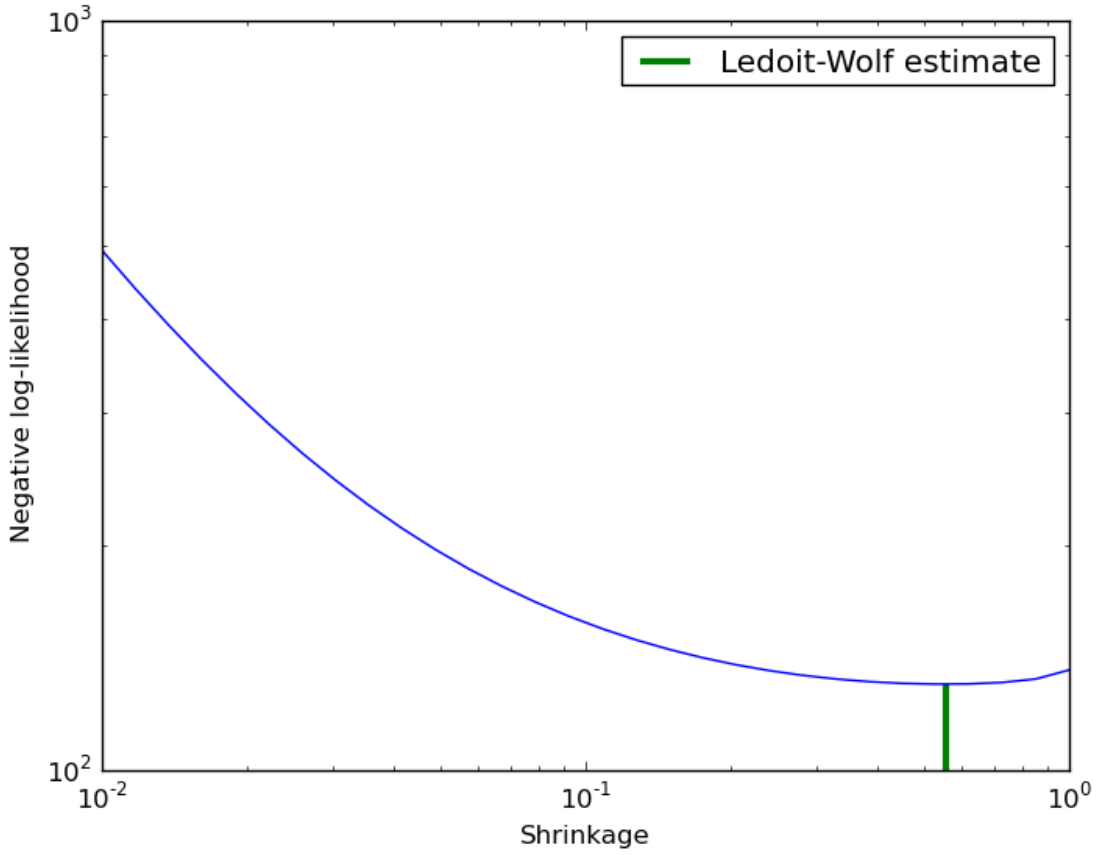
Python source code: `plot_confusion_matrix.py`

```
print __doc__

import random
import pylab as pl
from scikits.learn import svm, datasets
from scikits.learn.metrics import confusion_matrix

# import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target
n_samples, n_features = X.shape
p = range(n_samples)
random.seed(0)
random.shuffle(p)
X, y = X[p], y[p]
half = int(n_samples/2)
```

```
# Run classifier
classifier = svm.SVC(kernel='linear')
y_ = classifier.fit(X[:half],y[:half]).predict(X[half:])

# Compute confusion matrix
cm = confusion_matrix(y[half:], y_)

print cm

# Show confusion matrix
pl.matshow(cm)
pl.title('Confusion matrix')
pl.colorbar()
pl.show()
```

Ledoit-Wolf vs Covariance simple estimation

Covariance estimation can be regularized using a shrinkage parameter. Ledoit-Wolf estimates automatically this parameter. In this example, we compute the likelihood of unseen data for different values of the shrinkage parameter. The Ledoit-Wolf estimate reaches an almost optimal value.

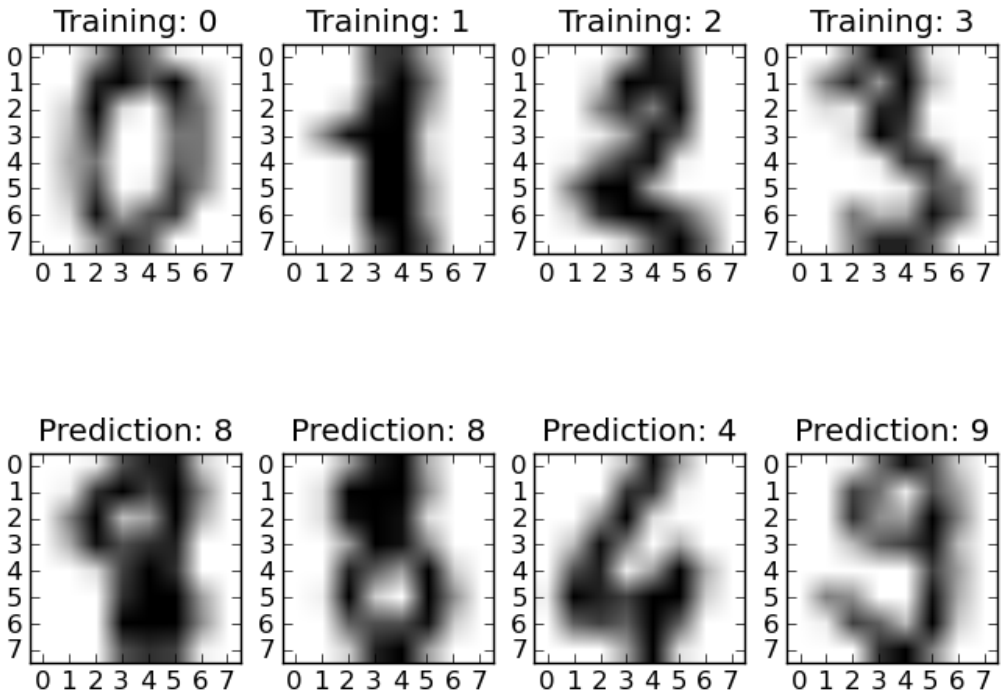


Python source code: `plot_covariance_estimation.py`

```
print __doc__

import numpy as np
import pylab as pl

###############################################################################
# Generate sample data
n_features, n_samples = 30, 20
X_train = np.random.normal(size=(n_samples, n_features))
X_test = np.random.normal(size=(n_samples, n_features))

# Color samples
coloring_matrix = np.random.normal(size=(n_features, n_features))
X_train = np.dot(X_train, coloring_matrix)
X_test = np.dot(X_test, coloring_matrix)

###############################################################################
# Compute Ledoit-Wolf and Covariances on a grid of shrinkages

from scikits.learn.covariance import LedoitWolf, ShrunkCovariance

lw = LedoitWolf()
loglik_lw = lw.fit(X_train).score(X_test)

cov = ShrunkCovariance()
shrinkages = np.logspace(-2, 0, 30)
negative_logliks = [-cov.fit(X_train, shrinkage=s).score(X_test) \
                                                        for s in shrinkages]

###############################################################################
# Plot results
pl.loglog(shrinkages, negative_logliks)
pl.xlabel('Shrinkage')
pl.ylabel('Negative log-likelihood')
pl.vlines(lw.shrinkage_, pl.ylim()[0], -loglik_lw, color='g',
                        linewidth=3, label='Ledoit-Wolf estimate')
pl.legend()
pl.show()
```

Recognizing hand-written digits

An example showing how the scikit-learn can be used to recognize images of hand-written digits.



Python source code: `plot_digits_classification.py`

```
print __doc__

# Author: Gael Varoquaux <gael dot varoquaux at normalesup dot org>
# License: Simplified BSD

# Standard scientific Python imports
import pylab as pl

# The digits dataset
from scikits.learn import datasets
digits = datasets.load_digits()

# The data that we are interested in is made of 8x8 images of digits,
# let's have a look at the first 3 images. We know which digit they
# represent: it is given in the 'target' of the dataset.
for index, (image, label) in enumerate(zip(digits.images, digits.target)[:4]):
    pl.subplot(2, 4, index+1)
    pl.imshow(image, cmap=pl.cm.gray_r)
    pl.title('Training: %i' % label)

# To apply an classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))
```

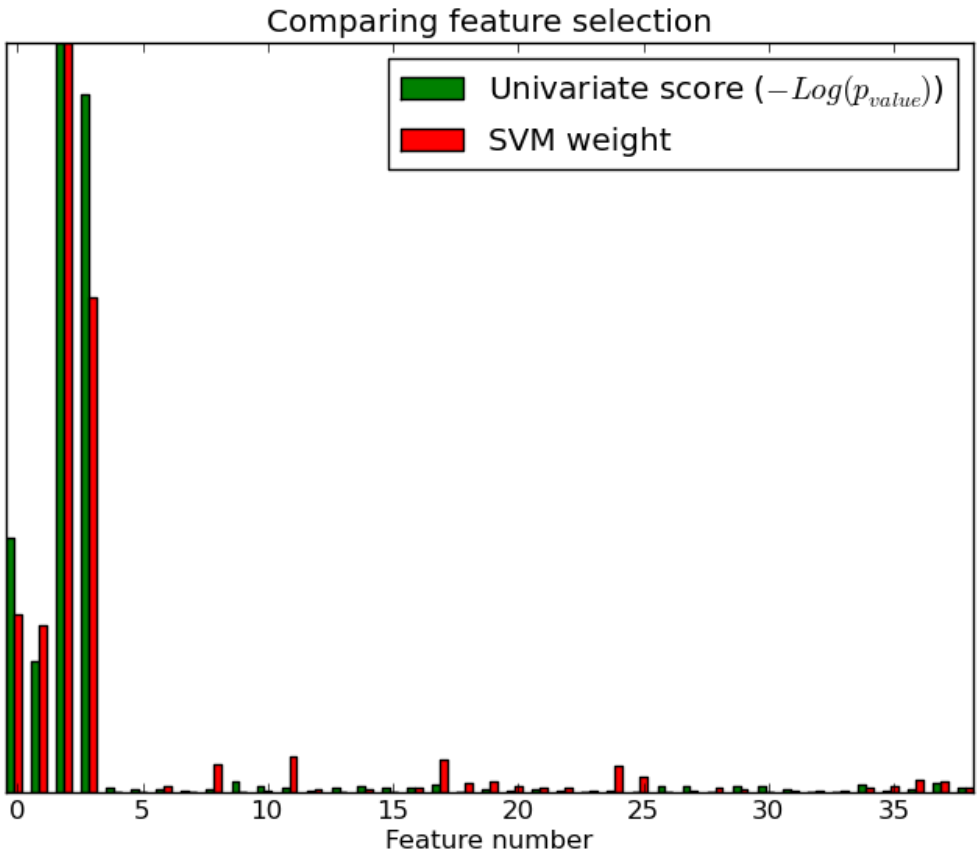
```
# Import a classifier:
from scikits.learn import svm
from scikits.learn.metrics import classification_report
from scikits.learn.metrics import confusion_matrix
classifier = svm.SVC()

# We learn the digits on the first half of the digits
classifier.fit(data[:n_samples/2], digits.target[:n_samples/2])

# Now predict the value of the digit on the second half:
expected = digits.target[n_samples/2:]
predicted = classifier.predict(data[n_samples/2:])

print "Classification report for classifier:"
print classifier
print
print classification_report(expected, predicted)
print
print "Confusion matrix:"
print confusion_matrix(expected, predicted)

for index, (image, prediction) in enumerate(
    zip(digits.images[n_samples/2:], predicted)[:4]):
    pl.subplot(2, 4, index+5)
    pl.imshow(image, cmap=pl.cm.gray_r)
    pl.title('Prediction: %i' % prediction)

pl.show()
```

Univariate Feature Selection

An example showing univariate feature selection.

Noisy (non informative) features are added to the iris data and univariate feature selection is applied. For each feature, we plot the p-values for the univariate feature selection and the corresponding weights of an SVM. We can see that univariate feature selection selects the informative features and that these have larger SVM weights.

In the total set of features, only the 4 first ones are significant. We can see that they have the highest score with univariate feature selection. The SVM attributes small weights to these features, but these weight are non zero. Applying univariate feature selection before the SVM increases the SVM weight attributed to the significant features, and will thus improve classification.



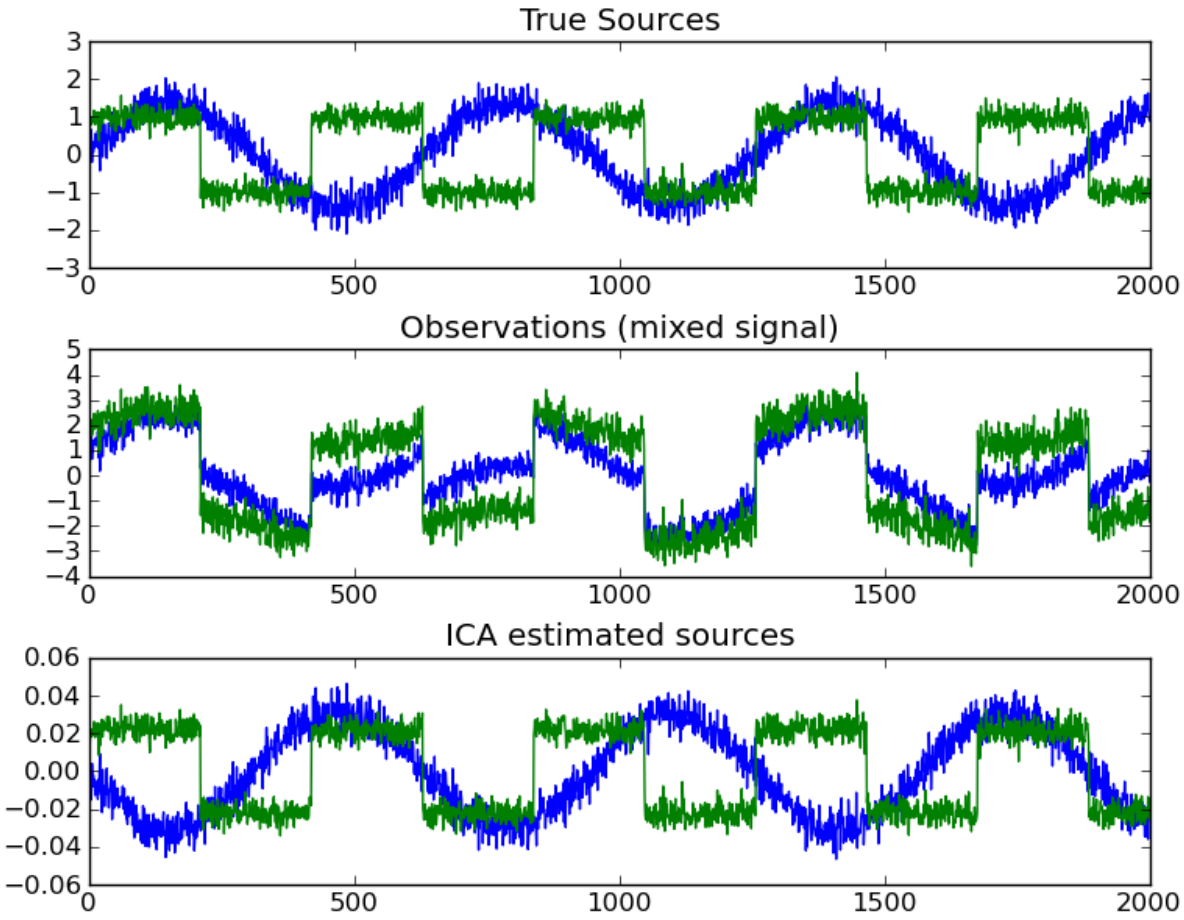
Python source code: `plot_feature_selection.py`

```
print __doc__

import numpy as np
import pylab as pl

###############################################################################
# import some data to play with

# The IRIS dataset
from scikits.learn import datasets, svm
iris = datasets.load_iris()

# Some noisy data not correlated
E = np.random.normal(size=(len(iris.data), 35))

# Add the noisy data to the informative features
x = np.hstack((iris.data, E))
y = iris.target

###############################################################################
pl.figure(1)
pl.clf()

x_indices = np.arange(x.shape[-1])
```

```
###############################################################################
# Univariate feature selection
from scikits.learn.feature_selection import SelectFpr, f_classif
# As a scoring function, we use a F test for classification
# We use the default selection function: the 10% most significant
# features

selector = SelectFpr(f_classif, alpha=0.1)
selector.fit(x, y)
scores = -np.log10(selector._pvalues)
scores /= scores.max()
pl.bar(x_indices-.45, scores, width=.3,
        label=r'Univariate score ($-Log(p_{value})$)',
        color='g')

###############################################################################
# Compare to the weights of an SVM
clf = svm.SVC(kernel='linear')
clf.fit(x, y)

svm_weights = (clf.coef_**2).sum(axis=0)
svm_weights /= svm_weights.max()
pl.bar(x_indices-.15, svm_weights, width=.3, label='SVM weight',
        color='r')


# ###############################################################################
# # Now fit an SVM with added feature selection
# selector = univ_selection.Univ(
#                 score_func=univ_selection.f_classif)

# selector.fit(x, clf.predict(x))
# svm_weights = (clf.support_**2).sum(axis=0)
# svm_weights /= svm_weights.max()
# full_svm_weights = np.zeros(selector.support_.shape)
# full_svm_weights[selector.support_] = svm_weights
# pl.bar(x_indices+.15, full_svm_weights, width=.3,
#         label='SVM weight after univariate selection',
#         color='b')

pl.title("Comparing feature selection")
pl.xlabel('Feature number')
pl.yticks(())
pl.axis('tight')
pl.legend(loc='upper right')
pl.show()
```

Blind source separation using FastICA

Independent component analysis (ICA) is used to estimate sources given noisy measurements. Imagine 2 instruments playing simultaneously and 2 microphones recording the mixed signals. ICA is used to recover the sources ie. what is played by each instrument.



Python source code: `plot_ica_blind_source_separation.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn.fastica import FastICA

###############################################################################
# Generate sample data
np.random.seed(0)
n_samples = 2000
time = np.linspace(0, 10, n_samples)
s1 = np.sin(2*time) # Signal 1 : sinusoidal signal
s2 = np.sign(np.sin(3*time)) # Signal 2 : square signal
S = np.c_[s1,s2].T
S += 0.2*np.random.normal(size=S.shape) # Add noise

S /= S.std(axis=1)[:,np.newaxis] # Standardize data
# Mix data
A = [[1, 1], [0.5, 2]] # Mixing matrix
X = np.dot(A, S) # Generate observations
# Compute ICA
ica = FastICA()
S_ = ica.fit(X).transform(X) # Get the estimated sources
A_ = ica.get_mixing_matrix() # Get estimated mixing matrix
```

```
assert np.allclose(X, np.dot(A_, S_))

###############################################################################
# Plot results
pl.figure()
pl.subplot(3, 1, 1)
pl.plot(S.T)
pl.title('True Sources')
pl.subplot(3, 1, 2)
pl.plot(X.T)
pl.title('Observations (mixed signal)')
pl.subplot(3, 1, 3)
pl.plot(S_.T)
pl.title('ICA estimated sources')
pl.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.36)
pl.show()
```

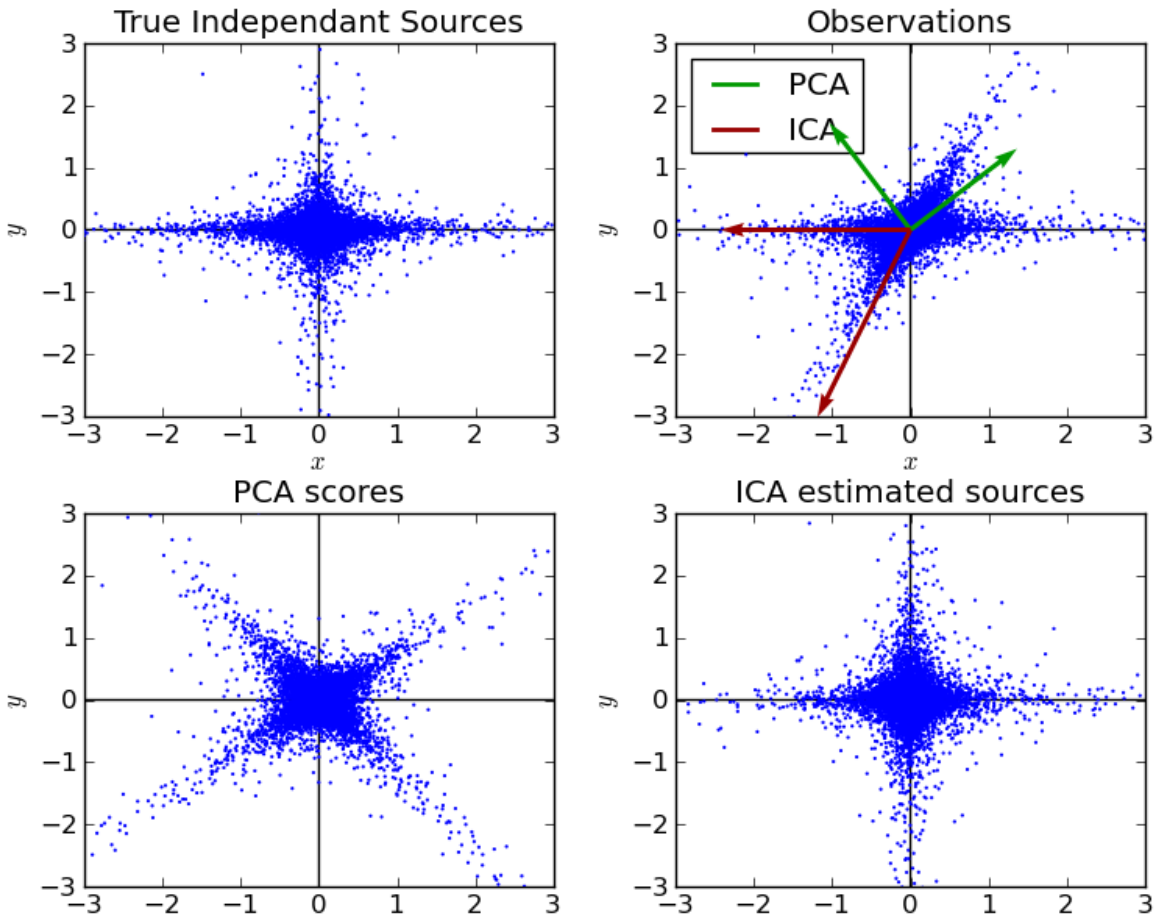
FastICA on 2D point clouds

Illustrate visually the results of *Independent component analysis (ICA)* vs *Principal component analysis (PCA)* in the feature space.

Representing ICA in the feature space gives the view of 'geometric ICA': ICA is an algorithm that finds directions in the feature space corresponding to projections with high non-Gaussianity. These directions need not be orthogonal in the original feature space, but they are orthogonal in the whitened feature space, in which all directions correspond to the same variance.

PCA, on the other hand, finds orthogonal directions in the raw feature space that correspond to directions accoutning for maximum variance.

Here we simulate independent sources using a highly non-Gaussian process, 2 student T with a low number of degrees of freedom (top left figure). We mix them to create observations (top right figure). In this raw observation space, directions identified by PCA are represented by green vectors. We represent the signal in the PCA space, after whitening by the variance corresponding to the PCA vectors (lower left). Running ICA corresponds to finding a rotation in this space to identify the directions of largest non-Gaussianity (lower right).



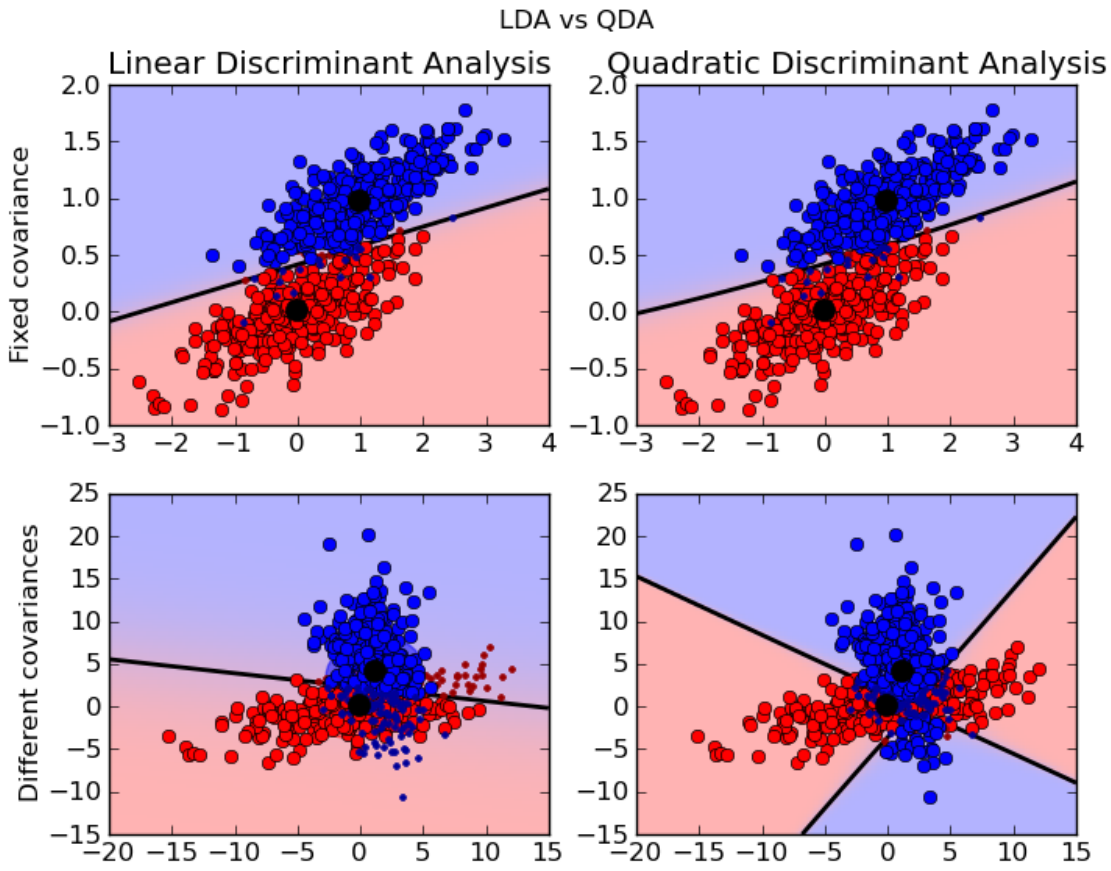
Python source code: `plot_ica_vs_pca.py`

```
print __doc__

# Authors: Alexandre Gramfort, Gael Varoquaux
# License: BSD

import numpy as np
import pylab as pl

from scikits.learn.pca import PCA
from scikits.learn.fastica import FastICA

###############################################################################
# Generate sample data
S = np.random.standard_t(1.5, size=(2, 10000))
S[0] *= 2.

# Mix data
A = [[1, 1], [0, 2]] # Mixing matrix

X = np.dot(A, S) # Generate observations

pca = PCA()
S_pca_ = pca.fit(X.T).transform(X.T).T
```

```
ica = FastICA()
S_ica_ = ica.fit(X).transform(X)  # Estimate the sources

S_ica_ /= S_ica_.std(axis=1)[:,np.newaxis]

###############################################################################
# Plot results

def plot_samples(S, axis_list=None):
    pl.scatter(S[0], S[1], s=2, marker='o', linewidths=0, zorder=10)
    if axis_list is not None:
        colors = [(0, 0.6, 0), (0.6, 0, 0)]
        for color, axis in zip(colors, axis_list):
            axis /= axis.std()
            x_axis, y_axis = axis
            # Trick to get legend to work
            pl.plot(0.1*x_axis, 0.1*y_axis, linewidth=2, color=color)
            # pl.quiver(x_axis, y_axis, x_axis, y_axis, zorder=11, width=0.01,
            pl.quiver(0, 0, x_axis, y_axis, zorder=11, width=0.01,
                      scale=6, color=color)

    pl.hlines(0, -3, 3)
    pl.vlines(0, -3, 3)
    pl.xlim(-3, 3)
    pl.ylim(-3, 3)
    pl.xlabel('$x$')
    pl.ylabel('$y$')

pl.subplot(2, 2, 1)
plot_samples(S / S.std())
pl.title('True Independant Sources')

axis_list = [pca.components_, ica.get_mixing_matrix()]
pl.subplot(2, 2, 2)
plot_samples(X / np.std(X), axis_list=axis_list)
pl.legend(['PCA', 'ICA'], loc='upper left')
pl.title('Observations')

pl.subplot(2, 2, 3)
plot_samples(S_pca_ / np.std(S_pca_, axis=-1)[:, np.newaxis])
pl.title('PCA scores')

pl.subplot(2, 2, 4)
plot_samples(S_ica_ / np.std(S_ica_))
pl.title('ICA estimated sources')

pl.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.26)

pl.show()
```

Linear Discriminant Analysis & Quadratic Discriminant Analysis with confidence

Plot the decision boundary



Python source code: `plot_lda_qda.py`

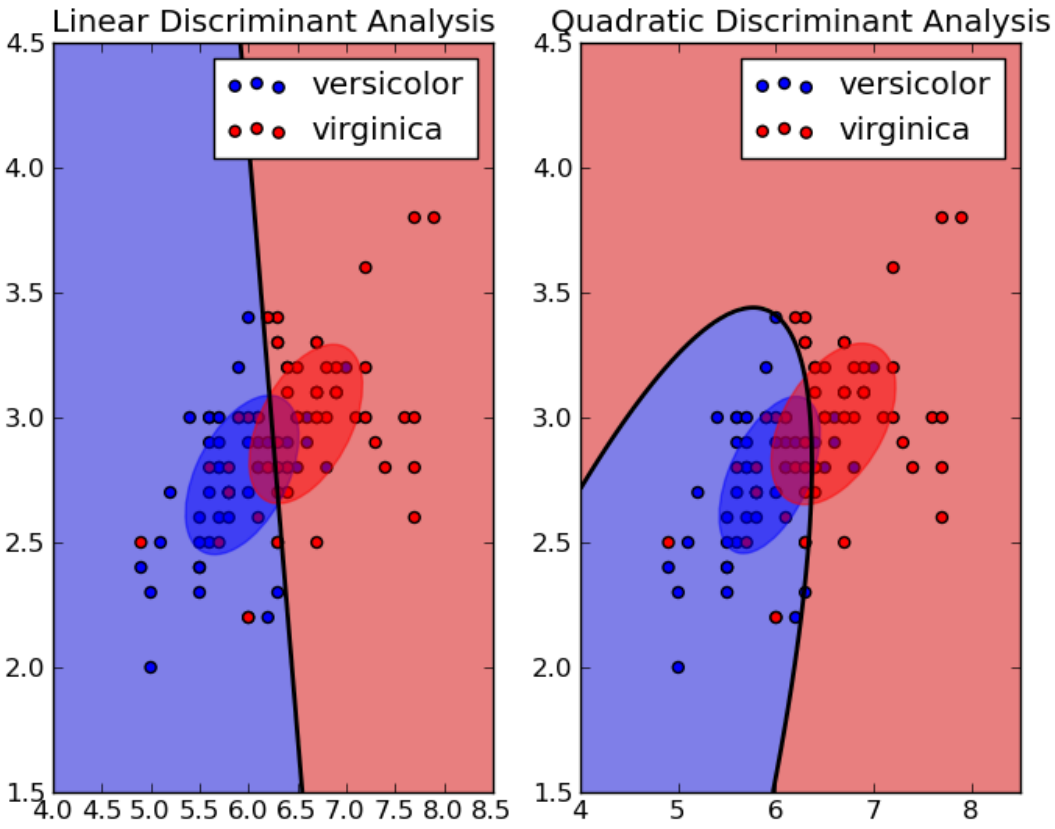
```
print __doc__

from scipy import linalg
import numpy as np
import pylab as pl
import matplotlib as mpl
from matplotlib import colors

from scikits.learn.lda import LDA
from scikits.learn.qda import QDA

###############################################################################
# colormap
cmap = colors.LinearSegmentedColormap('red_blue_classes',
    {'red' : [(0, 1, 1), (1, 0.7, 0.7)],
     'green' : [(0, 0.7, 0.7), (1, 0.7, 0.7)],
     'blue' : [(0, 0.7, 0.7), (1, 1, 1)]
    })
pl.cm.register_cmap(cmap=cmap)


###############################################################################
# generate datasets
def dataset_fixed_cov():
```

```
    '''Generate 2 Gaussians samples with the same covariance matrix'''
    n, dim = 300, 2
    np.random.seed(0)
    C = np.array([[0., -0.23], [0.83, .23]])
    X = np.r_[np.dot(np.random.randn(n, dim), C),
              np.dot(np.random.randn(n, dim), C) + np.array([1, 1])]
    y = np.hstack((np.zeros(n), np.ones(n)))
    return X, y

def dataset_cov():
    '''Generate 2 Gaussians samples with different covariance matrices'''
    n, dim = 300, 2
    np.random.seed(0)
    C = np.array([[0., -1.], [2.5, .7]]) * 2.
    X = np.r_[np.dot(np.random.randn(n, dim), C),
              np.dot(np.random.randn(n, dim), C.T) + np.array([1, 4])]
    y = np.hstack((np.zeros(n), np.ones(n)))
    return X, y

###############################################################################
# plot functions
def plot_data(lda, X, y, y_pred, fig_index):
    splot = pl.subplot(2, 2, fig_index)
    if fig_index == 1:
        pl.title('Linear Discriminant Analysis')
        pl.ylabel('Fixed covariance')
    elif fig_index == 2:
        pl.title('Quadratic Discriminant Analysis')
    elif fig_index == 3:
        pl.ylabel('Different covariances')

    tp = (y == y_pred) # True Positive
    tp0, tp1 = tp[y == 0], tp[y == 1]
    X0, X1 = X[y == 0], X[y == 1]
    X0_tp, X0_fp = X0[tp0], X0[tp0 != True]
    X1_tp, X1_fp = X1[tp1], X1[tp1 != True]
    xmin, xmax = X[:, 0].min(), X[:, 0].max()
    ymin, ymax = X[:, 1].min(), X[:, 1].max()

    # class 0: dots
    pl.plot(X0_tp[:, 0], X0_tp[:, 1], 'o', color='red')
    pl.plot(X0_fp[:, 0], X0_fp[:, 1], '.', color='#990000') # dark red

    # class 1: dots
    pl.plot(X1_tp[:, 0], X1_tp[:, 1], 'o', color='blue')
    pl.plot(X1_fp[:, 0], X1_fp[:, 1], '.', color='#000099') # dark blue

    # class 0 and 1 : areas
    nx, ny = 200, 100
    x_min, x_max = pl.xlim()
    y_min, y_max = pl.ylim()
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
                         np.linspace(y_min, y_max, ny))
    Z = lda.predict_proba(np.c_[xx.ravel(), yy.ravel()])
    Z = Z[:, 1].reshape(xx.shape)
    pl.pcolormesh(xx, yy, Z, cmap='red_blue_classes',
                  norm=colors.Normalize(0., 1.))
    pl.contour(xx, yy, Z, [0.5], linewidths=2., colors='k')
```

```
    # means
    pl.plot(lda.means_[0][0], lda.means_[0][1],
            'o', color='black', markersize=10)
    pl.plot(lda.means_[1][0], lda.means_[1][1],
            'o', color='black', markersize=10)

    return splot

def plot_ellipse(splot, mean, cov, color):
    v, w = linalg.eigh(cov)
    u = w[0] / linalg.norm(w[0])
    angle = np.arctan(u[1]/u[0])
    angle = 180 * angle / np.pi # convert to degrees
    # filled gaussian at 2 standard deviation
    ell = mpl.patches.Ellipse(mean, 2 * v[0] ** 0.5, 2 * v[1] ** 0.5,
                                          180 + angle, color=color)
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(0.5)
    splot.add_artist(ell)

def plot_lda_cov(lda, splot):
    plot_ellipse(splot, lda.means_[0], lda.covariance_, 'red')
    plot_ellipse(splot, lda.means_[1], lda.covariance_, 'blue')

def plot_qda_cov(qda, splot):
    plot_ellipse(splot, qda.means_[0], qda.covariances_[0], 'red')
    plot_ellipse(splot, qda.means_[1], qda.covariances_[1], 'blue')

##############################################################################
for i, (X, y) in enumerate([dataset_fixed_cov(), dataset_cov()]):
    # LDA
    lda = LDA()
    y_pred = lda.fit(X, y, store_covariance=True).predict(X)
    splot = plot_data(lda, X, y, y_pred, fig_index=2 * i + 1)
    plot_lda_cov(lda, splot)
    pl.axis('tight')

    # QDA
    qda = QDA()
    y_pred = qda.fit(X, y, store_covariances=True).predict(X)
    splot = plot_data(qda, X, y, y_pred, fig_index=2 * i + 2)
    plot_qda_cov(qda, splot)
    pl.axis('tight')
pl.suptitle('LDA vs QDA')
pl.show()
```

Linear and Quadratic Discriminant Analysis with confidence ellipsoid

Plot the confidence ellipsoids of each class and decision boundary



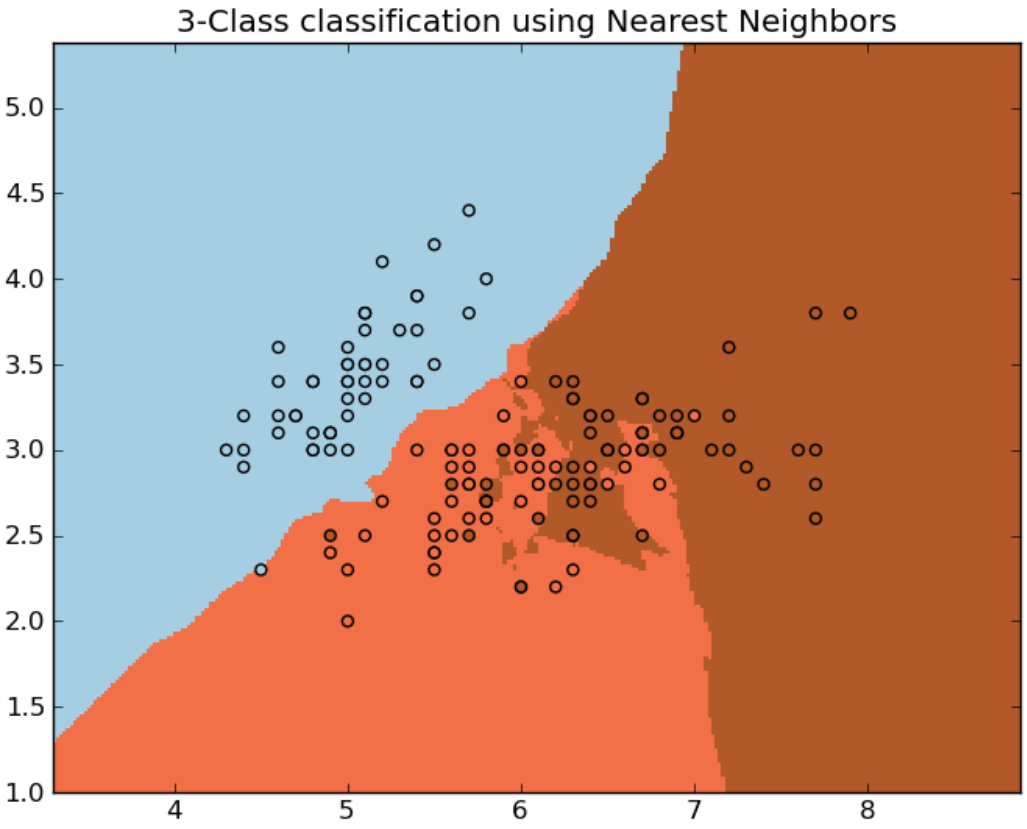
Python source code: `plot_lda_vs_qda.py`

```
print __doc__

from scipy import linalg
import numpy as np
import pylab as pl
import matplotlib as mpl

from scikits.learn.lda import LDA
from scikits.learn.qda import QDA

###############################################################################
# load sample dataset
from scikits.learn.datasets import load_iris

iris = load_iris()
X = iris.data[:,:2] # Take only 2 dimensions
y = iris.target
X = X[y > 0]
y = y[y > 0]
y -= 1
target_names = iris.target_names[1:]

###############################################################################
# LDA
```

```
lda = LDA()
y_pred = lda.fit(X, y, store_covariance=True).predict(X)

# QDA
qda = QDA()
y_pred = qda.fit(X, y, store_covariances=True).predict(X)

##############################################################################
# Plot results

def plot_ellipse(splot, mean, cov, color):
    v, w = linalg.eigh(cov)
    u = w[0] / linalg.norm(w[0])
    angle = np.arctan(u[1]/u[0])
    angle = 180 * angle / np.pi # convert to degrees
    # filled gaussian at 2 standard deviation
    ell = mpl.patches.Ellipse(mean, 2 * v[0] ** 0.5, 2 * v[1] ** 0.5,
                                          180 + angle, color=color)
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(0.5)
    splot.add_artist(ell)

xx, yy = np.meshgrid(np.linspace(4, 8.5, 200), np.linspace(1.5, 4.5, 200))
X_grid = np.c_[xx.ravel(), yy.ravel()]
zz_lda = lda.predict_proba(X_grid)[:,1].reshape(xx.shape)
zz_qda = qda.predict_proba(X_grid)[:,1].reshape(xx.shape)

pl.figure()
splot = pl.subplot(1, 2, 1)
pl.contourf(xx, yy, zz_lda > 0.5, alpha=0.5)
pl.scatter(X[y==0,0], X[y==0,1], c='b', label=target_names[0])
pl.scatter(X[y==1,0], X[y==1,1], c='r', label=target_names[1])
pl.contour(xx, yy, zz_lda, [0.5], linewidths=2., colors='k')
plot_ellipse(splot, lda.means_[0], lda.covariance_, 'b')
plot_ellipse(splot, lda.means_[1], lda.covariance_, 'r')
pl.legend()
pl.axis('tight')
pl.title('Linear Discriminant Analysis')

splot = pl.subplot(1, 2, 2)
pl.contourf(xx, yy, zz_qda > 0.5, alpha=0.5)
pl.scatter(X[y==0,0], X[y==0,1], c='b', label=target_names[0])
pl.scatter(X[y==1,0], X[y==1,1], c='r', label=target_names[1])
pl.contour(xx, yy, zz_qda, [0.5], linewidths=2., colors='k')
plot_ellipse(splot, qda.means_[0], qda.covariances_[0], 'b')
plot_ellipse(splot, qda.means_[1], qda.covariances_[1], 'r')
pl.legend()
pl.axis('tight')
pl.title('Quadratic Discriminant Analysis')
pl.show()
```

Nearest Neighbors

Sample usage of Support Vector Machines to classify a sample. It will plot the decision surface and the support vectors.



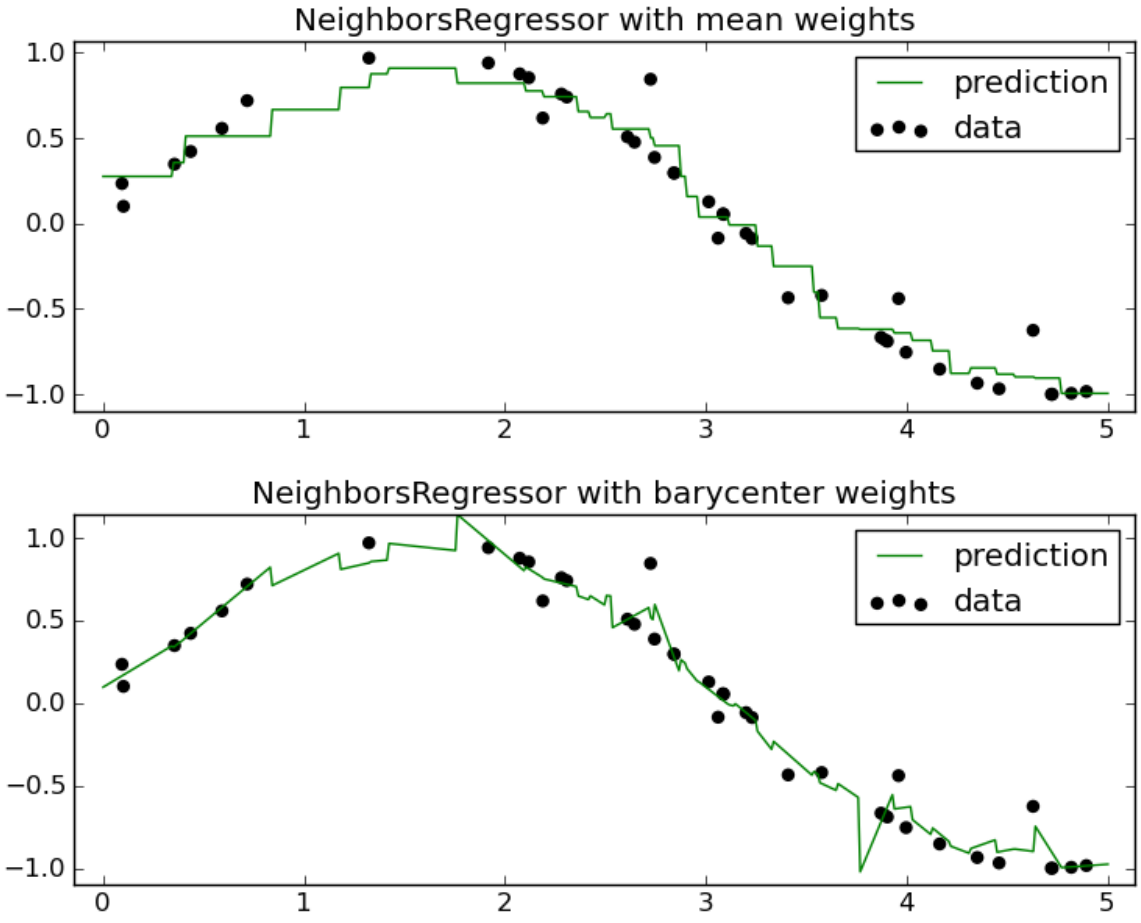
Python source code: `plot_neighbors.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn import neighbors, datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2] # we only take the first two features. We could
                     # avoid this ugly slicing by using a two-dim dataset
Y = iris.target

h = .02 # step size in the mesh

# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
clf = neighbors.NeighborsClassifier()
clf.fit(X, Y)

# Plot the decision boundary. For that, we will asign a color to each
# point in the mesh [x_min, m_max]x[y_min, y_max].
x_min, x_max = X[:,0].min()-1, X[:,0].max() + 1
y_min, y_max = X[:,1].min()-1, X[:,1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
```

```
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
pl.set_cmap(pl.cm.Paired)
pl.pcolormesh(xx, yy, Z)

# Plot also the training points
pl.scatter(X[:,0], X[:,1], c=Y)
# and the support vectors
pl.title('3-Class classification using Nearest Neighbors')
pl.axis('tight')
pl.show()
```

k-Nearest Neighbors regression

Demonstrate the resolution of a regression problem using a k-Nearest Neighbor and the interpolation of the target using both barycenter and constant weights.



Python source code: `plot_neighbors_regression.py`

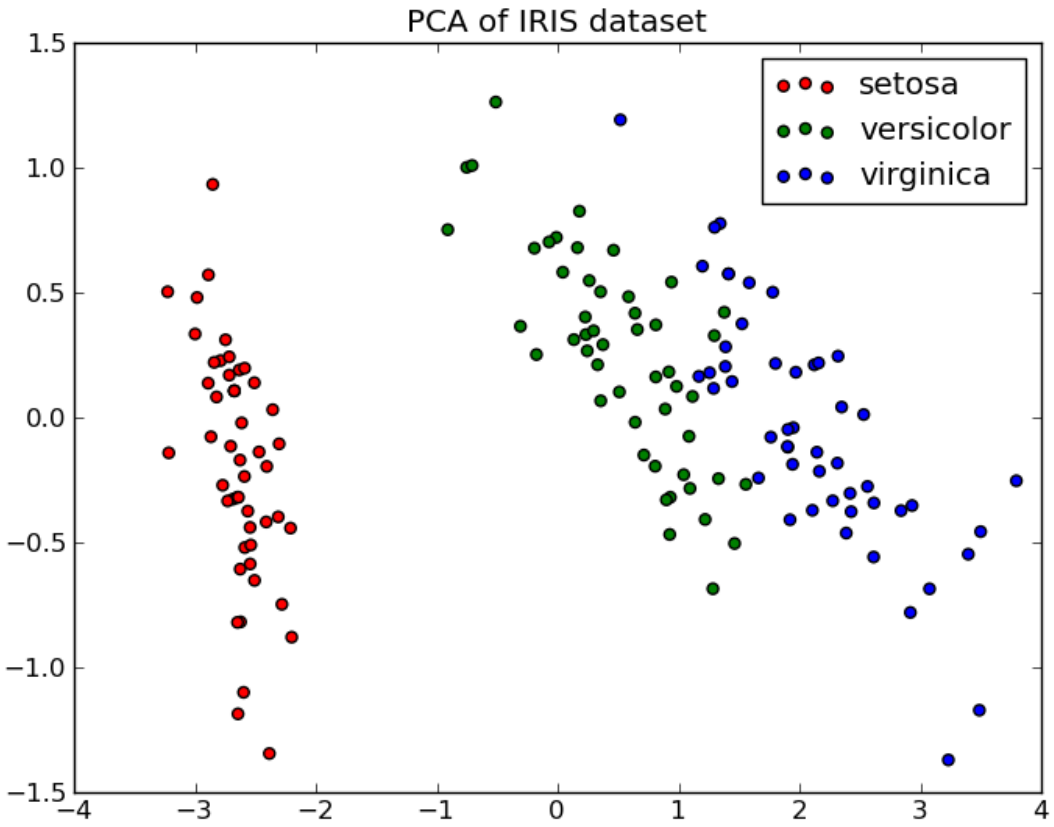
```
print __doc__

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
```

```
#         Fabian Pedregosa <fabian.pedregosa@inria.fr>
#
# License: BSD, (C) INRIA


###############################################################################
# Generate sample data
import numpy as np
import pylab as pl
from scikits.learn import neighbors

np.random.seed(0)
X = np.sort(5*np.random.rand(40, 1), axis=0)
T = np.linspace(0, 5, 500)[:, np.newaxis]
y = np.sin(X).ravel()

# Add noise to targets
y[::5] += 1*(0.5 - np.random.rand(8))

###############################################################################
# Fit regression model

for i, mode in enumerate(('mean', 'barycenter')):
    knn = neighbors.NeighborsRegressor(n_neighbors=4, mode=mode)
    y_ = knn.fit(X, y).predict(T)

    pl.subplot(2, 1, 1 + i)
    pl.scatter(X, y, c='k', label='data')
    pl.plot(T, y_, c='g', label='prediction')
    pl.axis('tight')
    pl.legend()
    pl.title('NeighborsRegressor with %s weights' % mode)

pl.subplots_adjust(0.1, 0.04, 0.95, 0.94, 0.3, 0.28)
pl.show()
```

PCA 2D projection of Iris dataset

The Iris dataset represents 3 kind of Iris flowers (Setosa, Versicolour and Virginica) with 4 attributes: sepal length, sepal width, petal length and petal width.

Principal Component Analysis (PCA) applied to this data identifies the combination of attributes (principal components, or directions in the feature space) that account for the most variance in the data. Here we plot the different samples on the 2 first principal components.



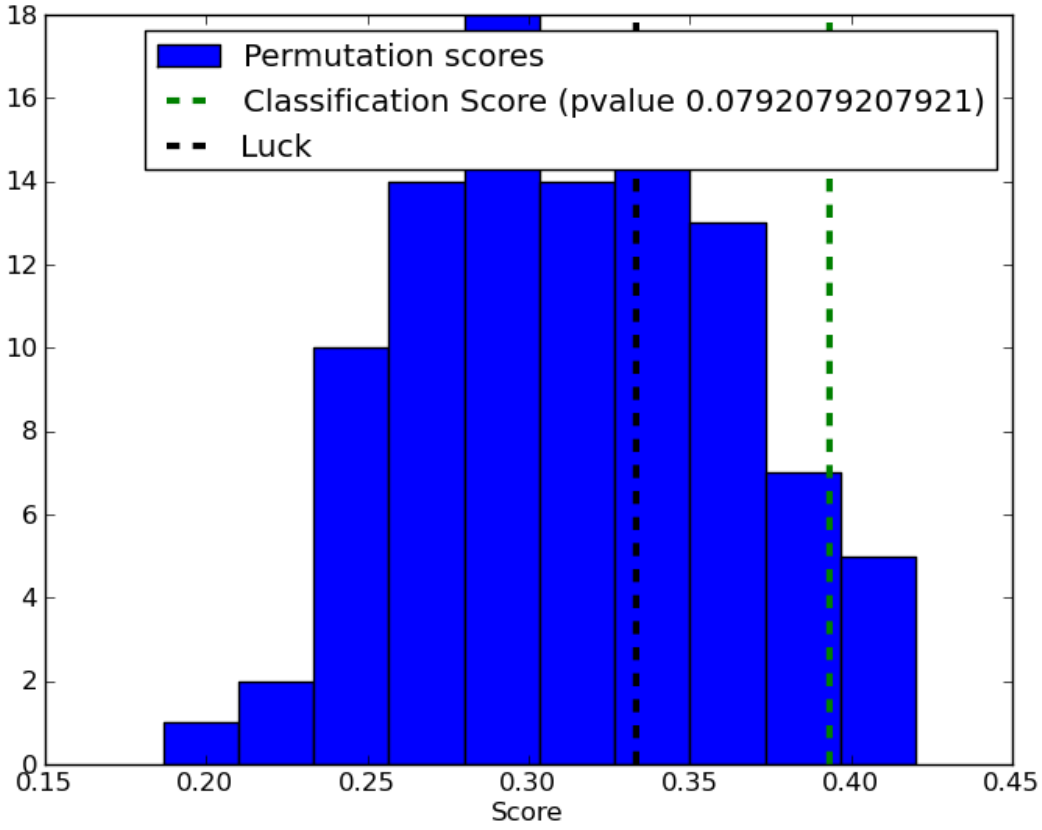
Python source code: `plot_pca.py`

```
print __doc__

import pylab as pl

from scikits.learn import datasets
from scikits.learn.pca import PCA

iris = datasets.load_iris()

X = iris.data
y = iris.target
target_names = iris.target_names

pca = PCA(n_components=2)
X_r = pca.fit(X).transform(X)

# Percentage of variance explained for each components
print pca.explained_variance_

pl.figure()
for c, i, target_name in zip("rgb", [0, 1, 2], target_names):
   pl.scatter(X_r[y==i,0], X_r[y==i,1], c=c, label=target_name)
pl.legend()
pl.title('PCA of IRIS dataset')
```

```
pl.show()
```

Test with permutations the significance of a classification score

In order to test if a classification score is significative a technique in repeating the classification procedure after randomizing, permuting, the labels. The p-value is then given by the percentage of runs for which the score obtained is greater than the classification score obtained in the first place.



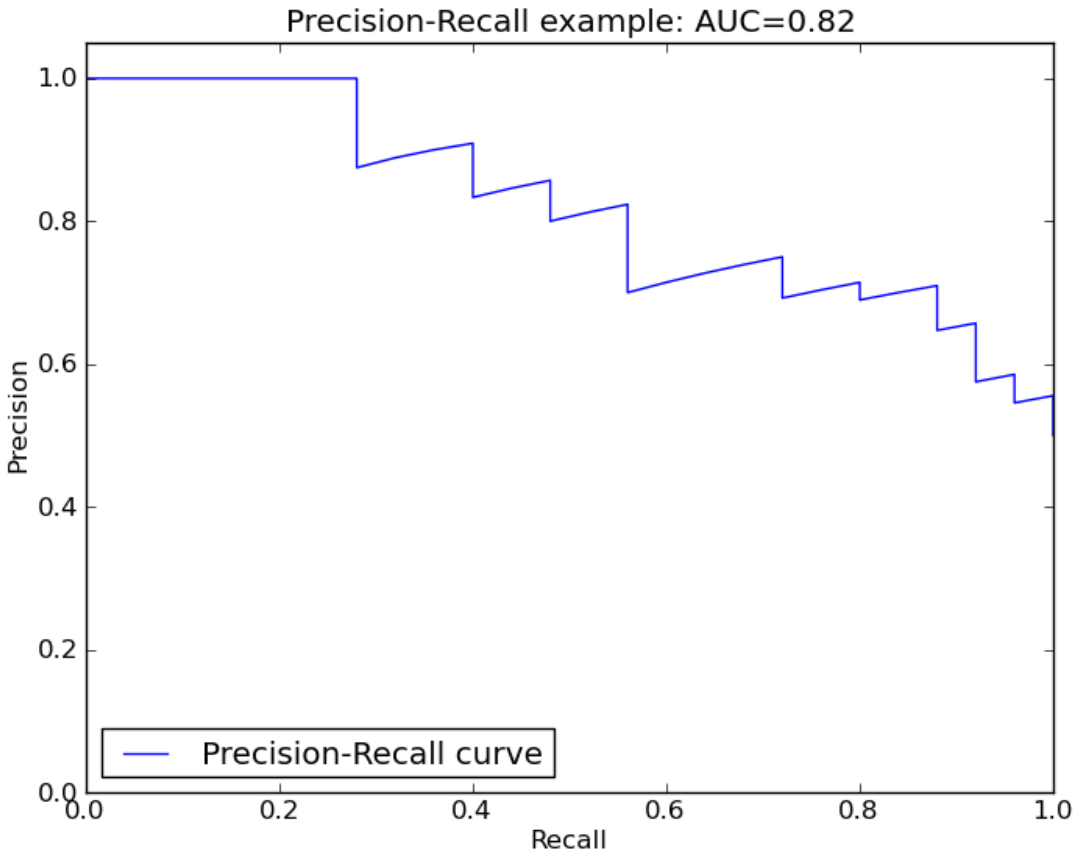
Python source code: `plot_permutation_test_for_classification.py`

```
# Author:  Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD

print __doc__

import numpy as np
import pylab as pl

from scikits.learn.svm import SVC
from scikits.learn.cross_val import StratifiedKFold, permutation_test_score
from scikits.learn import datasets
from scikits.learn.metrics import zero_one_score
```

```
##############################################################################
# Loading a dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target
n_classes = np.unique(y).size

# Some noisy data not correlated
random = np.random.RandomState(seed=0)
E = random.normal(size=(len(X), 2200))

# Add noisy data to the informative features for make the task harder
X = np.c_[X, E]

svm = SVC(kernel='linear')
cv = StratifiedKFold(y, 2)

score, permutation_scores, pvalue = permutation_test_score(svm, X, y,
                                         zero_one_score, cv=cv,
                                         n_permutations=100, n_jobs=1)

print "Classification score %s (pvalue : %s)" % (score, pvalue)

##############################################################################
# View histogram of permutation scores
pl.hist(permutation_scores, label='Permutation scores')
ylim = pl.ylim()
pl.vlines(score, ylim[0], ylim[1], linestyle='--',
          color='g', linewidth=3, label='Classification Score'
          ' (pvalue %s)' % pvalue)
pl.vlines(1.0 / n_classes, ylim[0], ylim[1], linestyle='--',
          color='k', linewidth=3, label='Luck')
pl.ylim(ylim)
pl.legend()
pl.xlabel('Score')
pl.show()
```

Precision-Recall

Example of Precision-Recall metric to evaluate the quality of the output of a classifier.



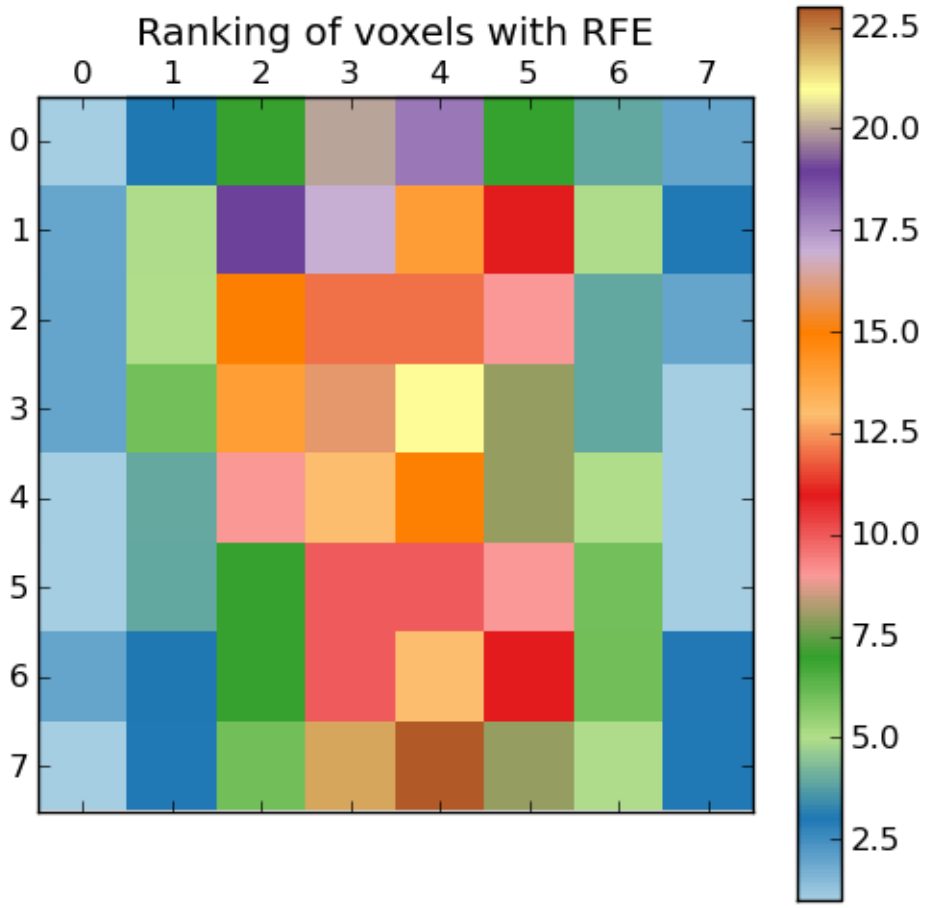
Python source code: `plot_precision_recall.py`

```
print __doc__

import random
import pylab as pl
import numpy as np
from scikits.learn import svm, datasets
from scikits.learn.metrics import precision_recall_curve
from scikits.learn.metrics import auc

# import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target
X, y = X[y!=2], y[y!=2] # Keep also 2 classes (0 and 1)
n_samples, n_features = X.shape
p = range(n_samples) # Shuffle samples
random.seed(0)
random.shuffle(p)
X, y = X[p], y[p]
half = int(n_samples/2)

# Add noisy features
np.random.seed(0)
X = np.c_[X,np.random.randn(n_samples, 200 * n_features)]
```

```
# Run classifier
classifier = svm.SVC(kernel='linear', probability=True)
probas_ = classifier.fit(X[:half], y[:half]).predict_proba(X[half:])

# Compute Precision-Recall and plot curve
precision, recall, thresholds = precision_recall_curve(y[half:], probas_[:,1])
area = auc(recall, precision)
print "Area Under Curve: %0.2f" % area

pl.figure(-1)
pl.clf()
pl.plot(recall, precision, label='Precision-Recall curve')
pl.xlabel('Recall')
pl.ylabel('Precision')
pl.ylim([0.0,1.05])
pl.xlim([0.0,1.0])
pl.title('Precision-Recall example: AUC=%0.2f' % area)
pl.legend(loc="lower left")
pl.show()
```

Recursive feature elimination

A recursive feature elimination is performed prior to SVM classification.



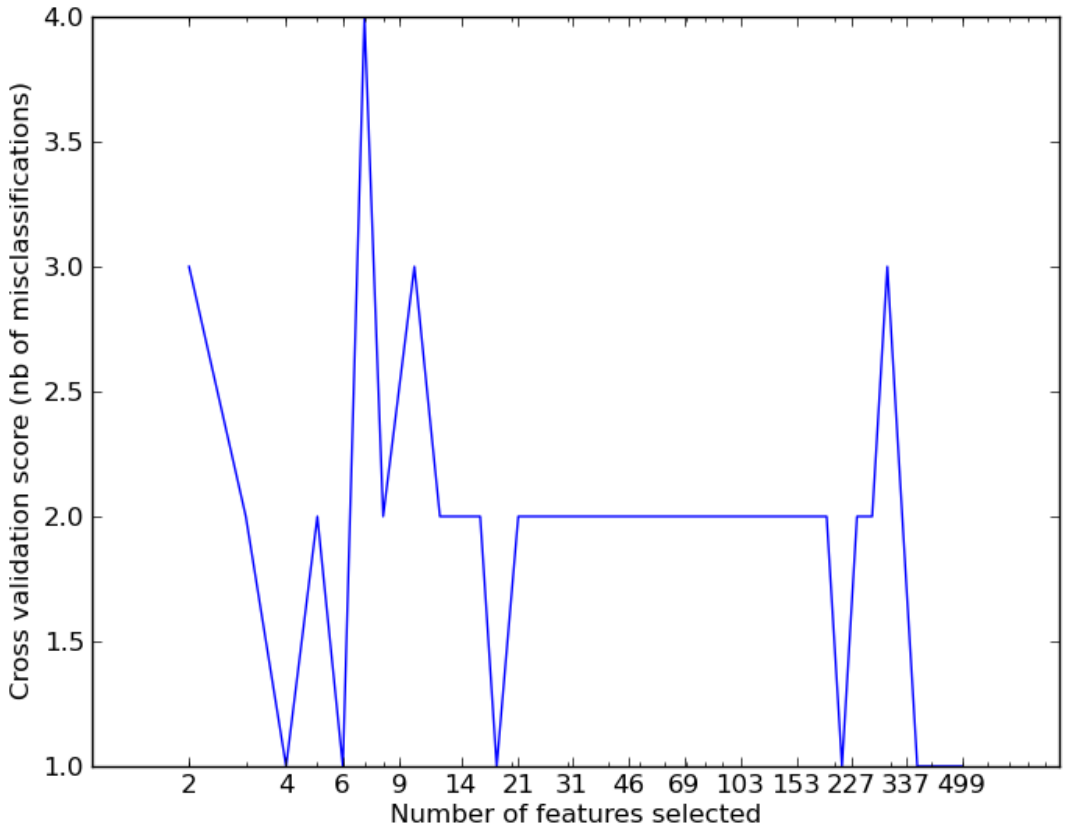
Python source code: `plot_rfe_digits.py`

```
print __doc__

from scikits.learn.svm import SVC
from scikits.learn import datasets
from scikits.learn.feature_selection import RFE

###############################################################################
# Loading the Digits dataset
digits = datasets.load_digits()

# To apply an classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
X = digits.images.reshape((n_samples, -1))
y = digits.target
```

```
###############################################################################
# Create the RFE object and compute a cross-validated score

svc = SVC(kernel="linear", C=1)
rfe = RFE(estimator=svc, n_features=1, percentage=0.1)
rfe.fit(X, y)

image_ranking_ = rfe.ranking_.reshape(digits.images[0].shape)

import pylab as pl
pl.matshow(image_ranking_)
pl.colorbar()
pl.title('Ranking of voxels with RFE')
pl.show()
```

Recursive feature elimination with cross-validation

Recursive feature elimination with automatic tuning of the number of features selected with cross-validation



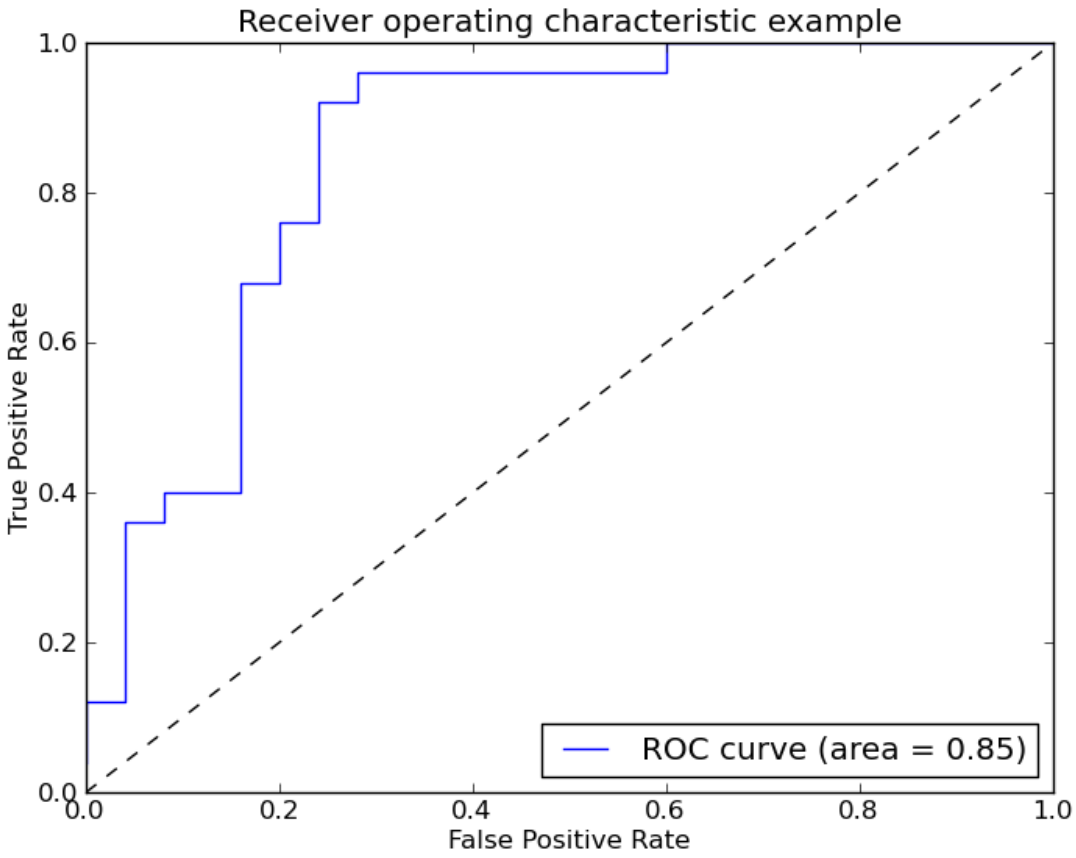
Python source code: `plot_rfe_with_cross_validation.py`

```
print __doc__
import numpy as np
```

```
from scikits.learn.svm import SVC
from scikits.learn.cross_val import StratifiedKFold
from scikits.learn.feature_selection import RFECV
from scikits.learn.datasets import samples_generator
from scikits.learn.metrics import zero_one

################################################################################
# Loading a dataset

X, y = samples_generator.test_dataset_classif(n_features=500, k=5, seed=0)

################################################################################
# Create the RFE object and compute a cross-validated score

svc = SVC(kernel='linear')
rfecv = RFECV(estimator=svc, n_features=2, percentage=0.1, loss_func=zero_one)
rfecv.fit(X, y, cv=StratifiedKFold(y, 2))

print 'Optimal number of features : %d' % rfecv.support_.sum()

import pylab as pl
pl.figure()
pl.semilogx(rfecv.n_features_, rfecv.cv_scores_)
pl.xlabel('Number of features selected')
pl.ylabel('Cross validation score (nb of misclassifications)')
# 15 ticks regularly-space in log
x_ticks = np.unique(np.logspace(np.log10(2),
                                np.log10(rfecv.n_features_.max()),
                                15,
                    ).astype(np.int))
pl.xticks(x_ticks, x_ticks)
pl.show()
```

Receiver operating characteristic (ROC)

Example of Receiver operating characteristic (ROC) metric to evaluate the quality of the output of a classifier.



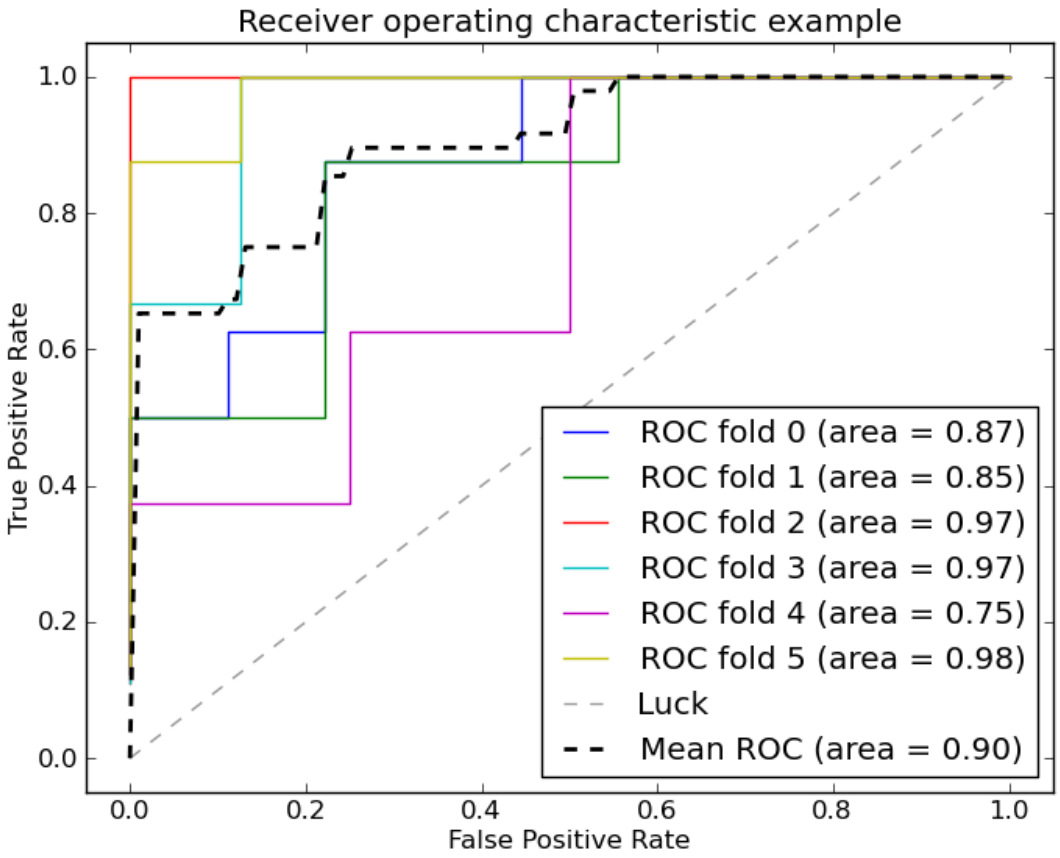
Python source code: `plot_roc.py`

```
print __doc__

import random
import numpy as np
import pylab as pl
from scikits.learn import svm, datasets
from scikits.learn.metrics import roc_curve, auc

# import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target
X, y = X[y!=2], y[y!=2]
n_samples, n_features = X.shape
p = range(n_samples)
random.seed(0)
random.shuffle(p)
X, y = X[p], y[p]
half = int(n_samples/2)

# Add noisy features
X = np.c_[X,np.random.randn(n_samples, 200*n_features)]

# Run classifier
```

```
classifier = svm.SVC(kernel='linear', probability=True)
probas_ = classifier.fit(X[:half],y[:half]).predict_proba(X[half:])

# Compute ROC curve and area the curve
fpr, tpr, thresholds = roc_curve(y[half:], probas_[:,1])
roc_auc = auc(fpr, tpr)
print "Area under the ROC curve : %f" % roc_auc

# Plot ROC curve
pl.figure(-1)
pl.clf()
pl.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
pl.plot([0, 1], [0, 1], 'k--')
pl.xlim([0.0,1.0])
pl.ylim([0.0,1.0])
pl.xlabel('False Positive Rate')
pl.ylabel('True Positive Rate')
pl.title('Receiver operating characteristic example')
pl.legend(loc="lower right")
pl.show()
```

Receiver operating characteristic (ROC) with cross validation

Example of Receiver operating characteristic (ROC) metric to evaluate the quality of the output of a classifier using cross-validation.



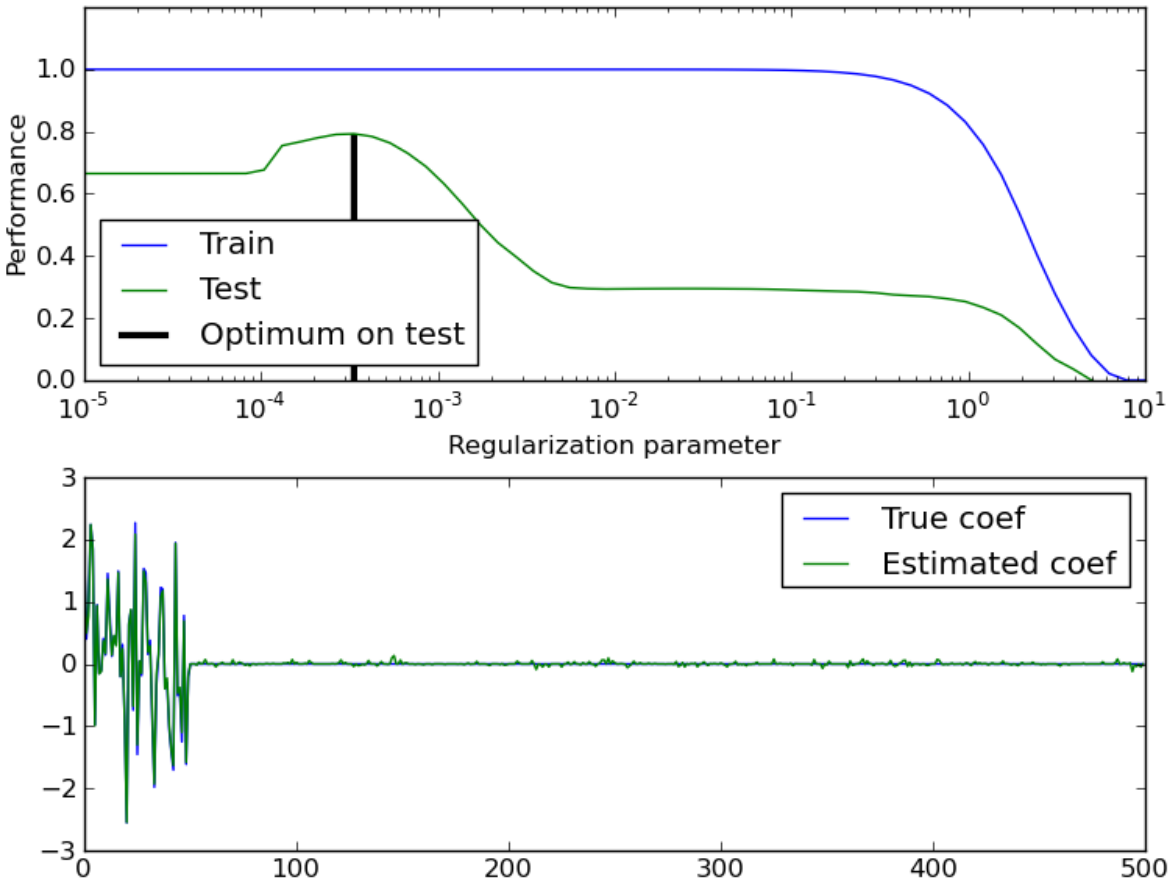
Python source code: `plot_roc_crossval.py`

```
print __doc__

import numpy as np
from scipy import interp
import pylab as pl

from scikits.learn import svm, datasets
from scikits.learn.metrics import roc_curve, auc
from scikits.learn.cross_val import StratifiedKFold

###############################################################################
# Data IO and generation

# import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target
X, y = X[y!=2], y[y!=2]
n_samples, n_features = X.shape

# Add noisy features
X = np.c_[X,np.random.randn(n_samples, 200*n_features)]

###############################################################################
```

```
# Classification and ROC analysis

# Run classifier with crossvalidation and plot ROC curves
cv = StratifiedKFold(y, k=6)
classifier = svm.SVC(kernel='linear', probability=True)

mean_tpr = 0.0
mean_fpr = np.linspace(0, 1, 100)
all_tpr = []

for i, (train, test) in enumerate(cv):
    probas_ = classifier.fit(X[train], y[train]).predict_proba(X[test])
    # Compute ROC curve and area the curve
    fpr, tpr, thresholds = roc_curve(y[test], probas_[:,1])
    mean_tpr += interp(mean_fpr, fpr, tpr)
    mean_tpr[0] = 0.0
    roc_auc = auc(fpr, tpr)
    pl.plot(fpr, tpr, lw=1, label='ROC fold %d (area = %0.2f)' % (i, roc_auc))

pl.plot([0, 1], [0, 1], '--', color=(0.6,0.6,0.6), label='Luck')

mean_tpr /= len(cv)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
pl.plot(mean_fpr, mean_tpr, 'k--',
        label='Mean ROC (area = %0.2f)' % mean_auc, lw=2)

pl.xlim([-0.05,1.05])
pl.ylim([-0.05,1.05])
pl.xlabel('False Positive Rate')
pl.ylabel('True Positive Rate')
pl.title('Receiver operating characteristic example')
pl.legend(loc="lower right")
pl.show()
```

Train error vs Test error

Illustration of how the performance of an estimator on unseen data (test data) is not the same as the performance on training data. As the regularization increases the performance on train decreases while the performance on test is optimal within a range of values of the regularization parameter. The example with an Elastic-Net regression model and the performance is measured using the explained variance a.k.a. R^2.



Python source code: `plot_train_error_vs_test_error.py`

```
print __doc__

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD Style.

import numpy as np
from scikits.learn import linear_model

##############################################################################
# Generate sample data
n_samples_train, n_samples_test, n_features = 75, 150, 500
np.random.seed(0)
coef = np.random.randn(n_features)
coef[50:] = 0.0 # only the top 10 features are impacting the model
X = np.random.randn(n_samples_train + n_samples_test, n_features)
y = np.dot(X, coef)

# Split train and test data
X_train, X_test = X[:n_samples_train], X[n_samples_train:]
y_train, y_test = y[:n_samples_train], y[n_samples_train:]

##############################################################################
# Compute train and test errors
alphas = np.logspace(-5, 1, 60)
```

```
enet = linear_model.ElasticNet(rho=0.7)
train_errors = list()
test_errors = list()
for alpha in alphas:
    enet.fit(X_train, y_train, alpha=alpha)
    train_errors.append(enet.score(X_train, y_train))
    test_errors.append(enet.score(X_test, y_test))

i_alpha_optim = np.argmax(test_errors)
alpha_optim = alphas[i_alpha_optim]
print "Optimal regularization parameter : %s" % alpha_optim

# Estimate the coef_ on full data with optimal regularization parameter
coef_ = enet.fit(X, y, alpha=alpha_optim).coef_

################################################################################
# Plot results functions

import pylab as pl
pl.subplot(2, 1, 1)
pl.semilogx(alphas, train_errors, label='Train')
pl.semilogx(alphas, test_errors, label='Test')
pl.vlines(alpha_optim, pl.ylim()[0], np.max(test_errors),
                        color='k', linewidth=3, label='Optimum on test')
pl.legend(loc='lower left')
pl.ylim([0, 1.2])
pl.xlabel('Regularization parameter')
pl.ylabel('Performance')

# Show estimated coef_ vs true coef
pl.subplot(2, 1, 2)
pl.plot(coef, label='True coef')
pl.plot(coef_, label='Estimated coef')
pl.legend()
pl.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.26)
pl.show()
```

2.1.2 Examples based on real world datasets

Applications to real world problems with some medium sized datasets or interactive user interface.

Faces recognition example using eigenfaces and SVMs

The dataset used in this example is a preprocessed excerpt of the "Labeled Faces in the Wild", aka LFW:

http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz (233MB)

Expected results for the top 5 most represented people in the dataset:

```
                   precision    recall  f1-score   support

Gerhard_Schroeder       0.91      0.75      0.82        28
  Donald_Rumsfeld       0.84      0.82      0.83        33
       Tony_Blair       0.65      0.82      0.73        34
     Colin_Powell       0.78      0.88      0.83        58
    George_W_Bush       0.93      0.86      0.90       129
```

```
avg / total       0.86      0.84      0.85       282
```

Python source code: `plot_face_recognition.py`

```
print __doc__

import os
from gzip import GzipFile

import numpy as np
import pylab as pl

from scikits.learn.grid_search import GridSearchCV
from scikits.learn.metrics import classification_report
from scikits.learn.metrics import confusion_matrix
from scikits.learn.pca import RandomizedPCA
from scikits.learn.svm import SVC

###############################################################################
# Download the data, if not already on disk
```

```
url = "https://downloads.sourceforge.net/project/scikit-learn/data/lfw_preprocessed.tar.gz"
archive_name = "lfw_preprocessed.tar.gz"
folder_name = "lfw_preprocessed"

if not os.path.exists(folder_name):
    if not os.path.exists(archive_name):
        import urllib
        print "Downloading data, please Wait (58.8MB)..."
        print url
        opener = urllib.urlopen(url)
        open(archive_name, 'wb').write(opener.read())
        print

    import tarfile
    print "Decompressiong the archive: " + archive_name
    tarfile.open(archive_name, "r:gz").extractall()
    print

###############################################################################
# Load dataset in memory

faces_filename = os.path.join(folder_name, "faces.npy.gz")
filenames_filename = os.path.join(folder_name, "face_filenames.txt")

faces = np.load(GzipFile(faces_filename))
face_filenames = [l.strip() for l in file(filenames_filename).readlines()]

# normalize each picture by centering brightness
faces -= faces.mean(axis=1)[:, np.newaxis]


###############################################################################
# Index category names into integers suitable for scikit-learn

# Here we do a little dance to convert file names in integer indices
# (class indices in machine learning talk) that are suitable to be used
# as a target for training a classifier. Note the use of an array with
# unique entries to store the relation between class index and name,
# often called a 'Look Up Table' (LUT).
# Also, note the use of 'searchsorted' to convert an array in a set of
# integers given a second array to use as a LUT.
categories = np.array([f.rsplit('_', 1)[0] for f in face_filenames])

# A unique integer per category
category_names = np.unique(categories)

# Turn the categories in their corresponding integer label
target = np.searchsorted(category_names, categories)

# Subsample the dataset to restrict to the most frequent categories
selected_target = np.argsort(np.bincount(target))[-5:]

# If you are using a numpy version >= 1.4, this can be done with 'np.in1d'
mask = np.array([item in selected_target for item in target])

X = faces[mask]
y = target[mask]
```

```
n_samples, n_features = X.shape

print "Dataset size:"
print "n_samples: %d" % n_samples
print "n_features: %d" % n_features

split = n_samples * 3 / 4

X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

###############################################################################
# Compute a PCA (eigenfaces) on the face dataset (treated as unlabeled
# dataset): unsupervised feature extraction / dimensionality reduction
n_components = 150

print "Extracting the top %d eigenfaces" % n_components
pca = RandomizedPCA(n_components=n_components, whiten=True).fit(X_train)

eigenfaces = pca.components_.T.reshape((n_components, 64, 64))

# project the input data on the eigenfaces orthonormal basis
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)


###############################################################################
# Train a SVM classification model

print "Fitting the classifier to the training set"
param_grid = {
 'C': [1, 5, 10, 50, 100],
 'gamma': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1],
}
clf = GridSearchCV(SVC(kernel='rbf'), param_grid,
                   fit_params={'class_weight': 'auto'})
clf = clf.fit(X_train_pca, y_train)
print "Best estimator found by grid search:"
print clf.best_estimator


###############################################################################
# Quantitative evaluation of the model quality on the test set

y_pred = clf.predict(X_test_pca)
print classification_report(y_test, y_pred, labels=selected_target,
                            class_names=category_names[selected_target])

print confusion_matrix(y_test, y_pred, labels=selected_target)


###############################################################################
# Qualitative evaluation of the predictions using matplotlib

n_row = 3
n_col = 4

pl.figure(figsize=(2 * n_col, 2.3 * n_row))
```

```
pl.subplots_adjust(bottom=0, left=.01, right=.99, top=.95, hspace=.15)
for i in range(n_row * n_col):
    pl.subplot(n_row, n_col, i + 1)
    pl.imshow(X_test[i].reshape((64, 64)), cmap=pl.cm.gray)
    pl.title('pred: %s\ntrue: %s' % (category_names[y_pred[i]],
                                     category_names[y_test[i]]), size=12)
    pl.xticks(())
    pl.yticks(())

pl.show()

# TODO: plot the top eigenfaces and the singular values absolute values
```

Species distribution modeling

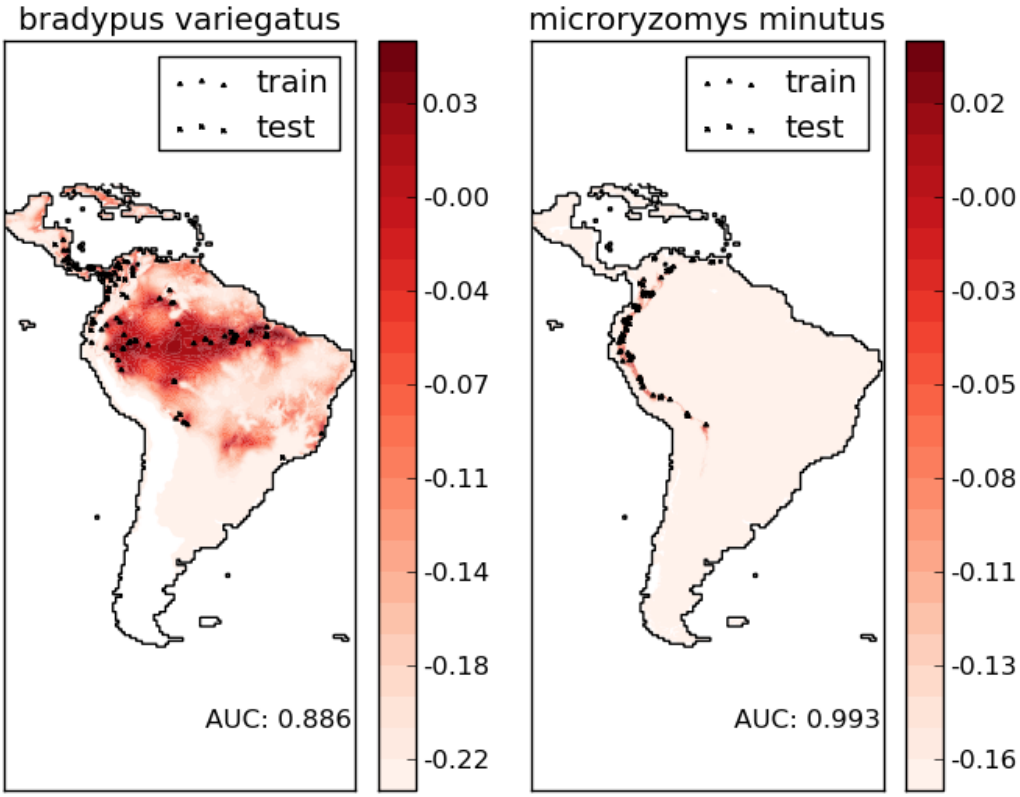
Modeling species' geographic distributions is an important problem in conservation biology. In this example we model the geographic distribution of two south american mammals given past observations and 14 environmental variables. Since we have only positive examples (there are no unsuccessful observations), we cast this problem as a density estimation problem and use the *OneClassSVM* provided by the package *scikits.learn.svm* as our modeling tool. The dataset is provided by Phillips et. al. (2006). If available, the example uses basemap to plot the coast lines and national boundaries of South America.

The two species are:

- Bradypus variegatus , the Brown-throated Sloth.
- Microryzomys minutus , also known as the Forest Small Rice Rat, a rodent that lives in Peru, Colombia, Ecuador, Peru, and Venezuela.

References:

- "Maximum entropy modeling of species geographic distributions" S. J. Phillips, R. P. Anderson, R. E. Schapire - Ecological Modelling, 190:231-259, 2006.



Python source code: `plot_species_distribution_modeling.py`

```
from __future__ import division

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#
# License: Simplified BSD

print __doc__

import pylab as pl
import numpy as np

try:
    from mpl_toolkits.basemap import Basemap
    basemap = True
except ImportError:
    basemap = False

from time import time
from os.path import normpath, split, exists
from glob import glob
from scikits.learn import svm
from scikits.learn.metrics import roc_curve, auc
from scikits.learn.datasets.base import Bunch
```

```
###############################################################################
# Download the data, if not already on disk
samples_url = "http://www.cs.princeton.edu/~schapire/maxent/datasets/" \
              "samples.zip"
coverage_url = "http://www.cs.princeton.edu/~schapire/maxent/datasets/" \
               "coverages.zip"
samples_archive_name = "samples.zip"
coverage_archive_name = "coverages.zip"


def download(url, archive_name):
    if not exists(archive_name[:-4]):
        if not exists(archive_name):
            import urllib
            print "Downloading data, please wait ..."
            print url
            opener = urllib.urlopen(url)
            open(archive_name, 'wb').write(opener.read())
            print

        import zipfile
        print "Decompressiong the archive: " + archive_name
        zipfile.ZipFile(archive_name).extractall()
        print


download(samples_url, samples_archive_name)
download(coverage_url, coverage_archive_name)

t0 = time()



###############################################################################
# Preprocess data

species = ["bradypus_variegatus_0", "microryzomys_minutus_0"]
species_map = dict([(s, i) for i, s in enumerate(species)])

# x,y coordinates of study area
x_left_lower_corner = -94.8
y_left_lower_corner = -56.05
n_cols = 1212
n_rows = 1592
grid_size = 0.05  # ~5.5 km

# x,y coordinates for each cell
xmin = x_left_lower_corner + grid_size
xmax = xmin + (n_cols * grid_size)
ymin = y_left_lower_corner + grid_size
ymax = ymin + (n_rows * grid_size)

# x coordinates of the grid cells
xx = np.arange(xmin, xmax, grid_size)
# y coordinates of the grid cells
yy = np.arange(ymin, ymax, grid_size)

print "Data grid"
```

```
print "---------"
print "xmin, xmax:", xmin, xmax
print "ymin, ymax:", ymin, ymax
print "grid size:", grid_size
print

##############################################################################
# Load data

print "loading data from disk..."
def read_file(fname):
    """Read coverage grid data; returns array of
    shape [n_rows, n_cols]. """
    f = open(fname)
    # Skip header
    for i in range(6):
        f.readline()
    X = np.fromfile(f, dtype=np.float32, sep=" ", count=-1)
    f.close()
    return X.reshape((n_rows, n_cols))

def load_dir(directory):
    """Loads each of the coverage grids and returns a
    tensor of shape [14, n_rows, n_cols].
    """
    data = []
    for fpath in glob("%s/*.asc" % normpath(directory)):
        fname = split(fpath)[-1]
        fname = fname[:fname.index(".")]
        X = read_file(fpath)  #np.loadtxt(fpath, skiprows=6, dtype=np.float32)
        data.append(X)
    return np.array(data, dtype=np.float32)

def get_coverages(points, coverages, xx, yy):
    """
    Returns
    -------
    array : shape = [n_points, 14]
    """
    rows = []
    cols = []
    for n in range(points.shape[0]):
        i = np.searchsorted(xx, points[n, 0])
        j = np.searchsorted(yy, points[n, 1])
        rows.append(-j)
        cols.append(i)
    return coverages[:, rows, cols].T

species2id = lambda s: species_map.get(s, -1)
train = np.loadtxt('samples/alltrain.csv', converters={0: species2id},
                   skiprows=1, delimiter=",")
test = np.loadtxt('samples/alltest.csv', converters={0: species2id},
                  skiprows=1, delimiter=",")
# Load env variable grids
coverage = load_dir("coverages")

# Per species data
bv = Bunch(name=" ".join(species[0].split("_")[:2]),
```

```
            train=train[train[:,0] == 0, 1:],
            test=test[test[:,0] == 0, 1:])
mm = Bunch(name=" ".join(species[1].split("_")[:2]),
            train=train[train[:,0] == 1, 1:],
            test=test[test[:,0] == 1, 1:])

# Get features (=coverages)
bv.train_cover = get_coverages(bv.train, coverage, xx, yy)
bv.test_cover = get_coverages(bv.test, coverage, xx, yy)
mm.train_cover = get_coverages(mm.train, coverage, xx, yy)
mm.test_cover = get_coverages(mm.test, coverage, xx, yy)


def predict(clf, mean, std):
    """Predict the density of the land grid cells
    under the model 'clf'.

    Returns
    -------
    array : shape [n_rows, n_cols]
    """
    Z = np.ones((n_rows, n_cols), dtype=np.float64)
    # the land points
    idx = np.where(coverage[2] > -9999)
    X = coverage[:, idx[0], idx[1]].T
    pred = clf.decision_function((X-mean)/std)[:,0]
    Z *= pred.min()
    Z[idx[0], idx[1]] = pred
    return Z

# background points (grid coordinates) for evaluation
np.random.seed(13)
background_points = np.c_[np.random.randint(low=0, high=n_rows, size=10000),
                          np.random.randint(low=0, high=n_cols, size=10000)].T

# The grid in x,y coordinates
X, Y = np.meshgrid(xx, yy[::-1])
#basemap = False
for i, species in enumerate([bv, mm]):
    print "_" * 80
    print "Modeling distribution of species '%s'" % species.name
    print
    # Standardize features
    mean = species.train_cover.mean(axis=0)
    std = species.train_cover.std(axis=0)
    train_cover_std = (species.train_cover - mean) / std

    # Fit OneClassSVM
    print "fit OneClassSVM ... ",
    clf = svm.OneClassSVM(nu=0.1, kernel="rbf", gamma=0.5)
    clf.fit(train_cover_std)
    print "done. "

    # Plot map of South America
    pl.subplot(1, 2, i + 1)
    if basemap:
        print "plot coastlines using basemap"
        m = Basemap(projection='cyl', llcrnrlat=ymin,
```

```
                urcrnrlat=ymax, llcrnrlon=xmin,
                urcrnrlon=xmax, resolution='c')
        m.drawcoastlines()
        m.drawcountries()
        #m.drawrivers()
    else:
        print "plot coastlines from coverage"
        CS = pl.contour(X, Y, coverage[2,:,:], levels=[-9999], colors="k",
                        linestyles="solid")
        pl.xticks([])
        pl.yticks([])

    print "predict species distribution"
    Z = predict(clf, mean, std)
    levels = np.linspace(Z.min(), Z.max(), 25)
    Z[coverage[2,:,:] == -9999] = -9999
    CS = pl.contourf(X, Y, Z, levels=levels, cmap=pl.cm.Reds)
    pl.colorbar(format='%.2f')
    pl.scatter(species.train[:, 0], species.train[:, 1], s=2**2, c='black',
               marker='^', label='train')
    pl.scatter(species.test[:, 0], species.test[:, 1], s=2**2, c='black',
               marker='x', label='test')
    pl.legend()
    pl.title(species.name)
    pl.axis('equal')

    # Compute AUC w.r.t. background points
    pred_background = Z[background_points[0], background_points[1]]
    pred_test = clf.decision_function((species.test_cover-mean)/std)[:,0]
    scores = np.r_[pred_test, pred_background]
    y = np.r_[np.ones(pred_test.shape), np.zeros(pred_background.shape)]
    fpr, tpr, thresholds = roc_curve(y, scores)
    roc_auc = auc(fpr, tpr)
    pl.text(-35, -70, "AUC: %.3f" % roc_auc, ha="right")
    print "Area under the ROC curve : %f" % roc_auc

print "time elapsed: %.3fs" % (time() - t0)

pl.show()
```

Finding structure in the stock market

An example of playing with stock market data to try and find some structure in it.

Python source code: `stock_market.py`

```
print __doc__

# Author: Gael Varoquaux gael.varoquaux@normalesup.org
# License: BSD

import datetime
from matplotlib import finance
import numpy as np

from scikits.learn import cluster
```

```
# Choose a time period reasonnably calm (not too long ago so that we get
# high-tech firms, and before the 2008 crash)
d1 = datetime.datetime(2003, 01, 01)
d2 = datetime.datetime(2008, 01, 01)

symbol_dict = {
        'TOT'  : 'Total',
        'XOM'  : 'Exxon',
        'CVX'  : 'Chevron',
        'COP'  : 'ConocoPhillips',
        'VLO'  : 'Valero Energy',
        'MSFT' : 'Microsoft',
        'IBM'  : 'IBM',
        'TWX'  : 'Time Warner',
        'CMCSA': 'Comcast',
        'CVC'  : 'Cablevision',
        'YHOO' : 'Yahoo',
        'DELL' : 'Dell',
        'HPQ'  : 'Hewlett-Packard',
        'AMZN' : 'Amazon',
        'TM'   : 'Toyota',
        'CAJ'  : 'Canon',
        'MTU'  : 'Mitsubishi',
        'SNE'  : 'Sony',
        'F'    : 'Ford',
        'HMC'  : 'Honda',
        'NAV'  : 'Navistar',
        'NOC'  : 'Northrop Grumman',
        'BA'   : 'Boeing',
        'KO'   : 'Coca Cola',
        'MMM'  : '3M',
        'MCD'  : 'Mc Donalds',
        'PEP'  : 'Pepsi',
        'KFT'  : 'Kraft Foods',
        'K'    : 'Kellogg',
        'UN'   : 'Unilever',
        'MAR'  : 'Marriott',
        'PG'   : 'Procter Gamble',
        'CL'   : 'Colgate-Palmolive',
        'NWS'  : 'News Corporation',
        'GE'   : 'General Electrics',
        'WFC'  : 'Wells Fargo',
        'JPM'  : 'JPMorgan Chase',
        'AIG'  : 'AIG',
        'AXP'  : 'American express',
        'BAC'  : 'Bank of America',
        'GS'   : 'Goldman Sachs',
        'AAPL' : 'Apple',
        'SAP'  : 'SAP',
        'CSCO' : 'Cisco',
        'TXN'  : 'Texas instruments',
        'XRX'  : 'Xerox',
        'LMT'  : 'Lookheed Martin',
        'WMT'  : 'Wal-Mart',
        'WAG'  : 'Walgreen',
        'HD'   : 'Home Depot',
        'GSK'  : 'GlaxoSmithKline',
        'PFE'  : 'Pfizer',
```

```
        'SNY'   : 'Sanofi-Aventis',
        'NVS'   : 'Novartis',
        'KMB'   : 'Kimberly-Clark',
        'R'     : 'Ryder',
        'GD'    : 'General Dynamics',
        'RTN'   : 'Raytheon',
        'CVS'   : 'CVS',
        'CAT'   : 'Caterpillar',
        'DD'    : 'DuPont de Nemours',
    }

symbols, names = np.array(symbol_dict.items()).T

quotes = [finance.quotes_historical_yahoo(symbol, d1, d2, asobject=True)
                for symbol in symbols]

#volumes = np.array([q.volume for q in quotes]).astype(np.float)
open    = np.array([q.open   for q in quotes]).astype(np.float)
close   = np.array([q.close  for q in quotes]).astype(np.float)
variation = close - open
correlations = np.corrcoef(variation)

_, labels = cluster.affinity_propagation(correlations)

for i in range(labels.max()+1):
    print 'Cluster %i: %s' % ((i+1),
                              ', '.join(names[labels==i]))
```

Libsvm GUI

A simple graphical frontend for Libsvm mainly intended for didactic purposes. You can create data points by point and click and visualize the decision region induced by different kernels and parameter settings.

To create positive examples click the left mouse button; to create negative examples click the right button.

If all examples are from the same class, it uses a one-class SVM.

TODO add labels to the panel.

Requirements

- Tkinter
- scikits.learn
- matplotlib with TkAgg

Python source code: `svm_gui.py`

```
from __future__ import division

print __doc__

# Author: Peter Prettenhoer <peter.prettenhofer@gmail.com>
#
# License: BSD Style.
```

```
import matplotlib
matplotlib.use('TkAgg')

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
#from matplotlib.backends.backend_tkagg import NavigationToolbar2TkAgg
from matplotlib.figure import Figure
from matplotlib.contour import ContourSet

import Tkinter as Tk
import sys
import numpy as np

from scikits.learn import svm

y_min, y_max = -50, 50
x_min, x_max = -50, 50


class Model(object):
    """The Model which hold the data. It implements the
    observable in the observer pattern and notifies the
    registered observers on change event.
    """

    def __init__(self):
        self.observers = []
        self.surface = None
        self.data = []
        self.cls = None
        self.surface_type = 0

    def changed(self, event):
        """Notify the observers. """
        for observer in self.observers:
            observer.update(event, self)

    def add_observer(self, observer):
        """Register an observer. """
        self.observers.append(observer)

    def set_surface(self, surface):
        self.surface = surface


class Controller(object):
    def __init__(self, model):
        self.model = model
        self.kernel = Tk.IntVar()
        self.surface_type = Tk.IntVar()
        # Whether or not a model has been fitted
        self.fitted = False

    def fit(self):
        print "fit the model"
        train = np.array(self.model.data)
        X = train[:, :2]
        y = train[:, 2]
```

```
        C = float(self.complexity.get())
        gamma = float(self.gamma.get())
        coef0 = float(self.coef0.get())
        degree = int(self.degree.get())
        kernel_map = {0: "linear", 1: "rbf", 2: "poly"}
        if len(np.unique(y)) == 1:
            clf = svm.OneClassSVM(kernel=kernel_map[self.kernel.get()],
                      gamma=gamma, coef0=coef0, degree=degree)
            clf.fit(X)
        else:
            clf = svm.SVC(kernel=kernel_map[self.kernel.get()], C=C,
                          gamma=gamma, coef0=coef0, degree=degree)
            clf.fit(X, y)
        if hasattr(clf, 'score'):
            print "Accuracy:", clf.score(X, y) * 100
        X1, X2, Z = self.decision_surface(clf)
        self.model.clf = clf
        self.model.set_surface((X1, X2, Z))
        self.model.surface_type = self.surface_type.get()
        self.fitted = True
        self.model.changed("surface")

    def decision_surface(self, cls):
        delta = 1
        x = np.arange(x_min, x_max + delta, delta)
        y = np.arange(y_min, y_max + delta, delta)
        X1, X2 = np.meshgrid(x, y)
        Z = cls.decision_function(np.c_[X1.ravel(), X2.ravel()])
        Z = Z.reshape(X1.shape)
        return X1, X2, Z

    def clear_data(self):
        self.model.data = []
        self.fitted = False
        self.model.changed("clear")

    def add_example(self, x, y, label):
        self.model.data.append((x, y, label))
        self.model.changed("example_added")

        # update decision surface if already fitted.
        self.refit()

    def refit(self):
        """Refit the model if already fitted. """
        if self.fitted:
            self.fit()


class View(object):
    """Test docstring. """
    def __init__(self, root, controller):
        f = Figure()
        ax = f.add_subplot(111)
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_xlim((x_min, x_max))
        ax.set_ylim((y_min, y_max))
```

```
        canvas = FigureCanvasTkAgg(f, master=root)
        canvas.show()
        canvas.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)
        canvas._tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)
        canvas.mpl_connect('button_press_event', self.onclick)
#        toolbar = NavigationToolbar2TkAgg(canvas, root)
#        toolbar.update()
        self.controllbar = ControllBar(root, controller)
        self.f = f
        self.ax = ax
        self.canvas = canvas
        self.controller = controller
        self.contours = []
        self.c_labels = None
        self.plot_kernels()

    def plot_kernels(self):
        self.ax.text(-50, -60, "Linear: $u^T v$")
        self.ax.text(-20, -60, "RBF: $\exp (-\gamma \| u-v \|^2)$")
        self.ax.text(10, -60, "Poly: $(\gamma \, u^T v + r)^d$")

    def onclick(self, event):
        if event.xdata and event.ydata:
            if event.button == 1:
                self.controller.add_example(event.xdata, event.ydata, 1)
            elif event.button == 3:
                self.controller.add_example(event.xdata, event.ydata, -1)

    def update(self, event, model):
        if event == "example_added":
            x, y, l = model.data[-1]
            if l == 1:
                color = 'w'
            elif l == -1:
                color = 'k'
            self.ax.plot([x], [y], "%so" % color, scalex=0.0, scaley=0.0)

        if event == "clear":
            self.ax.clear()
            self.ax.set_xticks([])
            self.ax.set_yticks([])
            self.contours = []
            self.c_labels = None
            self.plot_kernels()

        if event == "surface":
            self.remove_surface()
            self.plot_support_vectors(model.clf.support_vectors_)
            self.plot_decision_surface(model.surface, model.surface_type)

        self.canvas.draw()

    def remove_surface(self):
        """Remove old decision surface."""
        if len(self.contours) > 0:
            for contour in self.contours:
                if isinstance(contour, ContourSet):
                    for lineset in contour.collections:
```

```
                    lineset.remove()
                else:
                    contour.remove()
            self.contours = []

    def plot_support_vectors(self, support_vectors):
        """Plot the support vectors by placing circles over the
        corresponding data points and adds the circle collection
        to the contours list."""
        cs = self.ax.scatter(support_vectors[:, 0], support_vectors[:, 1],
                             s=80, edgecolors="k", facecolors="none")
        self.contours.append(cs)

    def plot_decision_surface(self, surface, type):
        X1, X2, Z = surface
        if type == 0:
            levels = [-1.0, 0.0, 1.0]
            linestyles = ['dashed', 'solid', 'dashed']
            colors = 'k'
            self.contours.append(self.ax.contour(X1, X2, Z, levels,
                                                 colors=colors,
                                                 linestyles=linestyles))
        elif type == 1:
            self.contours.append(self.ax.contourf(X1, X2, Z, 10,
                                                  cmap=matplotlib.cm.bone,
                                                  origin='lower',
                                                  alpha=0.85))
            self.contours.append(self.ax.contour(X1, X2, Z, [0.0],
                                                 colors='k',
                                                 linestyles=['solid']))
        else:
            raise ValueError("surface type unknown")


class ControllBar(object):
    def __init__(self, root, controller):
        fm = Tk.Frame(root)
        kernel_group = Tk.Frame(fm)
        Tk.Radiobutton(kernel_group, text="Linear", variable=controller.kernel,
                       value=0, command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(kernel_group, text="RBF", variable=controller.kernel,
                       value=1, command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(kernel_group, text="Poly", variable=controller.kernel,
                       value=2, command=controller.refit).pack(anchor=Tk.W)
        kernel_group.pack(side=Tk.LEFT)

        valbox = Tk.Frame(fm)
        controller.complexity = Tk.StringVar()
        controller.complexity.set("1.0")
        c = Tk.Frame(valbox)
        Tk.Label(c, text="C:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(c, width=6, textvariable=controller.complexity).pack(
            side=Tk.LEFT)
        c.pack()

        controller.gamma = Tk.StringVar()
        controller.gamma.set("0.01")
        g = Tk.Frame(valbox)
```

```
        Tk.Label(g, text="gamma:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(g, width=6, textvariable=controller.gamma).pack(side=Tk.LEFT)
        g.pack()

        controller.degree = Tk.StringVar()
        controller.degree.set("3")
        d = Tk.Frame(valbox)
        Tk.Label(d, text="degree:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(d, width=6, textvariable=controller.degree).pack(side=Tk.LEFT)
        d.pack()

        controller.coef0 = Tk.StringVar()
        controller.coef0.set("0")
        r = Tk.Frame(valbox)
        Tk.Label(r, text="coef0:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(r, width=6, textvariable=controller.coef0).pack(side=Tk.LEFT)
        r.pack()
        valbox.pack(side=Tk.LEFT)

        cmap_group = Tk.Frame(fm)
        Tk.Radiobutton(cmap_group, text="Hyperplanes",
                       variable=controller.surface_type, value=0,
                       command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(cmap_group, text="Surface",
                       variable=controller.surface_type, value=1,
                       command=controller.refit).pack(anchor=Tk.W)

        cmap_group.pack(side=Tk.LEFT)

        train_button = Tk.Button(fm, text='Fit', width=5,
                                 command=controller.fit)
        train_button.pack()
        fm.pack(side=Tk.LEFT)
        Tk.Button(fm, text='Clear', width=5,
                  command=controller.clear_data).pack(side=Tk.LEFT)


def main(argv):
    root = Tk.Tk()
    model = Model()
    controller = Controller(model)
    root.wm_title("Scikit-learn Libsvm GUI")
    view = View(root, controller)
    model.add_observer(view)
    Tk.mainloop()

if __name__ == "__main__":
    main(sys.argv)
```

Wikipedia princial eigenvector

A classical way to assert the relative importance of vertices in a graph is to compute the principal eigenvector of the adjacency matrix so as to assign to each vertex the values of the components of the first eigenvector as a centrality score:

http://en.wikipedia.org/wiki/Eigenvector_centrality

On the graph of webpages and links those values are called the PageRank scores by Google.

The goal of this example is to analyze the graph of links inside wikipedia articles to rank articles by relative importance according to this eigenvector centrality.

The traditional way to compute the principal eigenvector is to use the power iteration method:

http://en.wikipedia.org/wiki/Power_iteration

Here the computation is achieved thanks to Martinsson's Randomized SVD algoritm implemented in the scikit.

The graph data is fetched from the DBpedia dumps. DBpedia is an extraction of the latent structured data of the Wikipedia content.

Python source code: `wikipedia_principal_eigenvector.py`

```
print __doc__

# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: Simplified BSD

from bz2 import BZ2File
import os
from datetime import datetime
from pprint import pprint
from time import time

import numpy as np

from scipy import sparse

from scikits.learn.utils.extmath import fast_svd
from scikits.learn.externals.joblib import Memory


################################################################################
# Where to download the data, if not already on disk
redirects_url = "http://downloads.dbpedia.org/3.5.1/en/redirects_en.nt.bz2"
redirects_filename = redirects_url.rsplit("/", 1)[1]

page_links_url = "http://downloads.dbpedia.org/3.5.1/en/page_links_en.nt.bz2"
page_links_filename = page_links_url.rsplit("/", 1)[1]

resources = [
    (redirects_url, redirects_filename),
    (page_links_url, page_links_filename),
]

for url, filename in resources:
    if not os.path.exists(filename):
        import urllib
        print "Downloading data from '%s', please wait..." % url
        opener = urllib.urlopen(url)
        open(filename, 'wb').write(opener.read())
        print


################################################################################
# Loading the redirect files

memory = Memory(cachedir=".")
```

```
def index(redirects, index_map, k):
    """Find the index of an article name after redirect resolution"""
    k = redirects.get(k, k)
    return index_map.setdefault(k, len(index_map))


DBPEDIA_RESOURCE_PREFIX_LEN = len("http://dbpedia.org/resource/")
SHORTNAME_SLICE = slice(DBPEDIA_RESOURCE_PREFIX_LEN + 1, -1)


def short_name(nt_uri):
    """Remove the < and > URI markers and the common URI prefix"""
    return nt_uri[SHORTNAME_SLICE]


def get_redirects(redirects_filename):
    """Parse the redirections and build a transitively closed map out of it"""
    redirects = {}
    print "Parsing the NT redirect file"
    for l, line in enumerate(BZ2File(redirects_filename)):
        split = line.split()
        if len(split) != 4:
            print "ignoring malformed line: " + line
            continue
        redirects[short_name(split[0])] = short_name(split[2])
        if l % 1000000 == 0:
            print "[%s] line: %08d" % (datetime.now().isoformat(), l)

    # compute the transitive closure
    print "Computing the transitive closure of the redirect relation"
    for l, source in enumerate(redirects.keys()):
        transitive_target = None
        target = redirects[source]
        seen = set([source])
        while True:
            transitive_target = target
            target = redirects.get(target)
            if target is None or target in seen:
                break
            seen.add(target)
        redirects[source] = transitive_target
        if l % 1000000 == 0:
            print "[%s] line: %08d" % (datetime.now().isoformat(), l)

    return redirects


# disabling joblib as the pickling of large dicts seems much too slow
#@memory.cache
def get_adjacency_matrix(redirects_filename, page_links_filename, limit=None):
    """Extract the adjacency graph as a scipy sparse matrix

    Redirects are resolved first.

    Returns X, the scipy sparse adjacency matrix, redirects as python
    dict from article names to article names and index_map a python dict
    from article names to python int (article indexes).
```

```
    """

    print "Computing the redirect map"
    redirects = get_redirects(redirects_filename)

    print "Computing the integer index map"
    index_map = dict()
    links = list()
    for l, line in enumerate(BZ2File(page_links_filename)):
        split = line.split()
        if len(split) != 4:
            print "ignoring malformed line: " + line
            continue
        i = index(redirects, index_map, short_name(split[0]))
        j = index(redirects, index_map, short_name(split[2]))
        links.append((i, j))
        if l % 1000000 == 0:
            print "[%s] line: %08d" % (datetime.now().isoformat(), l)

        if limit is not None and l >= limit - 1:
            break

    print "Computing the adjacency matrix"
    X = sparse.lil_matrix((len(index_map), len(index_map)), dtype=np.float32)
    for i, j in links:
        X[i, j] = 1.0
    del links
    print "Converting to CSR representation"
    X = X.tocsr()
    print "CSR conversion done"
    return X, redirects, index_map


# stop after 5M links to make it possible to work in RAM
X, redirects, index_map = get_adjacency_matrix(
    redirects_filename, page_links_filename, limit=5000000)
names = dict((i, name) for name, i in index_map.iteritems())

print "Computing the principal singular vectors using fast_svd"
t0 = time()
U, s, V = fast_svd(X, 5, q=3)
print "done in %0.3fs" % (time() - t0)

# print the names of the wikipedia related strongest compenents of the the
# principal singular vector which should be similar to the highest eigenvector
print "Top wikipedia pages according to principal singular vectors"
pprint([names[i] for i in np.abs(U.T[0]).argsort()[-10:]])
pprint([names[i] for i in np.abs(V[0]).argsort()[-10:]])


def centrality_scores(X, alpha=0.85, max_iter=100, tol=1e-10):
    """Power iteration computation of the principal eigenvector

    This method is also known as Google PageRank and the implementation
    is based on the one from the NetworkX project (BSD licensed too)
    with copyrights by:

      Aric Hagberg <hagberg@lanl.gov>
```

```
      Dan Schult <dschult@colgate.edu>
      Pieter Swart <swart@lanl.gov>
    """
    n = X.shape[0]
    X = X.copy()
    incoming_counts = np.asarray(X.sum(axis=1)).ravel()

    print "Normalizing the graph"
    for i in incoming_counts.nonzero()[0]:
        X.data[X.indptr[i]:X.indptr[i + 1]] *= 1.0 / incoming_counts[i]
    dangle = np.asarray(np.where(X.sum(axis=1) == 0, 1.0 / n, 0)).ravel()

    scores = np.ones(n, dtype=np.float32) / n  # initial guess
    for i in range(max_iter):
        print "power iteration #%d" % i
        prev_scores = scores
        scores = (alpha * (scores * X + np.dot(dangle, prev_scores))
                  + (1 - alpha) * prev_scores.sum() / n)
        # check convergence: normalized l_inf norm
        scores_max = np.abs(scores).max()
        if scores_max == 0.0:
            scores_max = 1.0
        err = np.abs(scores - prev_scores).max() / scores_max
        print "error: %0.6f" % err
        if err < n * tol:
            return scores

    return scores

print "Computing principal eigenvector score using a power iteration method"
t0 = time()
scores = centrality_scores(X, max_iter=100, tol=1e-10)
print "done in %0.3fs" % (time() - t0)
pprint([names[i] for i in np.abs(scores).argsort()[-10:]])
```

2.1.3 Clustering

Examples concerning the *scikits.learn.cluster* package.

A demo of K-Means clustering on the handwritten digits data

Comparing various initialization strategies in terms of runtime and quality of the results.

TODO: explode the ouput of the cluster labeling and digits.target groundtruth as categorical boolean arrays of shape (n_sample, n_unique_labels) and measure the Pearson correlation as an additional measure of the clustering quality.

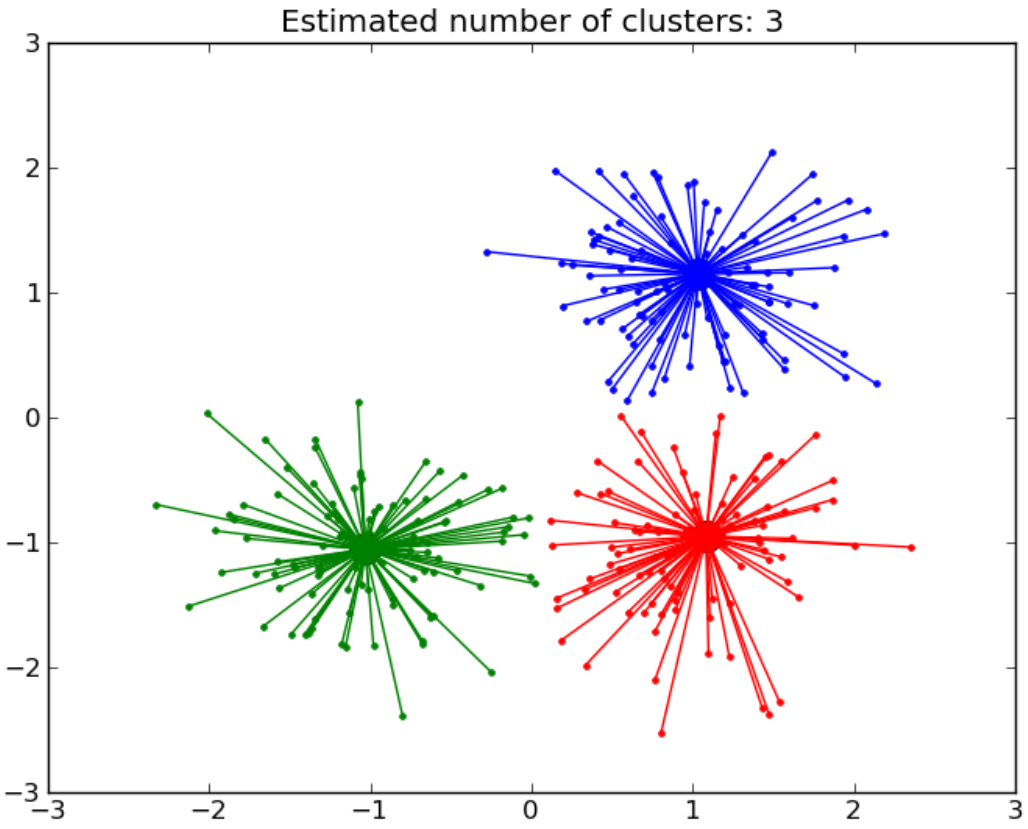
Python source code: `kmeans_digits.py`

```
print __doc__

from time import time
import numpy as np

from scikits.learn.cluster import KMeans
from scikits.learn.datasets import load_digits
from scikits.learn.pca import PCA
```

```
from scikits.learn.preprocessing import scale

np.random.seed(42)

digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))

print "n_digits: %d" % n_digits
print "n_features: %d" % n_features
print "n_samples: %d" % n_samples
print

print "Raw k-means with k-means++ init..."
t0 = time()
km = KMeans(init='k-means++', k=n_digits, n_init=10).fit(data)
print "done in %0.3fs" % (time() - t0)
print "inertia: %f" % km.inertia_
print

print "Raw k-means with random centroid init..."
t0 = time()
km = KMeans(init='random', k=n_digits, n_init=10).fit(data)
print "done in %0.3fs" % (time() - t0)
print "inertia: %f" % km.inertia_
print

print "Raw k-means with PCA-based centroid init..."
# in this case the seeding of the centers is deterministic, hence we run the
# kmeans algorithm only once with n_init=1
t0 = time()
pca = PCA(n_components=n_digits).fit(data)
km = KMeans(init=pca.components_.T, k=n_digits, n_init=1).fit(data)
print "done in %0.3fs" % (time() - t0)
print "inertia: %f" % km.inertia_
print
```

Demo of affinity propagation clustering algorithm

Reference: Brendan J. Frey and Delbert Dueck, "Clustering by Passing Messages Between Data Points", Science Feb. 2007



Python source code: `plot_affinity_propagation.py`

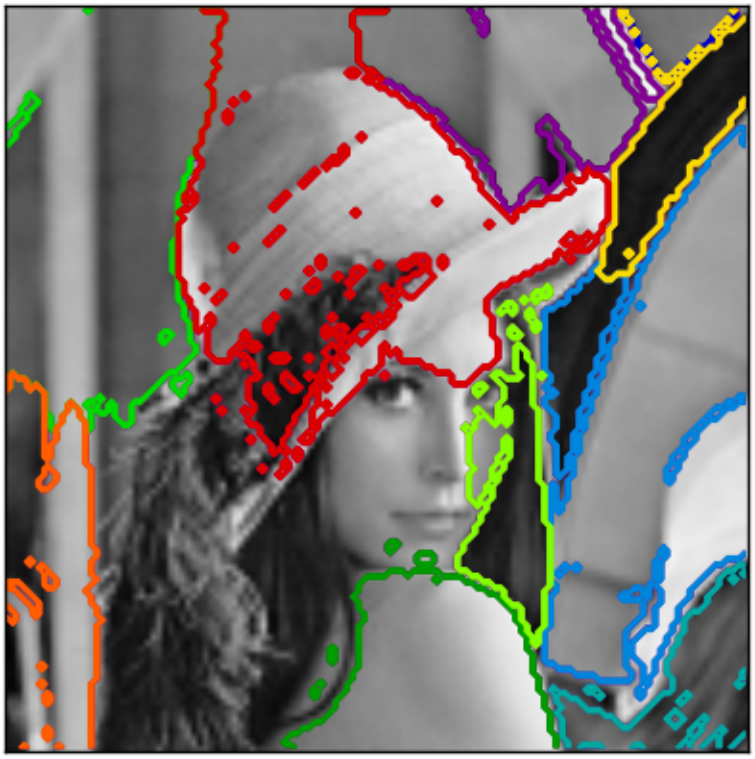
```
print __doc__

import numpy as np
from scikits.learn.cluster import AffinityPropagation

##############################################################################
# Generate sample data
##############################################################################
np.random.seed(0)

n_points_per_cluster = 100
n_clusters = 3
n_points = n_points_per_cluster*n_clusters
means = np.array([[1,1],[-1,-1],[1,-1]])
std = .5

X = np.empty((0, 2))
for i in range(n_clusters):
    X = np.r_[X, means[i] + std * np.random.randn(n_points_per_cluster, 2)]

##############################################################################
# Compute similarities
##############################################################################
X_norms = np.sum(X*X, axis=1)
```

```
S = - X_norms[:,np.newaxis] - X_norms[np.newaxis,:] + 2 * np.dot(X, X.T)
p = 10*np.median(S)

##############################################################################
# Compute Affinity Propagation
##############################################################################

af = AffinityPropagation()
af.fit(S, p)
cluster_centers_indices = af.cluster_centers_indices_
labels = af.labels_

n_clusters_ = len(cluster_centers_indices)

print 'Estimated number of clusters: %d' % n_clusters_

##############################################################################
# Plot result
##############################################################################

import pylab as pl
from itertools import cycle

pl.close('all')
pl.figure(1)
pl.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    class_members = labels == k
    cluster_center = X[cluster_centers_indices[k]]
    pl.plot(X[class_members,0], X[class_members,1], col+'.')
    pl.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
                                    markeredgecolor='k', markersize=14)
    for x in X[class_members]:
        pl.plot([cluster_center[0], x[0]], [cluster_center[1], x[1]], col)

pl.title('Estimated number of clusters: %d' % n_clusters_)
pl.show()
```

Segmenting the picture of Lena in regions

This example uses spectral clustering on a graph created from voxel-to-voxel difference on an image to break this image into multiple partly-homogenous regions.

This procedure (spectral clustering on an image) is an efficient approximate solution for finding normalized graph cuts.



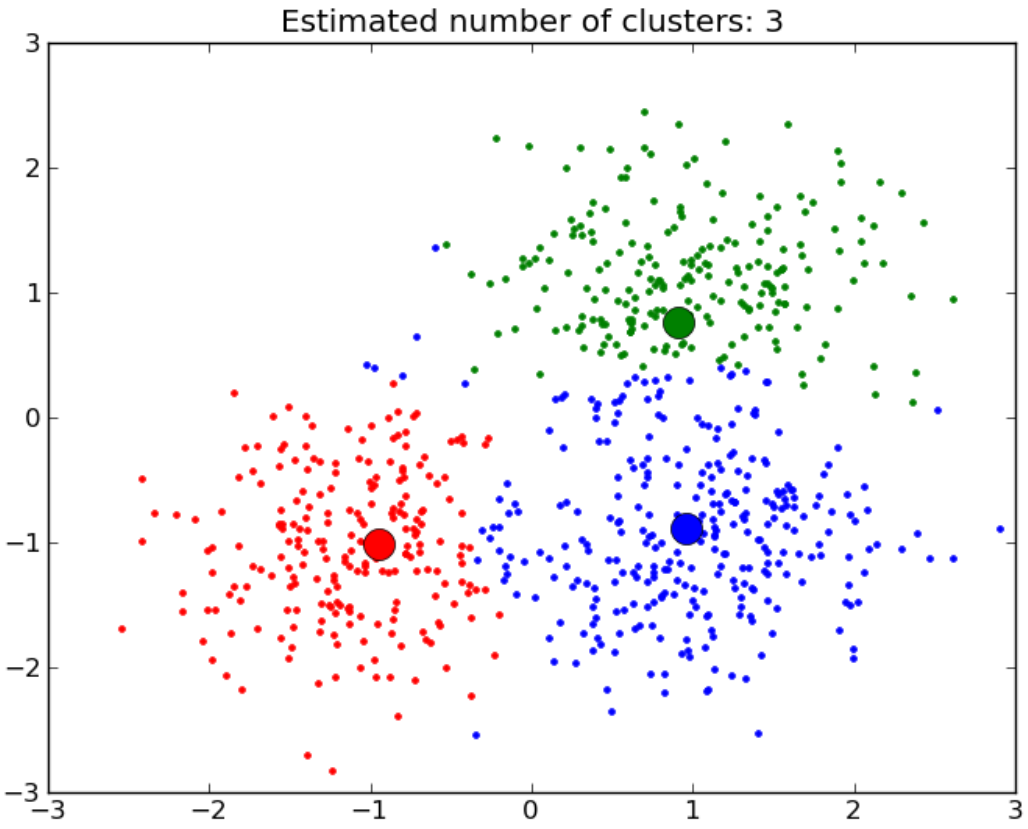
Python source code: `plot_lena_segmentation.py`

```
print __doc__

# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>
# License: BSD

import numpy as np
import scipy as sp
import pylab as pl

from scikits.learn.feature_extraction import image
from scikits.learn.cluster import spectral_clustering

lena = sp.lena()
# Downsample the image by a factor of 4
lena = lena[::2, ::2] + lena[1::2, ::2] + lena[::2, 1::2] + lena[1::2, 1::2]
lena = lena[::2, ::2] + lena[1::2, ::2] + lena[::2, 1::2] + lena[1::2, 1::2]

# Convert the image into a graph with the value of the gradient on the
# edges.
graph = image.img_to_graph(lena)

# Take a decreasing function of the gradient: an exponential
# The smaller beta is, the more independant the segmentation is of the
```

```
# actual image. For beta=1, the segmentation is close to a voronoi
beta = 5
eps  = 1e-6
graph.data = np.exp(-beta*graph.data/lena.std()) + eps

# Apply spectral clustering (this step goes much faster if you have pyamg
# installed)
N_REGIONS = 11
labels = spectral_clustering(graph, k=N_REGIONS)
labels = labels.reshape(lena.shape)

###############################################################################
# Visualize the resulting regions
pl.figure(figsize=(5, 5))
pl.imshow(lena,   cmap=pl.cm.gray)
for l in range(N_REGIONS):
    pl.contour(labels == l, contours=1,
            colors=[pl.cm.spectral(l/float(N_REGIONS)), ])
pl.xticks(())
pl.yticks(())
pl.show()
```

A demo of the mean-shift clustering algorithm

Reference: K. Funkunaga and L.D. Hosteler, "The Estimation of the Gradient of a Density Function, with Applications in Pattern Recognition"



Python source code: `plot_mean_shift.py`

```
print __doc__

import numpy as np
from scikits.learn.cluster import MeanShift, estimate_bandwidth

##############################################################################
# Generate sample data
np.random.seed(0)

n_points_per_cluster = 250
n_clusters = 3
n_points = n_points_per_cluster*n_clusters
means = np.array([[1,1],[-1,-1],[1,-1]])
std = .6
clustMed = []

X = np.empty((0, 2))
for i in range(n_clusters):
    X = np.r_[X, means[i] + std * np.random.randn(n_points_per_cluster, 2)]

##############################################################################
# Compute clustering with MeanShift
bandwidth = estimate_bandwidth(X, quantile=0.3)
ms = MeanShift(bandwidth=bandwidth)
```

```
ms.fit(X)
labels = ms.labels_
cluster_centers = ms.cluster_centers_

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)

print "number of estimated clusters : %d" % n_clusters_

##############################################################################
# Plot result
import pylab as pl
from itertools import cycle

pl.figure(1)
pl.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    cluster_center = cluster_centers[k]
    pl.plot(X[my_members,0], X[my_members,1], col+'.')
    pl.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
                                    markeredgecolor='k', markersize=14)
pl.title('Estimated number of clusters: %d' % n_clusters_)
pl.show()
```

Spectral clustering for image segmentation

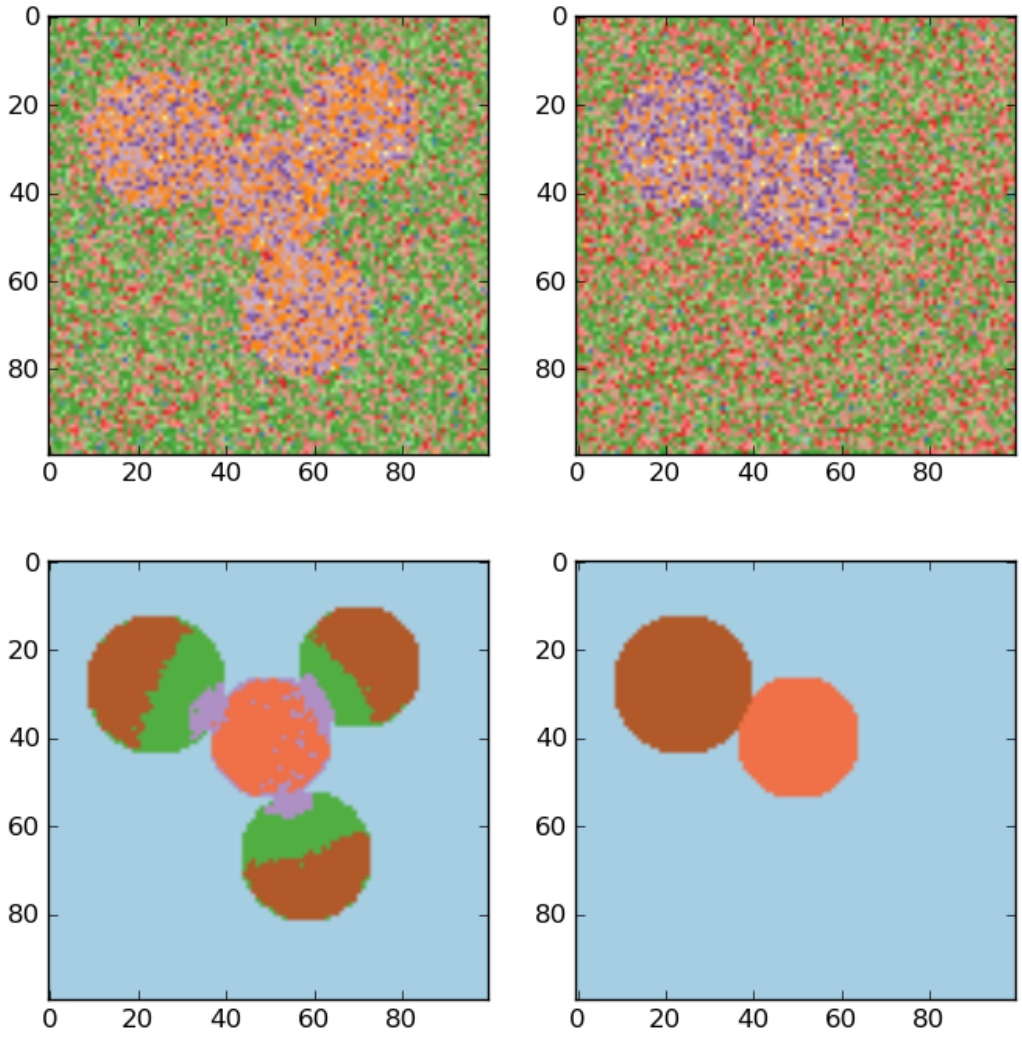
In this example, an image with connected circles is generated and spectral clustering is used to separate the circles.

In these settings, the spectral clustering approach solves the problem know as 'normalized graph cuts': the image is seen as a graph of connected voxels, and the spectral clustering algorithm amounts to choosing graph cuts defining regions while minimizing the ratio of the gradient along the cut, and the volume of the region.

As the algorithm tries to balance the volume (ie balance the region sizes), if we take circles with different sizes, the segmentation fails.

In addition, as there is no useful information in the intensity of the image, or its gradient, we choose to perform the spectral clustering on a graph that is only weakly informed by the gradient. This is close to performing a Voronoi partition of the graph.

In addition, we use the mask of the objects to restrict the graph to the outline of the objects. In this example, we are interested in separating the objects one from the other, and not from the background.



Python source code: `plot_segmentation_toy.py`

```
print __doc__

# Authors:  Emmanuelle Gouillart <emmanuelle.gouillart@normalesup.org>
#           Gael Varoquaux <gael.varoquaux@normalesup.org>
# License: BSD

import numpy as np
import pylab as pl

from scikits.learn.feature_extraction import image
from scikits.learn.cluster import spectral_clustering

################################################################################
```

```
l = 100
x, y = np.indices((l, l))

center1 = (28, 24)
center2 = (40, 50)
center3 = (67, 58)
center4 = (24, 70)

radius1, radius2, radius3, radius4 = 16, 14, 15, 14

circle1 = (x - center1[0])**2 + (y - center1[1])**2 < radius1**2
circle2 = (x - center2[0])**2 + (y - center2[1])**2 < radius2**2
circle3 = (x - center3[0])**2 + (y - center3[1])**2 < radius3**2
circle4 = (x - center4[0])**2 + (y - center4[1])**2 < radius4**2

#############################################################################
# 4 circles
img = circle1 + circle2 + circle3 + circle4
mask = img.astype(bool)
img = img.astype(float)

img += 1 + 0.2*np.random.randn(*img.shape)

# Convert the image into a graph with the value of the gradient on the
# edges.
graph = image.img_to_graph(img, mask=mask)

# Take a decreasing function of the gradient: we take it weakly
# dependant from the gradient the segmentation is close to a voronoi
graph.data = np.exp(-graph.data/graph.data.std())

labels = spectral_clustering(graph, k=4)
label_im = -np.ones(mask.shape)
label_im[mask] = labels

pl.figure(1, figsize=(8, 8))
pl.clf()
pl.subplot(2, 2, 1)
pl.imshow(img)
pl.subplot(2, 2, 3)
pl.imshow(label_im)

#############################################################################
# 2 circles
img = circle1 + circle2
mask = img.astype(bool)
img = img.astype(float)

img += 1 + 0.2*np.random.randn(*img.shape)

graph = image.img_to_graph(img, mask=mask)
graph.data = np.exp(-graph.data/graph.data.std())

labels = spectral_clustering(graph, k=2)
label_im = -np.ones(mask.shape)
label_im[mask] = labels

pl.subplot(2, 2, 2)
```

```
pl.imshow(img)
pl.subplot(2, 2, 4)
pl.imshow(label_im)

pl.show()
```

2.1.4 Gaussian Process for Machine Learning

Examples concerning the *scikits.learn.gaussian_process* package.

Gaussian Processes regression: goodness-of-fit on the 'diabetes' dataset

This example consists in fitting a Gaussian Process model onto the diabetes dataset.

The correlation parameters are determined by means of maximum likelihood estimation (MLE). An anisotropic squared exponential correlation model with a constant regression model are assumed. We also used a nugget = 1e-2 in order to account for the (strong) noise in the targets.

We compute then compute a cross-validation estimate of the coefficient of determination (R2) without reperforming MLE, using the set of correlation parameters found on the whole dataset.

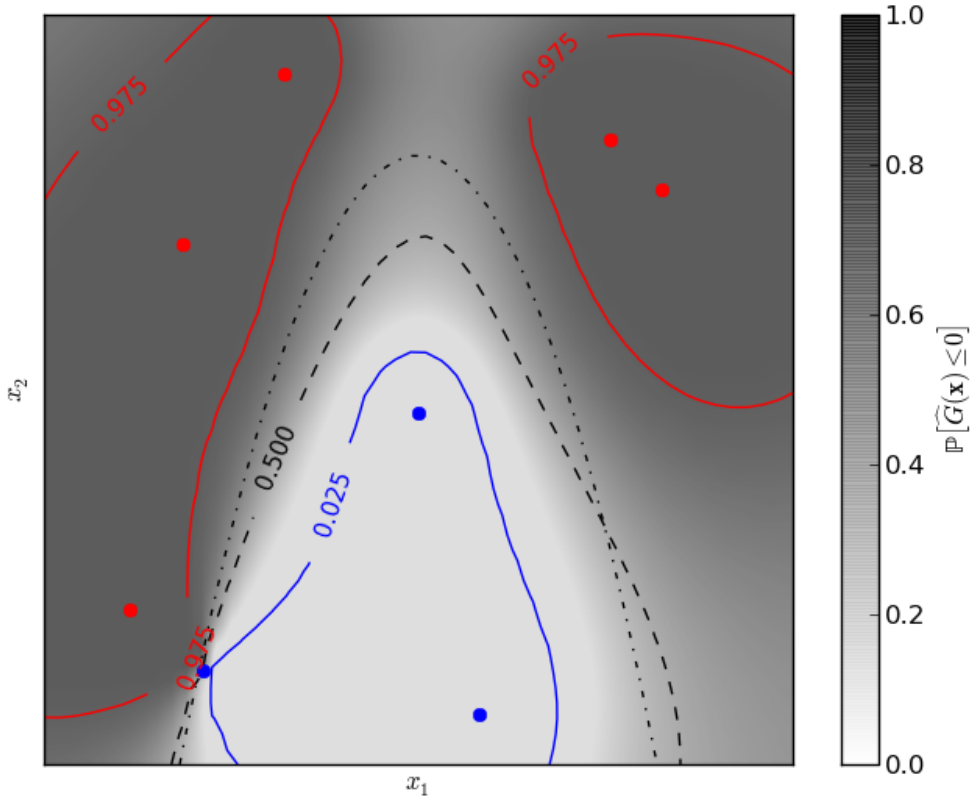
Python source code: `gp_diabetes_dataset.py`

```
print __doc__

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
# License: BSD style

from scikits.learn import datasets
from scikits.learn.gaussian_process import GaussianProcess
from scikits.learn.cross_val import cross_val_score, KFold

# Load the dataset from scikits' data sets
diabetes = datasets.load_diabetes()
X, y = diabetes.data, diabetes.target

# Instanciate a GP model
gp = GaussianProcess(regr='constant', corr='absolute_exponential',
                     theta0=[1e-4] * 10, thetaL=[1e-12] * 10,
                     thetaU=[1e-2] * 10, nugget=1e-2, optimizer='Welch')

# Fit the GP model to the data performing maximum likelihood estimation
gp.fit(X, y)

# Deactivate maximum likelihood estimation for the cross-validation loop
gp.theta0 = gp.theta # Given correlation parameter = MLE
gp.thetaL, gp.thetaU = None, None # None bounds deactivate MLE

# Perform a cross-validation estimate of the coefficient of determination using
# the cross_val module using all CPUs available on the machine
K = 20 # folds
R2 = cross_val_score(gp, X, y=y, cv=KFold(y.size, K), n_jobs=-1).mean()
print("The %d-Folds estimate of the coefficient of determination is R2 = %s"
    % (K, R2))
```

Gaussian Processes classification example: exploiting the probabilistic output

A two-dimensional regression exercise with a post-processing allowing for probabilistic classification thanks to the Gaussian property of the prediction.

The figure illustrates the probability that the prediction is negative with respect to the remaining uncertainty in the prediction. The red and blue lines corresponds to the 95% confidence interval on the prediction of the zero level set.



Python source code: `plot_gp_probabilistic_classification_after_regression.py`

```
print __doc__

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
# License: BSD style

import numpy as np
from scipy import stats
from scikits.learn.gaussian_process import GaussianProcess
from matplotlib import pyplot as pl
from matplotlib import cm

# Standard normal distribution functions
phi = stats.distributions.norm().pdf
PHI = stats.distributions.norm().cdf
PHIinv = stats.distributions.norm().ppf
```

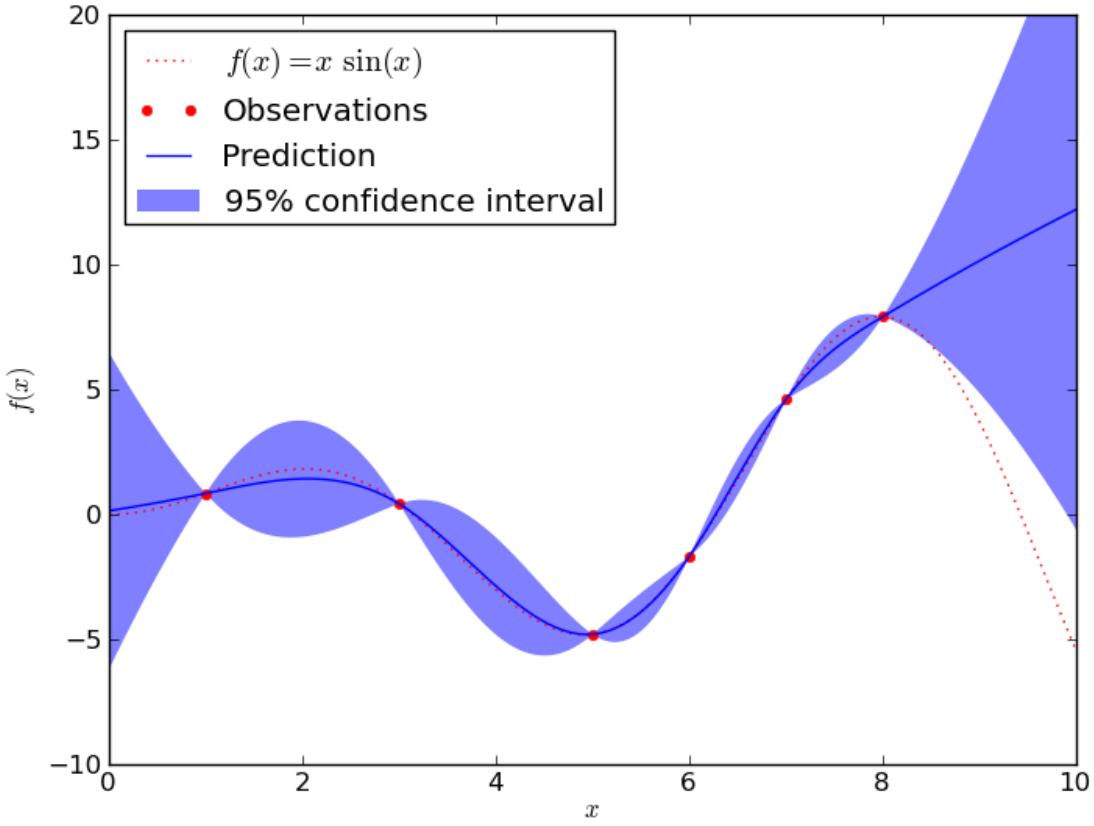
```
# A few constants
lim = 8


def g(x):
    """The function to predict (classification will then consist in predicting
    whether g(x) <= 0 or not)"""
    return 5. - x[:, 1] - .5 * x[:, 0] ** 2.

# Design of experiments
X = np.array([[-4.61611719, -6.00099547],
              [4.10469096, 5.32782448],
              [0.00000000, -0.50000000],
              [-6.17289014, -4.6984743],
              [1.3109306, -6.93271427],
              [-5.03823144, 3.10584743],
              [-2.87600388, 6.74310541],
              [5.21301203, 4.26386883]])

# Observations
y = g(X)

# Instanciate and fit Gaussian Process Model
gp = GaussianProcess(theta0=5e-1)

# Don't perform MLE or you'll get a perfect prediction for this simple example!
gp.fit(X, y)

# Evaluate real function, the prediction and its MSE on a grid
res = 50
x1, x2 = np.meshgrid(np.linspace(- lim, lim, res), \
                     np.linspace(- lim, lim, res))
xx = np.vstack([x1.reshape(x1.size), x2.reshape(x2.size)]).T

y_true = g(xx)
y_pred, MSE = gp.predict(xx, eval_MSE=True)
sigma = np.sqrt(MSE)
y_true = y_true.reshape((res, res))
y_pred = y_pred.reshape((res, res))
sigma = sigma.reshape((res, res))
k = PHIinv(.975)

# Plot the probabilistic classification iso-values using the Gaussian property
# of the prediction
fig = pl.figure(1)
ax = fig.add_subplot(111)
ax.axes.set_aspect('equal')
pl.xticks([])
pl.yticks([])
ax.set_xticklabels([])
ax.set_yticklabels([])
pl.xlabel('$x_1$')
pl.ylabel('$x_2$')

cax = pl.imshow(np.flipud(PHI(- y_pred / sigma)), cmap=cm.gray_r, alpha=0.8, \
                extent=(- lim, lim, - lim, lim))
norm = pl.matplotlib.colors.Normalize(vmin=0., vmax=0.9)
cb = pl.colorbar(cax, ticks=[0., 0.2, 0.4, 0.6, 0.8, 1.], norm=norm)
```

```
cb.set_label('${\\rm \mathbb{P}}\left[\widehat{G}(\mathbf{x}) \leq 0\\right]$')

pl.plot(X[y <= 0, 0], X[y <= 0, 1], 'r.', markersize=12)

pl.plot(X[y > 0, 0], X[y > 0, 1], 'b.', markersize=12)

cs = pl.contour(x1, x2, y_true, [0.], colors='k', \
                linestyles='dashdot')

cs = pl.contour(x1, x2, PHI(- y_pred / sigma), [0.025], colors='b', \
                linestyles='solid')
pl.clabel(cs, fontsize=11)

cs = pl.contour(x1, x2, PHI(- y_pred / sigma), [0.5], colors='k', \
                linestyles='dashed')
pl.clabel(cs, fontsize=11)

cs = pl.contour(x1, x2, PHI(- y_pred / sigma), [0.975], colors='r', \
                linestyles='solid')
pl.clabel(cs, fontsize=11)

pl.show()
```

Gaussian Processes regression: basic introductory example

A simple one-dimensional regression exercise with a cubic correlation model whose parameters are estimated using the maximum likelihood principle.

The figure illustrates the interpolating property of the Gaussian Process model as well as its probabilistic nature in the form of a pointwise 95% confidence interval.



Python source code: `plot_gp_regression.py`

```
print __doc__

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
# License: BSD style

import numpy as np
from scikits.learn.gaussian_process import GaussianProcess
from matplotlib import pyplot as pl


def f(x):
    """The function to predict."""
    return x * np.sin(x)

# The design of experiments
X = np.atleast_2d([1., 3., 5., 6., 7., 8.]).T

# Observations
y = f(X).ravel()

# Mesh the input space for evaluations of the real function, the prediction and
# its MSE
x = np.atleast_2d(np.linspace(0, 10, 1000)).T
```

```
# Instanciate a Gaussian Process model
gp = GaussianProcess(corr='cubic', theta0=1e-2, thetaL=1e-4, thetaU=1e-1, \
                     random_start=100)

# Fit to data using Maximum Likelihood Estimation of the parameters
gp.fit(X, y)

# Make the prediction on the meshed x-axis (ask for MSE as well)
y_pred, MSE = gp.predict(x, eval_MSE=True)
sigma = np.sqrt(MSE)

# Plot the function, the prediction and the 95% confidence interval based on
# the MSE
fig = pl.figure()
pl.plot(x, f(x), 'r:', label=u'$f(x) = x\,\sin(x)$')
pl.plot(X, y, 'r.', markersize=10, label=u'Observations')
pl.plot(x, y_pred, 'b-', label=u'Prediction')
pl.fill(np.concatenate([x, x[::-1]]), \
        np.concatenate([y_pred - 1.9600 * sigma,
                        (y_pred + 1.9600 * sigma)[::-1]]), \
        alpha=.5, fc='b', ec='None', label='95% confidence interval')
pl.xlabel('$x$')
pl.ylabel('$f(x)$')
pl.ylim(-10, 20)
pl.legend(loc='upper left')

pl.show()
```

2.1.5 Generalized Linear Models

Examples concerning the *scikits.learn.linear_model* package.

Lasso regression example

Python source code: `lasso_and_elasticnet.py`

```
print __doc__

import numpy as np

################################################################################
# generate some sparse data to play with

n_samples, n_features = 50, 200
X = np.random.randn(n_samples, n_features)
coef = 3*np.random.randn(n_features)
coef[10:] = 0 # sparsify coef
y = np.dot(X, coef)

# add noise
y += 0.01*np.random.normal((n_samples,))

# Split data in train set and test set
n_samples = X.shape[0]
X_train, y_train = X[:n_samples/2], y[:n_samples/2]
X_test, y_test = X[n_samples/2:], y[n_samples/2:]
```

```
##############################################################################
# Lasso
from scikits.learn.linear_model import Lasso

alpha = 0.1
lasso = Lasso(alpha=alpha)

y_pred_lasso = lasso.fit(X_train, y_train).predict(X_test)
print lasso
print "r^2 on test data : %f" % (1 - np.linalg.norm(y_test - y_pred_lasso)**2
                                    / np.linalg.norm(y_test)**2)

##############################################################################
# ElasticNet
from scikits.learn.linear_model import ElasticNet

enet = ElasticNet(alpha=alpha, rho=0.7)

y_pred_enet = enet.fit(X_train, y_train).predict(X_test)
print enet
print "r^2 on test data : %f" % (1 - np.linalg.norm(y_test - y_pred_enet)**2
                                    / np.linalg.norm(y_test)**2)
```

Lasso on dense and sparse data

We show that linear_model.Lasso and linear_model.sparse.Lasso provide the same results and that in the case of sparse data linear_model.sparse.Lasso improves the speed.

Python source code: `lasso_dense_vs_sparse_data.py`

```
print __doc__

from time import time
import numpy as np
from scipy import sparse
from scipy import linalg

from scikits.learn.linear_model.sparse import Lasso as SparseLasso
from scikits.learn.linear_model import Lasso as DenseLasso


##############################################################################
# The two Lasso implementations on Dense data
print "--- Dense matrices"

n_samples, n_features = 200, 10000
np.random.seed(0)
y = np.random.randn(n_samples)
X = np.random.randn(n_samples, n_features)

alpha = 1
sparse_lasso = SparseLasso(alpha=alpha, fit_intercept=False)
dense_lasso = DenseLasso(alpha=alpha, fit_intercept=False)

t0 = time()
sparse_lasso.fit(X, y, max_iter=1000)
print "Sparse Lasso done in %fs" % (time() - t0)
```

```
t0 = time()
dense_lasso.fit(X, y, max_iter=1000)
print "Dense Lasso done in %fs" % (time() - t0)

print "Distance between coefficients : %s" % linalg.norm(sparse_lasso.coef_
                                                         - dense_lasso.coef_)

##############################################################################
# The two Lasso implementations on Sparse data
print "--- Sparse matrices"

Xs = X.copy()
Xs[Xs < 2.5] = 0.0
Xs = sparse.coo_matrix(Xs)
Xs = Xs.tocsc()

print "Matrix density : %s %%" % (Xs.nnz / float(X.size) * 100)

alpha = 0.1
sparse_lasso = SparseLasso(alpha=alpha, fit_intercept=False)
dense_lasso = DenseLasso(alpha=alpha, fit_intercept=False)

t0 = time()
sparse_lasso.fit(Xs, y, max_iter=1000)
print "Sparse Lasso done in %fs" % (time() - t0)

t0 = time()
dense_lasso.fit(Xs.todense(), y, max_iter=1000)
print "Dense Lasso done in %fs" % (time() - t0)

print "Distance between coefficients : %s" % linalg.norm(sparse_lasso.coef_
                                                         - dense_lasso.coef_)
```

Lasso parameter estimation with path and cross-validation

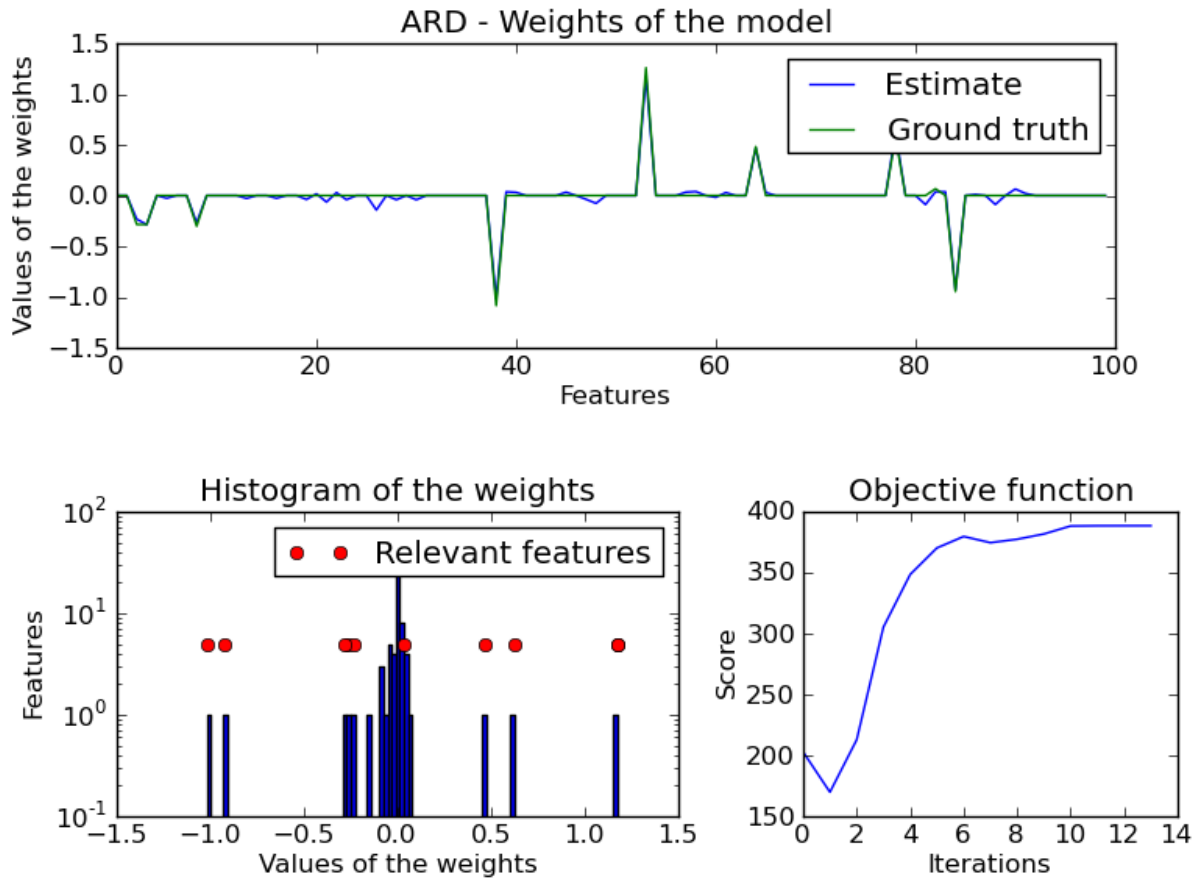
Python source code: `lasso_path_with_crossvalidation.py`

```
print __doc__

import numpy as np

##############################################################################
# generate some sparse data to play with

n_samples, n_features = 60, 100

np.random.seed(1)
X = np.random.randn(n_samples, n_features)
coef = 3*np.random.randn(n_features)
coef[10:] = 0 # sparsify coef
y = np.dot(X, coef)

# add noise
y += 0.01 * np.random.normal((n_samples,))

# Split data in train set and test set
X_train, y_train = X[:n_samples/2], y[:n_samples/2]
```

```
X_test, y_test = X[n_samples/2:], y[n_samples/2:]


##############################################################################
# Lasso with path and cross-validation using LassoCV path
from scikits.learn.linear_model import LassoCV
from scikits.learn.cross_val import KFold

cv = KFold(n_samples/2, 5)
lasso_cv = LassoCV()

# fit_params = {'max_iter':100}

y_ = lasso_cv.fit(X_train, y_train, cv=cv, max_iter=100).predict(X_test)

print "Optimal regularization parameter  = %s" % lasso_cv.alpha

# Compute explained variance on test data
print "r^2 on test data : %f" % (1 - np.linalg.norm(y_test - y_)**2
                                   / np.linalg.norm(y_test)**2)
```

Automatic Relevance Determination Regression (ARD)

Fit regression model with ARD



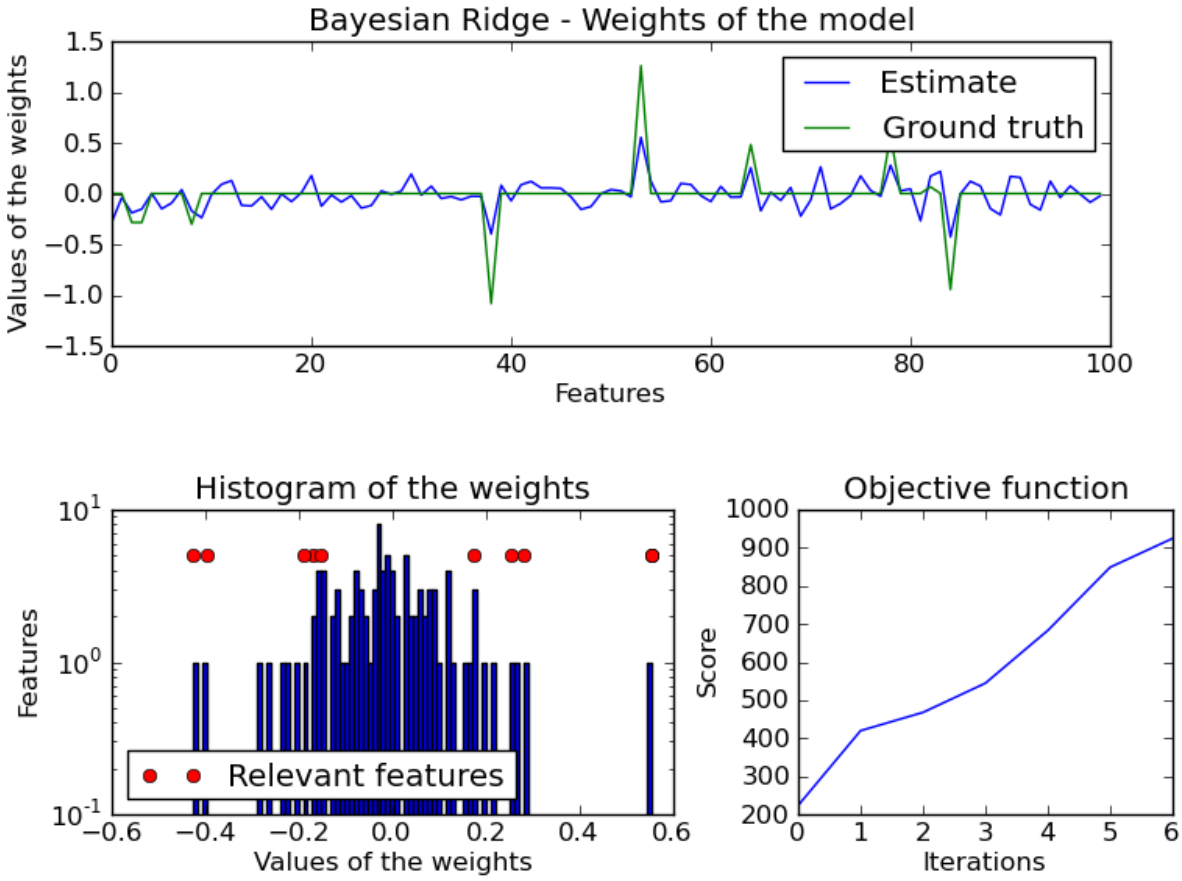
Python source code: plot_ard.py

```
print __doc__

import numpy as np
import pylab as pl
from scipy import stats

from scikits.learn.linear_model import ARDRegression

###############################################################################
# Generating simulated data with Gaussian weigthts

### Parameters of the example
np.random.seed(0)
n_samples, n_features = 50, 100
### Create gaussian data
X = np.random.randn(n_samples, n_features)
### Create weigts with a precision lambda_ of 4.
lambda_ = 4.
w = np.zeros(n_features)
### Only keep 10 weights of interest
relevant_features = np.random.randint(0, n_features, 10)
for i in relevant_features:
    w[i] = stats.norm.rvs(loc=0, scale=1. / np.sqrt(lambda_))
### Create noite with a precision alpha of 50.
alpha_ = 50.
noise =  stats.norm.rvs(loc=0, scale=1. / np.sqrt(alpha_), size=n_samples)
### Create the target
y = np.dot(X, w) + noise

###############################################################################
### Fit the ARD Regression
clf = ARDRegression(compute_score = True)
clf.fit(X, y)

###############################################################################
### Plot the true weights, the estimated weights and the histogram of the
### weights
pl.figure()
axe = pl.axes([0.1,0.6,0.8,0.325])
axe.set_title("ARD - Weights of the model")
axe.plot(clf.coef_, 'b-', label="Estimate")
axe.plot(w, 'g-', label="Ground truth")
axe.set_xlabel("Features")
axe.set_ylabel("Values of the weights")
axe.legend(loc=1)

axe = pl.axes([0.1,0.1,0.45,0.325])
axe.set_title("Histogram of the weights")
axe.hist(clf.coef_, bins=n_features, log=True)
axe.plot(clf.coef_[relevant_features],5*np.ones(len(relevant_features)),'ro',
label="Relevant features")
axe.set_ylabel("Features")
axe.set_xlabel("Values of the weights")
axe.legend(loc=1)

axe = pl.axes([0.65,0.1,0.3,0.325])
axe.set_title("Objective function")
```

```
axe.plot(clf.scores_)
axe.set_ylabel("Score")
axe.set_xlabel("Iterations")
pl.show()
```

Bayesian Ridge Regression

Computes a Bayesian Ridge Regression on a synthetic dataset



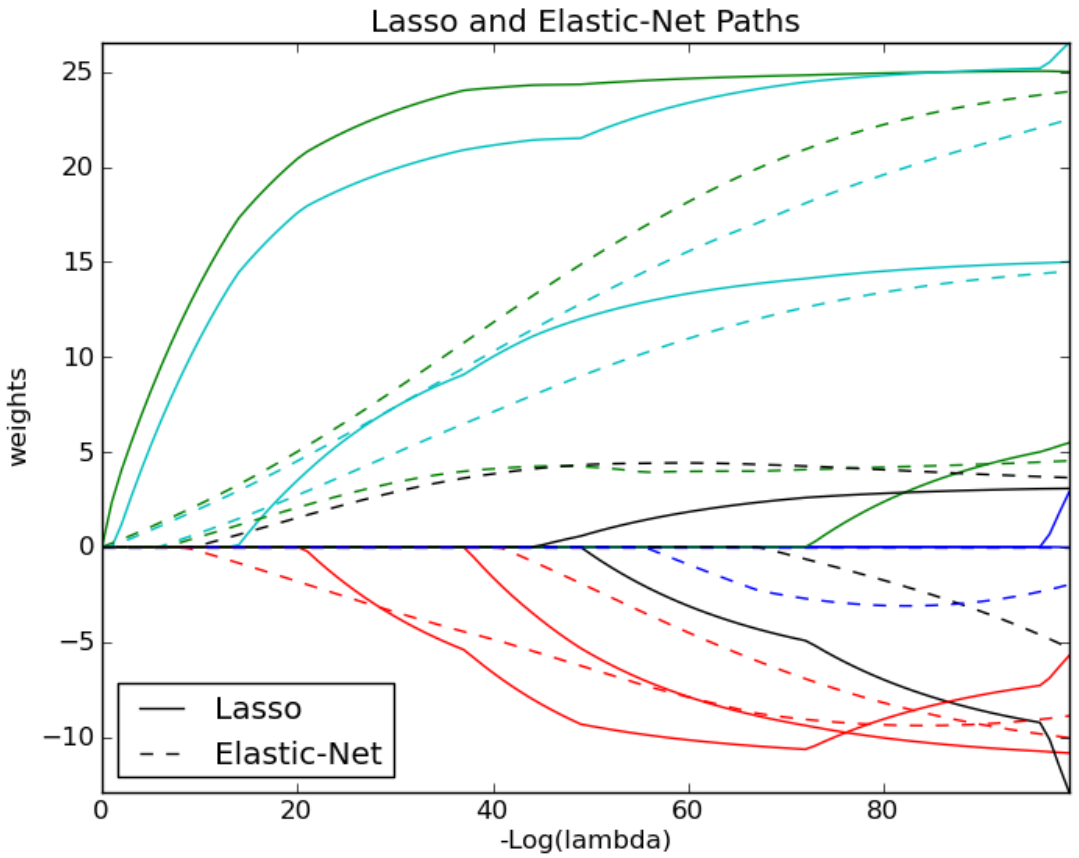
Python source code: `plot_bayesian_ridge.py`

```
print __doc__

import numpy as np
import pylab as pl
from scipy import stats

from scikits.learn.linear_model import BayesianRidge

###############################################################################
# Generating simulated data with Gaussian weigthts
np.random.seed(0)
n_samples, n_features = 50, 100
X = np.random.randn(n_samples, n_features) # Create gaussian data
```

```
# Create weigts with a precision lambda_ of 4.
lambda_ = 4.
w = np.zeros(n_features)
# Only keep 10 weights of interest
relevant_features = np.random.randint(0, n_features, 10)
for i in relevant_features:
    w[i] = stats.norm.rvs(loc = 0, scale = 1./np.sqrt(lambda_))
# Create noite with a precision alpha of 50.
alpha_ = 50.
noise =  stats.norm.rvs(loc = 0, scale = 1./np.sqrt(alpha_), size = n_samples)
# Create the target
y = np.dot(X, w) + noise

##############################################################################
# Fit the Bayesian Ridge Regression
clf = BayesianRidge(compute_score=True)
clf.fit(X, y)

##############################################################################
# Plot true weights, estimated weights and histogram of the weights
pl.figure()
axe = pl.axes([0.1,0.6,0.8,0.325])
axe.set_title("Bayesian Ridge - Weights of the model")
axe.plot(clf.coef_, 'b-', label="Estimate")
axe.plot(w, 'g-', label="Ground truth")
axe.set_xlabel("Features")
axe.set_ylabel("Values of the weights")
axe.legend(loc="upper right")

axe = pl.axes([0.1,0.1,0.45,0.325])
axe.set_title("Histogram of the weights")
axe.hist(clf.coef_, bins=n_features, log=True)
axe.plot(clf.coef_[relevant_features],5*np.ones(len(relevant_features)),'ro',
label="Relevant features")
axe.set_ylabel("Features")
axe.set_xlabel("Values of the weights")
axe.legend(loc="lower left")

axe = pl.axes([0.65,0.1,0.3,0.325])
axe.set_title("Objective function")
axe.plot(clf.scores_)
axe.set_ylabel("Score")
axe.set_xlabel("Iterations")
pl.show()
```

Lasso and Elastic Net

Lasso and elastic net (L1 and L2 penalisation) implemented using a coordinate descent.



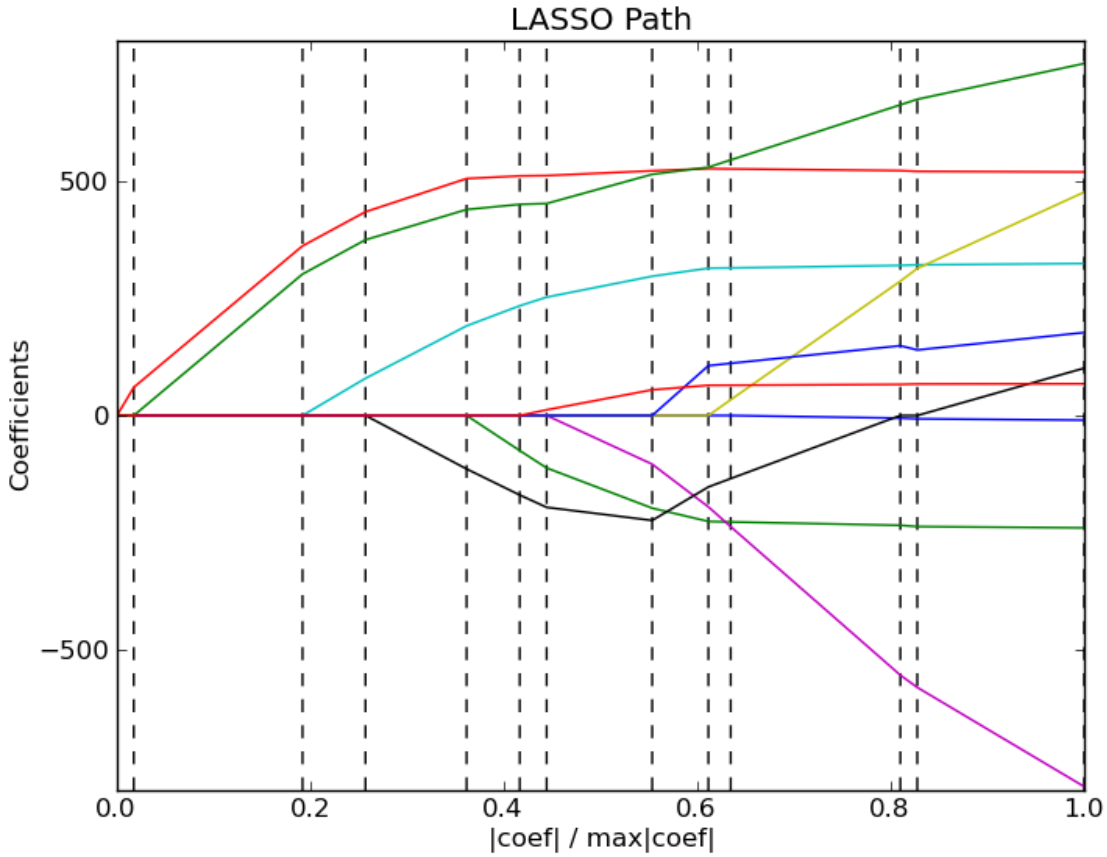
Python source code: `plot_lasso_coordinate_descent_path.py`

```
print __doc__

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD Style.

import numpy as np
import pylab as pl

from scikits.learn.linear_model import lasso_path, enet_path
from scikits.learn import datasets

diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target

X /= X.std(0) # Standardize data (easier to set the rho parameter)

##############################################################################
# Compute paths

eps = 5e-3 # the smaller it is the longer is the path

print "Computing regularization path using the lasso..."
models = lasso_path(X, y, eps=eps)
```

```
alphas_lasso = np.array([model.alpha for model in models])
coefs_lasso = np.array([model.coef_ for model in models])

print "Computing regularization path using the elastic net..."
models = enet_path(X, y, eps=eps, rho=0.8)
alphas_enet = np.array([model.alpha for model in models])
coefs_enet = np.array([model.coef_ for model in models])

##############################################################################
# Display results

ax = pl.gca()
ax.set_color_cycle(2 * ['b', 'r', 'g', 'c', 'k'])
l1 = pl.plot(coefs_lasso)
l2 = pl.plot(coefs_enet, linestyle='--')

pl.xlabel('-Log(lambda)')
pl.ylabel('weights')
pl.title('Lasso and Elastic-Net Paths')
pl.legend((l1[-1], l2[-1]), ('Lasso', 'Elastic-Net'), loc='lower left')
pl.axis('tight')
pl.show()
```

Lasso path using LARS

Computes Lasso Path along the regularization parameter using the LARS algorithm on the diabetest dataset.



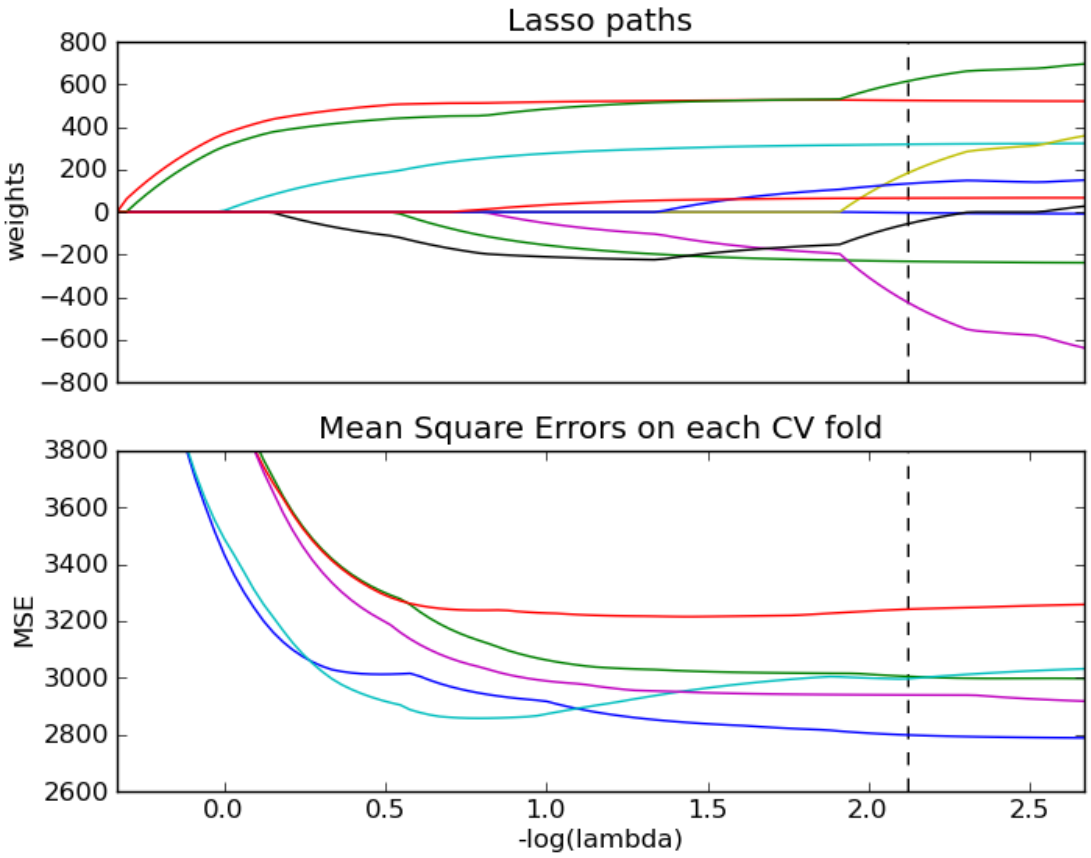
Python source code: `plot_lasso_lars.py`

```
print __doc__

# Author: Fabian Pedregosa <fabian.pedregosa@inria.fr>
#         Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD Style.

import numpy as np
import pylab as pl

from scikits.learn import linear_model
from scikits.learn import datasets

diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target

print "Computing regularization path using the LARS ..."
alphas, _, coefs = linear_model.lars_path(X, y, method='lasso', verbose=True)

xx = np.sum(np.abs(coefs.T), axis=1)
xx /= xx[-1]

pl.plot(xx, coefs.T)
ymin, ymax = pl.ylim()
```

```
pl.vlines(xx, ymin, ymax, linestyle='dashed')
pl.xlabel('|coef| / max|coef|')
pl.ylabel('Coefficients')
pl.title('LASSO Path')
pl.axis('tight')
pl.show()
```

Cross validated Lasso path with coordinate descent

Compute a 5-fold cross-validated Lasso path with coordinate descent to find the optimal value of alpha.



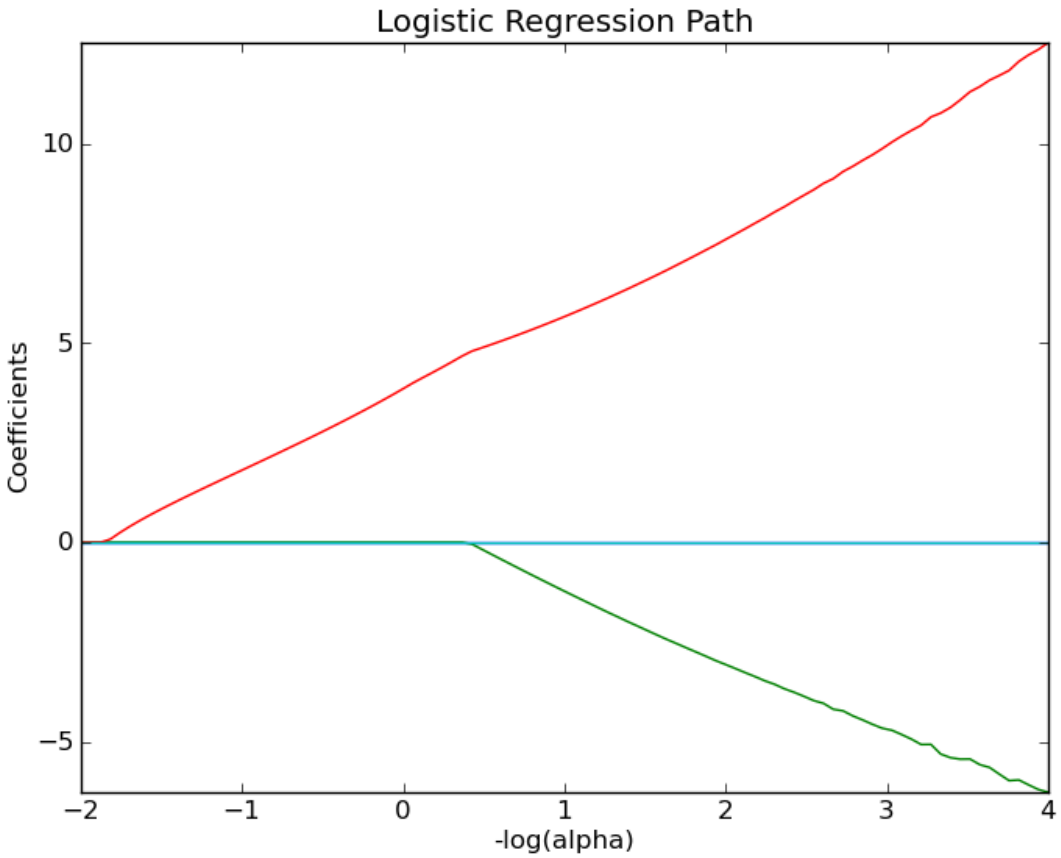
Python source code: `plot_lasso_path_crossval.py`

```
print __doc__

# Author: Olivier Grisel
# License: BSD Style.

import numpy as np
import pylab as pl

from scikits.learn.linear_model import LassoCV
from scikits.learn import datasets
```

```
diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target

# normalize data as done by LARS to allow for comparison
X /= np.sqrt(np.sum(X ** 2, axis=0))

##############################################################################
# Compute paths

eps = 1e-3 # the smaller it is the longer is the path

print "Computing regularization path using the lasso..."
model = LassoCV(eps=eps).fit(X, y)

##############################################################################
# Display results
m_log_alphas = -np.log10(model.alphas)
m_log_alpha = -np.log10(model.alpha)

ax = pl.gca()
ax.set_color_cycle(2 * ['b', 'r', 'g', 'c', 'k'])
pl.subplot(2, 1, 1)
pl.plot(m_log_alphas, model.coef_path_)

ymin, ymax = pl.ylim()
pl.vlines([m_log_alpha], ymin, ymax, linestyle='dashed')

pl.xticks(())
pl.ylabel('weights')
pl.title('Lasso paths')
pl.axis('tight')

pl.subplot(2, 1, 2)
ymin, ymax = 2600, 3800
pl.plot(m_log_alphas, model.mse_path_)
pl.vlines([m_log_alpha], ymin, ymax, linestyle='dashed')

pl.xlabel('-log(lambda)')
pl.ylabel('MSE')
pl.title('Mean Square Errors on each CV fold')
pl.axis('tight')
pl.ylim(ymin, ymax)

pl.show()
```

Path with L1- Logistic Regression

Computes path on IRIS dataset.



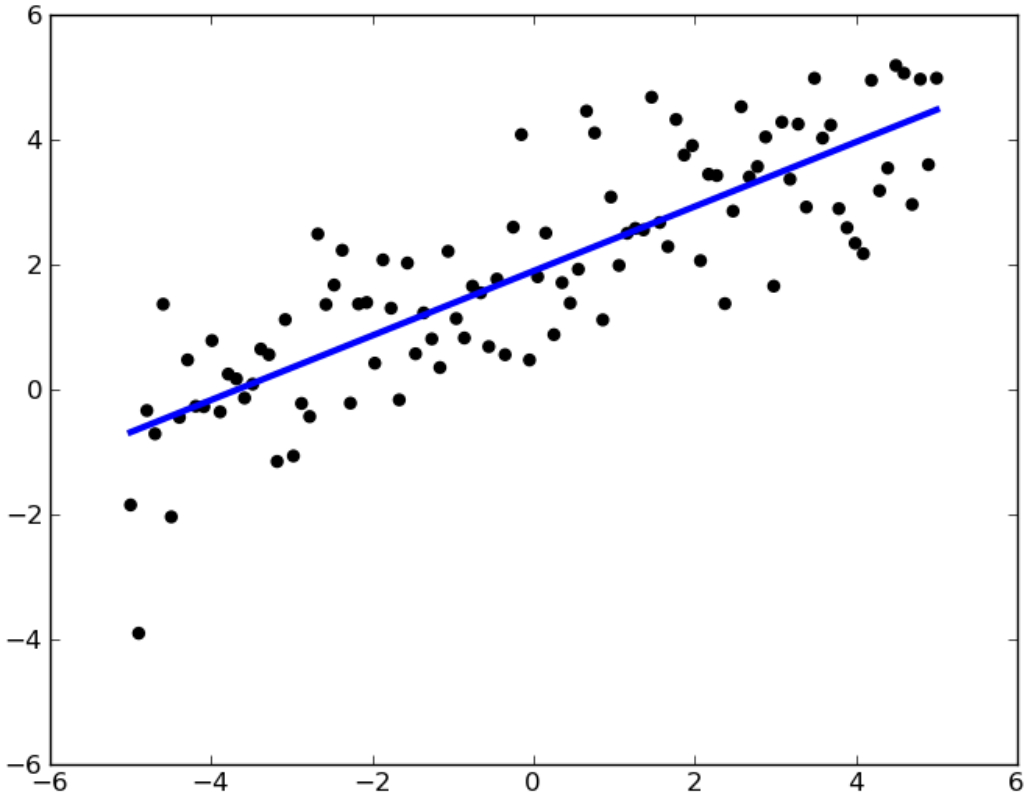
Python source code: `plot_logistic_path.py`

```
print __doc__

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD Style.

from datetime import datetime
import numpy as np
import pylab as pl

from scikits.learn import linear_model
from scikits.learn import datasets

iris = datasets.load_iris()
X = iris.data
y = iris.target

X = X[y != 2]
y = y[y != 2]

X -= np.mean(X, 0)

################################################################################
# Demo path functions
```

```
alphas = np.logspace(2, -4, 100)

print "Computing regularization path ..."
start = datetime.now()
clf = linear_model.LogisticRegression(C=1.0, penalty='l1', eps=1e-6)
coefs_ = [clf.fit(X, y, C=1.0/alpha).coef_.ravel().copy() for alpha in alphas]
print "This took ", datetime.now() - start

coefs_ = np.array(coefs_)
pl.plot(-np.log10(alphas), coefs_)
ymin, ymax = pl.ylim()
pl.xlabel('-log(alpha)')
pl.ylabel('Coefficients')
pl.title('Logistic Regression Path')
pl.axis('tight')
pl.show()
```

Ordinary Least Squares

Simple Ordinary Least Squares example, we draw the linear least squares solution for a random set of points in the plane.

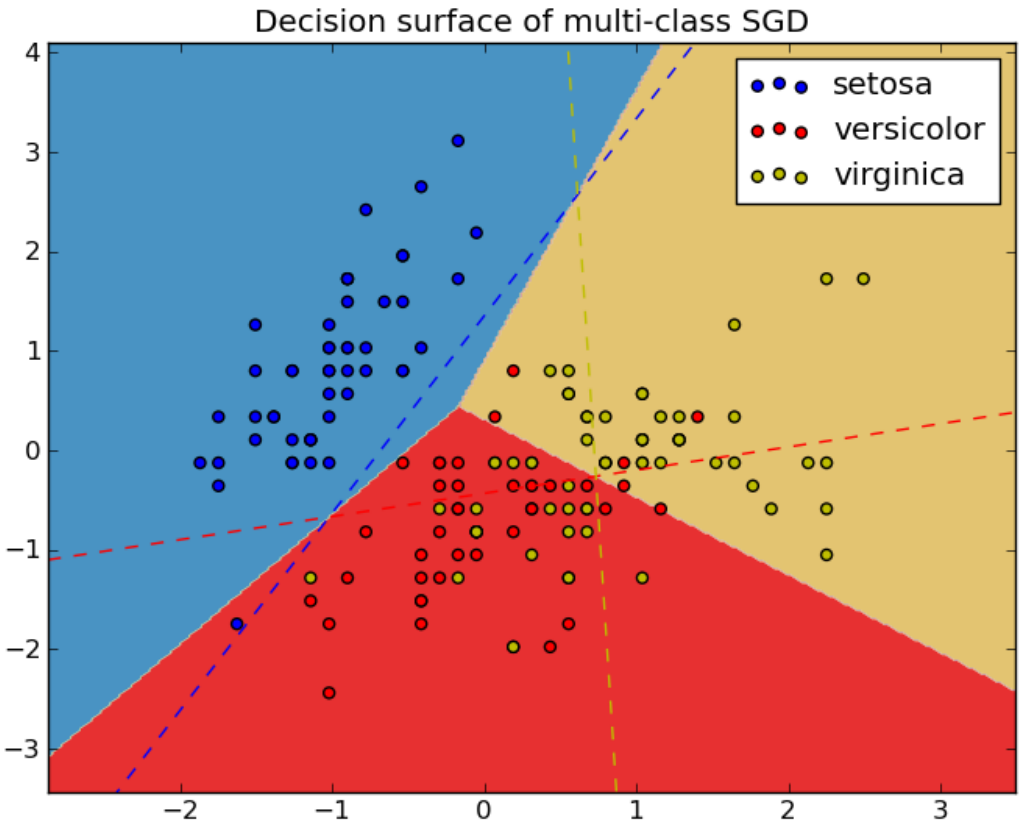


Python source code: `plot_ols.py`

```
print __doc__

import numpy as np
import pylab as pl

from scikits.learn import linear_model

# this is our test set, it's just a straight line with some
# gaussian noise
xmin, xmax = -5, 5
n_samples = 100
X = [[i] for i in np.linspace(xmin, xmax, n_samples)]
Y = 2 + 0.5 * np.linspace(xmin, xmax, n_samples) \
      + np.random.randn(n_samples, 1).ravel()

# run the classifier
clf = linear_model.LinearRegression()
clf.fit(X, Y)

# and plot the result
pl.scatter(X, Y, color='black')
pl.plot(X, clf.predict(X), color='blue', linewidth=3)
pl.show()
```

Plot multi-class SGD on the iris dataset

Plot decision surface of multi-class SGD on iris dataset. The hyperplanes corresponding to the three one-versus-all (OVA) classifiers are represented by the dashed lines.



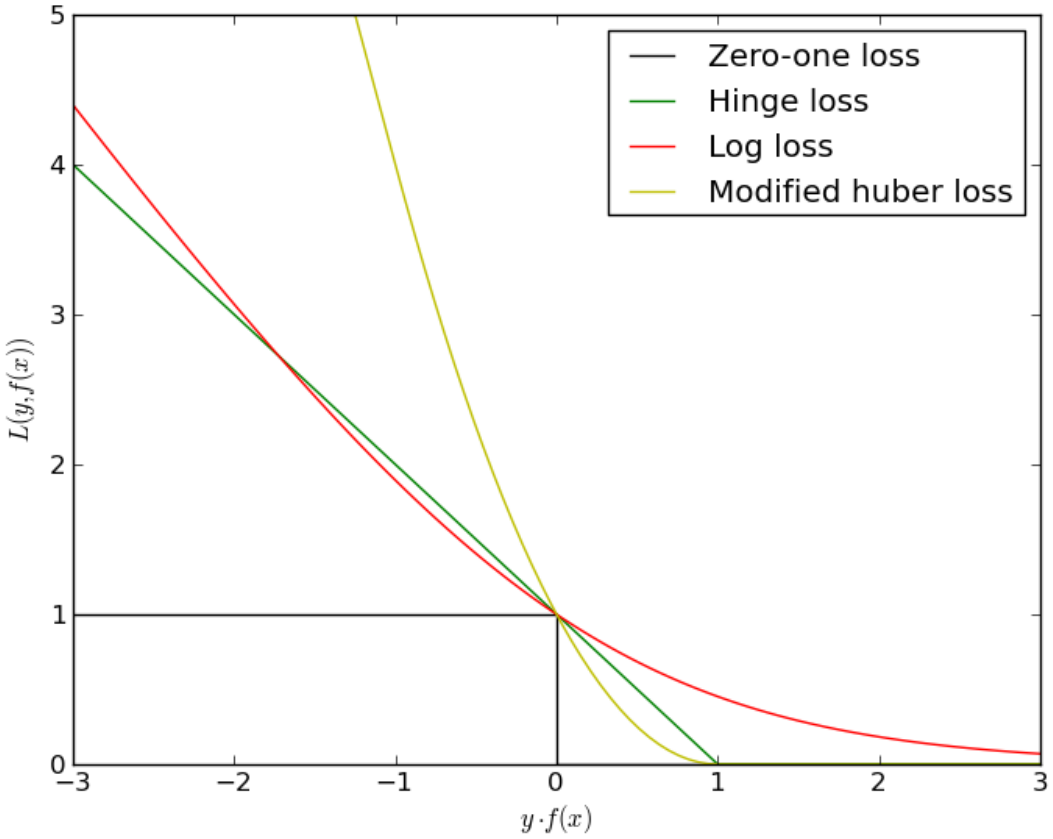
Python source code: `plot_sgd_iris.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn import datasets
from scikits.learn.linear_model import SGDClassifier

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features. We could
                      # avoid this ugly slicing by using a two-dim dataset
y = iris.target
colors = "bry"

# shuffle
idx = np.arange(X.shape[0])
np.random.seed(13)
np.random.shuffle(idx)
X = X[idx]
y = y[idx]

# standardize
mean = X.mean(axis=0)
std = X.std(axis=0)
```

```
X = (X - mean) / std

h = .02  # step size in the mesh

clf = SGDClassifier(alpha=0.001, n_iter=100).fit(X, y)

# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

pl.set_cmap(pl.cm.Paired)

# Plot the decision boundary. For that, we will asign a color to each
# point in the mesh [x_min, m_max]x[y_min, y_max].
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
pl.set_cmap(pl.cm.Paired)
cs = pl.contourf(xx, yy, Z)
pl.axis('tight')

# Plot also the training points
for i, color in zip(clf.classes, colors):
    idx = np.where(y == i)
    pl.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[i])
pl.title("Decision surface of multi-class SGD")
pl.axis('tight')

# Plot the three one-against-all classifiers
xmin, xmax = pl.xlim()
ymin, ymax = pl.ylim()
coef = clf.coef_
intercept = clf.intercept_


def plot_hyperplane(c, color):
    def line(x0):
        return (-(x0 * coef[c, 0]) - intercept[c]) / coef[c, 1]

    pl.plot([xmin, xmax], [line(xmin), line(xmax)],
            ls="--", color=color)

for i, color in zip(clf.classes, colors):
    plot_hyperplane(i, color)
pl.legend()
pl.show()
```

SGD: Convex Loss Functions

Plot the convex loss functions supported by *scikits.learn.linear_model.stochastic_gradient*.



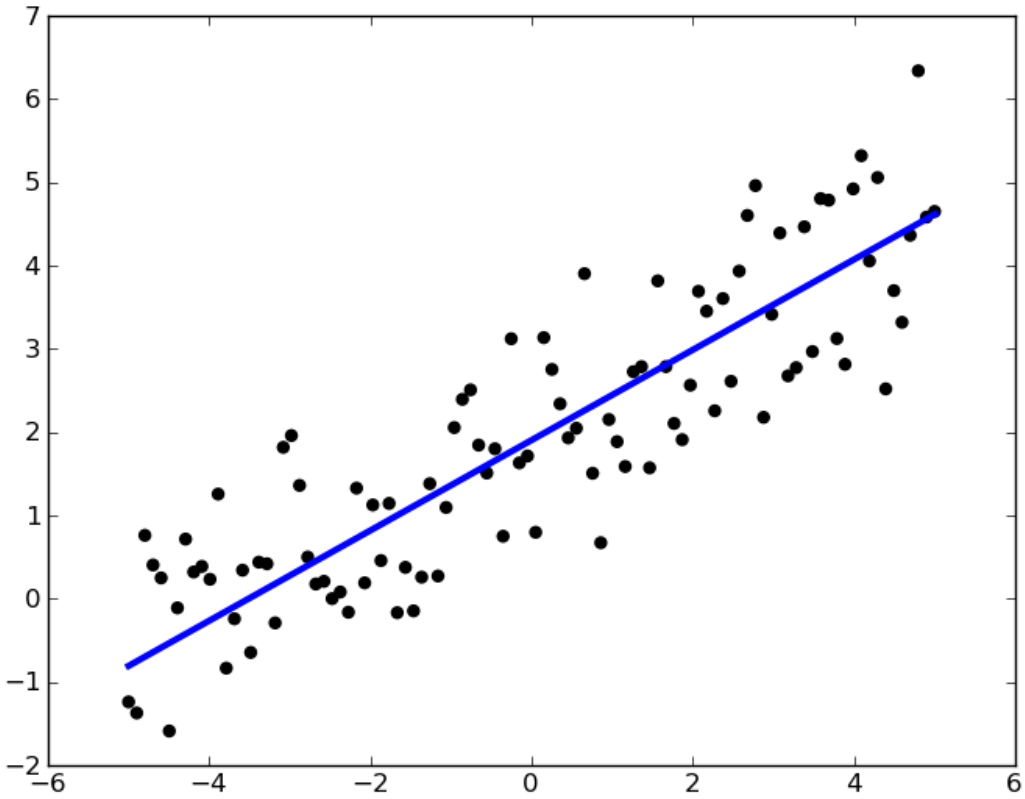
Python source code: `plot_sgd_loss_functions.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn.linear_model.sgd_fast import Hinge, \
     ModifiedHuber, SquaredLoss

###############################################################################
# Define loss funcitons
xmin, xmax = -3, 3
hinge = Hinge()
log_loss = lambda z, p: np.log2(1.0 + np.exp(-z))
modified_huber = ModifiedHuber()
squared_loss = SquaredLoss()

###############################################################################
# Plot loss funcitons
xx = np.linspace(xmin, xmax, 100)
pl.plot([xmin, 0, 0, xmax], [1, 1, 0, 0], 'k-',
        label="Zero-one loss")
pl.plot(xx, [hinge.loss(x,1) for x in xx], 'g-',
        label="Hinge loss")
pl.plot(xx, [log_loss(x,1) for x in xx], 'r-',
        label="Log loss")
```

```
pl.plot(xx, [modified_huber.loss(x,1) for x in xx], 'y-',
        label="Modified huber loss")
#pl.plot(xx, [2.0*squared_loss.loss(x,1) for x in xx], 'c-',
#        label="Squared loss")
pl.ylim((0, 5))
pl.legend(loc="upper right")
pl.xlabel(r"$y \cdot f(x)$")
pl.ylabel("$L(y, f(x))$")
pl.show()
```

Ordinary Least Squares with SGD

Simple Ordinary Least Squares example with stochastic gradient descent, we draw the linear least squares solution for a random set of points in the plane.



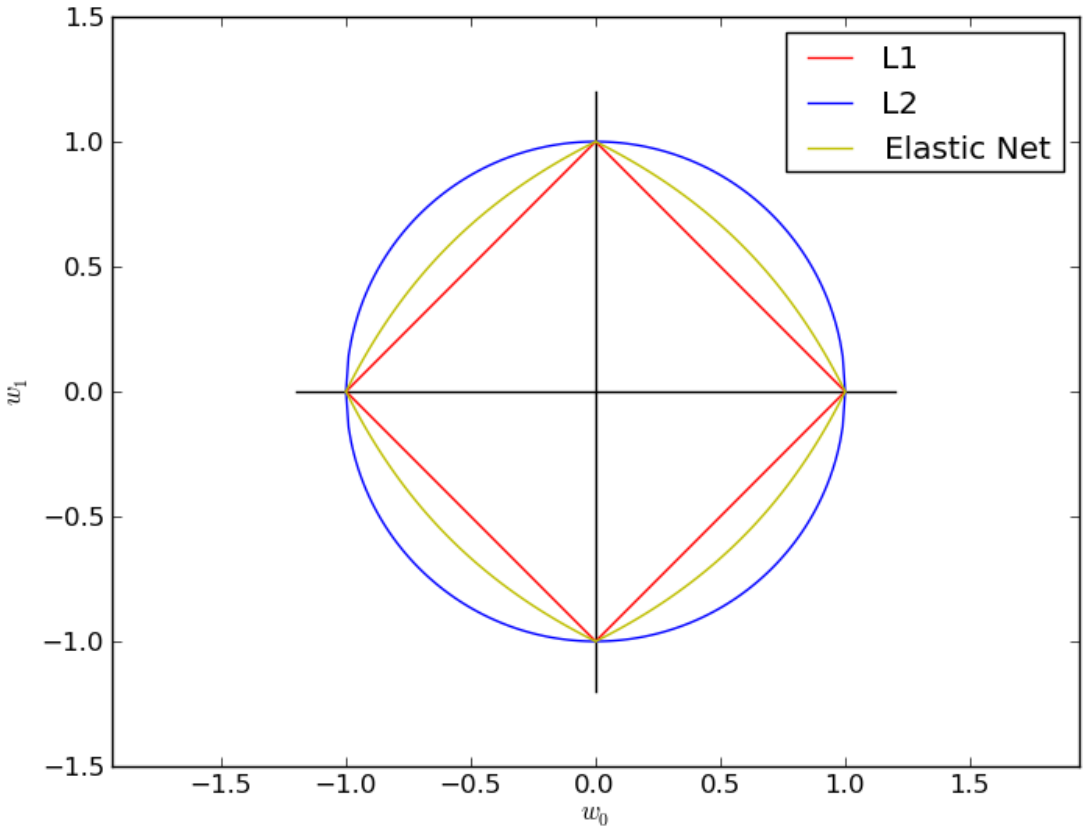
Python source code: `plot_sgd_ols.py`

```
print __doc__

import numpy as np
import pylab as pl

from scikits.learn.linear_model import SGDRegressor
```

```
# this is our test set, it's just a straight line with some
# gaussian noise
xmin, xmax = -5, 5
n_samples = 100
X = [[i] for i in np.linspace(xmin, xmax, n_samples)]
Y = 2 + 0.5 * np.linspace(xmin, xmax, n_samples) \
      + np.random.randn(n_samples, 1).ravel()

# run the classifier
clf = SGDRegressor(alpha=0.1, n_iter=20)
clf.fit(X, Y)

# and plot the result
pl.scatter(X, Y, color='black')
pl.plot(X, clf.predict(X), color='blue', linewidth=3)
pl.show()
```

SGD: Penalties

Plot the contours of the three penalties supported by *scikits.learn.linear_model.stochastic_gradient*.

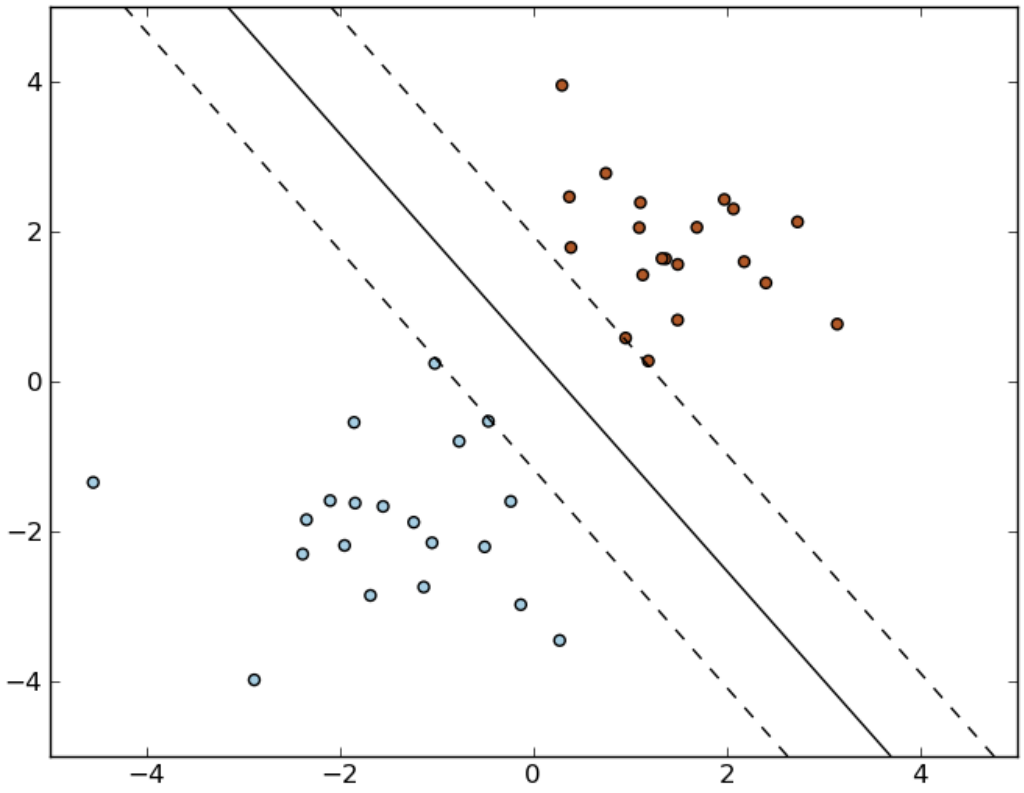


Python source code: `plot_sgd_penalties.py`

```
from __future__ import division
print __doc__

import numpy as np
import pylab as pl

def l1(xs): return np.array([np.sqrt((1 - np.sqrt(x**2.0))**2.0) for x in xs])

def l2(xs): return np.array([np.sqrt(1.0-x**2.0) for x in xs])

def el(xs, z):
    return np.array([(2 - 2*x - 2*z + 4*x*z -
                      (4*z**2 - 8*x*z**2 + 8*x**2*z**2 -
                       16*x**2*z**3 + 8*x*z**3 + 4*x**2*z**4)**(1/2) -
                      2*x*z**2)/(2 - 4*z) for x in xs])

def cross(ext):
    pl.plot([-ext,ext],[0,0], "k-")
    pl.plot([0,0], [-ext,ext], "k-")

xs = np.linspace(0, 1, 100)

alpha = 0.501 # 0.5 division throuh zero

cross(1.2)

pl.plot(xs, l1(xs), "r-", label="L1")
pl.plot(xs, -1.0*l1(xs), "r-")
pl.plot(-1*xs, l1(xs), "r-")
pl.plot(-1*xs, -1.0*l1(xs), "r-")

pl.plot(xs, l2(xs), "b-", label="L2")
pl.plot(xs, -1.0 * l2(xs), "b-")
pl.plot(-1*xs, l2(xs), "b-")
pl.plot(-1*xs, -1.0 * l2(xs), "b-")

pl.plot(xs, el(xs, alpha), "y-", label="Elastic Net")
pl.plot(xs, -1.0 * el(xs, alpha), "y-")
pl.plot(-1*xs, el(xs, alpha), "y-")
pl.plot(-1*xs, -1.0 * el(xs, alpha), "y-")

pl.xlabel(r"$w_0$")
pl.ylabel(r"$w_1$")
pl.legend()

pl.axis("equal")
pl.show()
```

SGD: Maximum margin separating hyperplane

Plot the maximum margin separating hyperplane within a two-class separable dataset using a linear Support Vector Machines classifier trained using SGD.



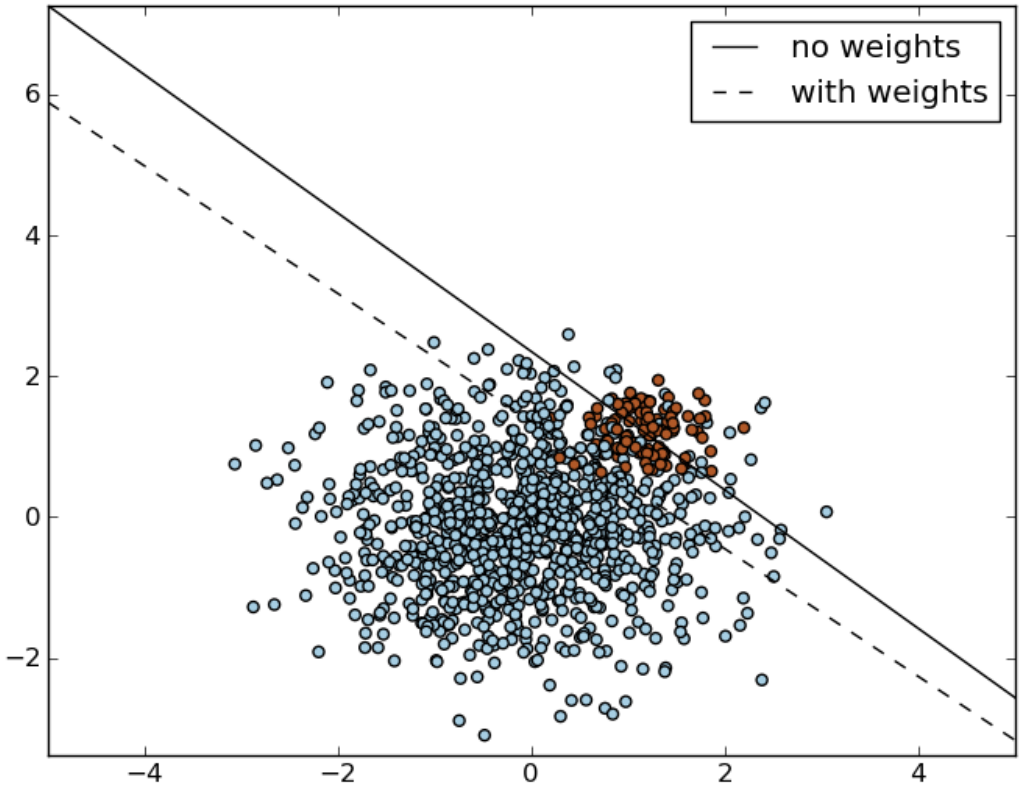
Python source code: `plot_sgd_separating_hyperplane.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn.linear_model import SGDClassifier

# we create 40 separable points
np.random.seed(0)
X = np.r_[np.random.randn(20, 2) - [2,2], np.random.randn(20, 2) + [2, 2]]
Y = [0]*20 + [1]*20

# fit the model
clf = SGDClassifier(loss="hinge", alpha = 0.01, n_iter=50,
                    fit_intercept=True)
clf.fit(X, Y)

# plot the line, the points, and the nearest vectors to the plane
xx = np.linspace(-5, 5, 10)
yy = np.linspace(-5, 5, 10)
X1, X2 = np.meshgrid(xx, yy)
Z = np.empty(X1.shape)
for (i,j), val in np.ndenumerate(X1):
    x1 = val
    x2 = X2[i,j]
```

```
    p = clf.decision_function([x1, x2])
    Z[i,j] = p[0]
levels = [-1.0, 0.0, 1.0]
linestyles = ['dashed','solid', 'dashed']
colors = 'k'
pl.set_cmap(pl.cm.Paired)
pl.contour(X1, X2, Z, levels, colors=colors, linestyles=linestyles)
pl.scatter(X[:,0], X[:,1], c=Y)

pl.axis('tight')
pl.show()
```

SGD: Separating hyperplane with weighted classes

Fit linear SVMs with and without class weighting. Allows to handle problems with unbalanced classes.



Python source code: `plot_sgd_weighted_classes.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn.linear_model import SGDClassifier
```

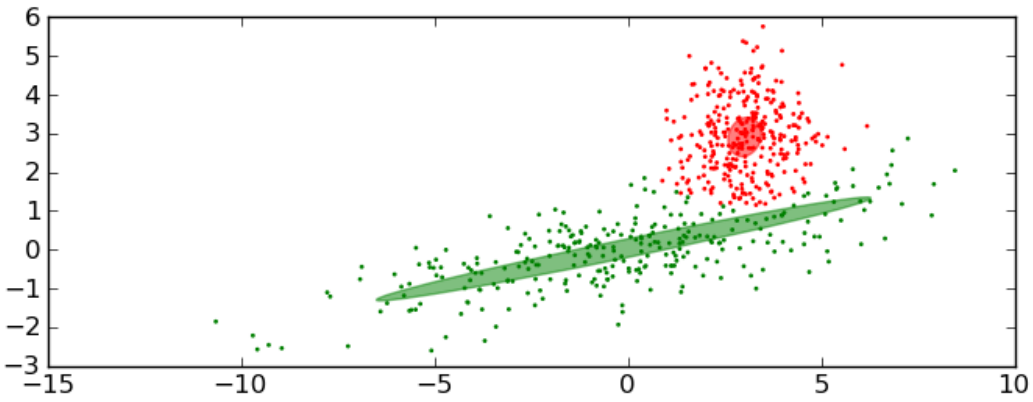
```
# we create 40 separable points
np.random.seed(0)
n_samples_1 = 1000
n_samples_2 = 100
X = np.r_[1.5*np.random.randn(n_samples_1, 2),
          0.5*np.random.randn(n_samples_2, 2) + [2, 2]]
y = np.array([0]*(n_samples_1) + [1]*(n_samples_2), dtype=np.float64)
idx = np.arange(y.shape[0])
np.random.shuffle(idx)
X = X[idx]
y = y[idx]
mean = X.mean(axis=0)
std = X.std(axis=0)
X = (X - mean) / std

# fit the model and get the separating hyperplane
clf = SGDClassifier(n_iter=100, alpha=0.01)
clf.fit(X, y)

w = clf.coef_
a = -w[0] / w[1]
xx = np.linspace(-5, 5)
yy = a * xx - clf.intercept_ / w[1]


# get the separating hyperplane using weighted classes
wclf = SGDClassifier(n_iter=100, alpha=0.01)
wclf.fit(X, y, class_weight={1: 10})

ww = wclf.coef_
wa = -ww[0] / ww[1]
wyy = wa * xx - wclf.intercept_ / ww[1]

# plot separating hyperplanes and samples
pl.set_cmap(pl.cm.Paired)
h0 = pl.plot(xx, yy, 'k-')
h1 = pl.plot(xx, wyy, 'k--')
pl.scatter(X[:,0], X[:,1], c=y)
pl.legend((h0, h1), ('no weights', 'with weights'))

pl.axis('tight')
pl.show()
```

2.1.6 Gaussian Mixture Models

Examples concerning the *scikits.learn.mixture* package.

Gaussian Mixture Model Ellipsoids

Plot the confidence ellipsoids of a mixture of two gaussians.



Python source code: `plot_gmm.py`

```
import numpy as np
from scikits.learn import mixture
import itertools

import pylab as pl
import matplotlib as mpl

n, m = 300, 2

# generate random sample, two components
np.random.seed(0)
C = np.array([[0., -0.7], [3.5, .7]])
X = np.r_[np.dot(np.random.randn(n, 2), C),
          np.random.randn(n, 2) + np.array([3, 3])]

clf = mixture.GMM(n_states=2, cvtype='full')
clf.fit(X)

splot = pl.subplot(111, aspect='equal')
color_iter = itertools.cycle (['r', 'g', 'b', 'c'])

Y_ = clf.predict(X)

for i, (mean, covar, color) in enumerate(zip(clf.means, clf.covars, color_iter)):
```

```
    v, w = np.linalg.eigh(covar)
    u = w[0] / np.linalg.norm(w[0])
    pl.scatter(X[Y_==i, 0], X[Y_==i, 1], .8, color=color)
    angle = np.arctan(u[1]/u[0])
    angle = 180 * angle / np.pi # convert to degrees
    ell = mpl.patches.Ellipse (mean, v[0], v[1], 180 + angle, color=color)
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(0.5)
    splot.add_artist(ell)

pl.show()
```

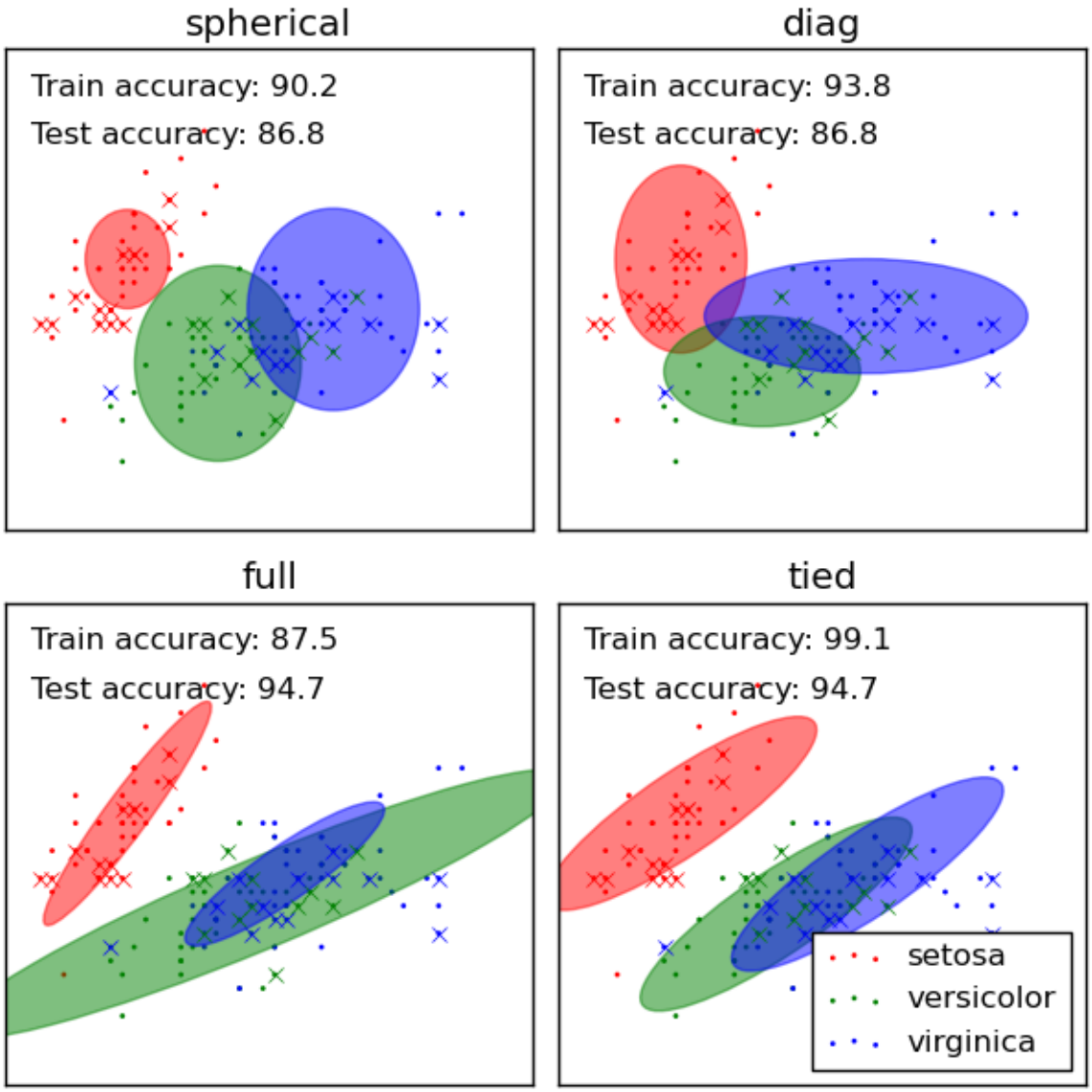
GMM classification

Demonstration of *gmm* for classification.

Plots predicted labels on both training and held out test data using a variety of GMM classifiers on the iris dataset.

Compares GMMs with spherical, diagonal, full, and tied covariance matrices in increasing order of performance. Although one would expect full covariance to perform best in general, it is prone to overfitting on small datasets and does not generalize well to held out test data.

On the plots, train data is shown as dots, while test data is shown as crosses. The iris dataset is four-dimensional. Only the first two dimensions are shown here, and thus some points are separated in other dimensions.



Python source code: `plot_gmm_classifier.py`

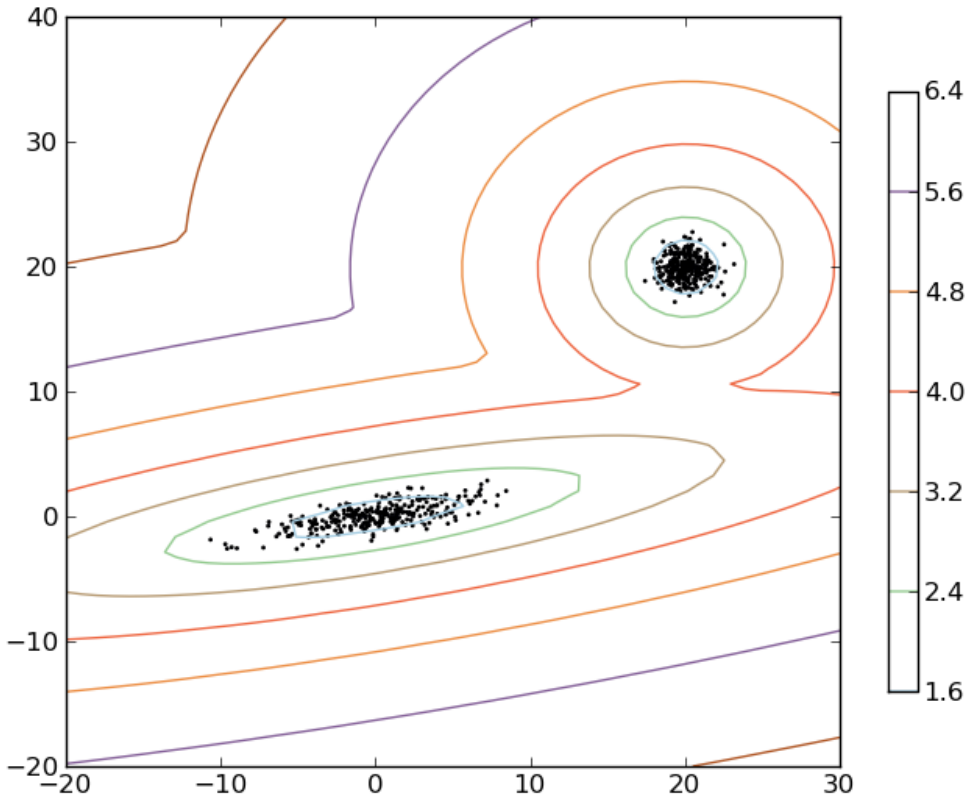
```
print __doc__

# Author: Ron Weiss <ronweiss@gmail.com>, Gael Varoquaux
# License: BSD Style.

# $Id$

import pylab as pl
import matplotlib as mpl
import numpy as np

from scikits.learn import datasets
from scikits.learn.cross_val import StratifiedKFold
from scikits.learn.mixture import GMM

def make_ellipses(gmm, ax):
    for n, color in enumerate('rgb'):
```

```
        v, w = np.linalg.eigh(gmm.covars[n][:2, :2])
        u = w[0] / np.linalg.norm(w[0])
        angle = np.arctan(u[1]/u[0])
        angle = 180 * angle / np.pi # convert to degrees
        v *= 9
        ell = mpl.patches.Ellipse(gmm.means[n, :2], v[0], v[1], 180 + angle,
                                  color=color)
        ell.set_clip_box(ax.bbox)
        ell.set_alpha(0.5)
        ax.add_artist(ell)

iris = datasets.load_iris()

# Break up the dataset into non-overlapping training (75%) and testing
# (25%) sets.
skf = StratifiedKFold(iris.target, k=4)
# Only take the first fold.
train_index, test_index = skf.__iter__().next()


X_train = iris.data[train_index]
y_train = iris.target[train_index]
X_test = iris.data[test_index]
y_test = iris.target[test_index]

n_classes = len(np.unique(y_train))

# Try GMMs using different types of covariances.
classifiers = dict((x, GMM(n_states=n_classes, cvtype=x))
                   for x in ['spherical', 'diag', 'tied', 'full'])

n_classifiers = len(classifiers)

pl.figure(figsize=(3*n_classifiers/2, 6))
pl.subplots_adjust(bottom=.01, top=0.95, hspace=.15, wspace=.05,
                   left=.01, right=.99)


for index, (name, classifier) in enumerate(classifiers.iteritems()):
    # Since we have class labels for the training data, we can
    # initialize the GMM parameters in a supervised manner.
    classifier.means = [X_train[y_train == i, :].mean(axis=0)
                        for i in xrange(n_classes)]

    # Train the other parameters using the EM algorithm.
    classifier.fit(X_train, init_params='wc', n_iter=20)

    h = pl.subplot(2, n_classifiers/2, index + 1)
    make_ellipses(classifier, h)

    for n, color in enumerate('rgb'):
        data = iris.data[iris.target == n]
        pl.scatter(data[:,0], data[:, 1], 0.8, color=color,
                   label=iris.target_names[n])
    # Plot the test data with crosses
    for n, color in enumerate('rgb'):
        data = X_test[y_test == n]
        pl.plot(data[:, 0], data[:, 1], 'x', color=color)
```

```
    y_train_pred = classifier.predict(X_train)
    train_accuracy  = np.mean(y_train_pred.ravel() == y_train.ravel()) * 100
    pl.text(0.05, 0.9, 'Train accuracy: %.1f' % train_accuracy,
                    transform=h.transAxes)

    y_test_pred = classifier.predict(X_test)
    test_accuracy  = np.mean(y_test_pred.ravel() == y_test.ravel()) * 100
    pl.text(0.05, 0.8, 'Test accuracy: %.1f' % test_accuracy,
                    transform=h.transAxes)

    pl.xticks(())
    pl.yticks(())
    pl.title(name)

pl.legend(loc='lower right', prop=dict(size=12))


pl.show()
```

Density Estimation for a mixture of Gaussians

Plot the density estimation of a mixture of two gaussians. Data is generated from two gaussians with different centers and covariance matrices.



Python source code: `plot_gmm_pdf.py`

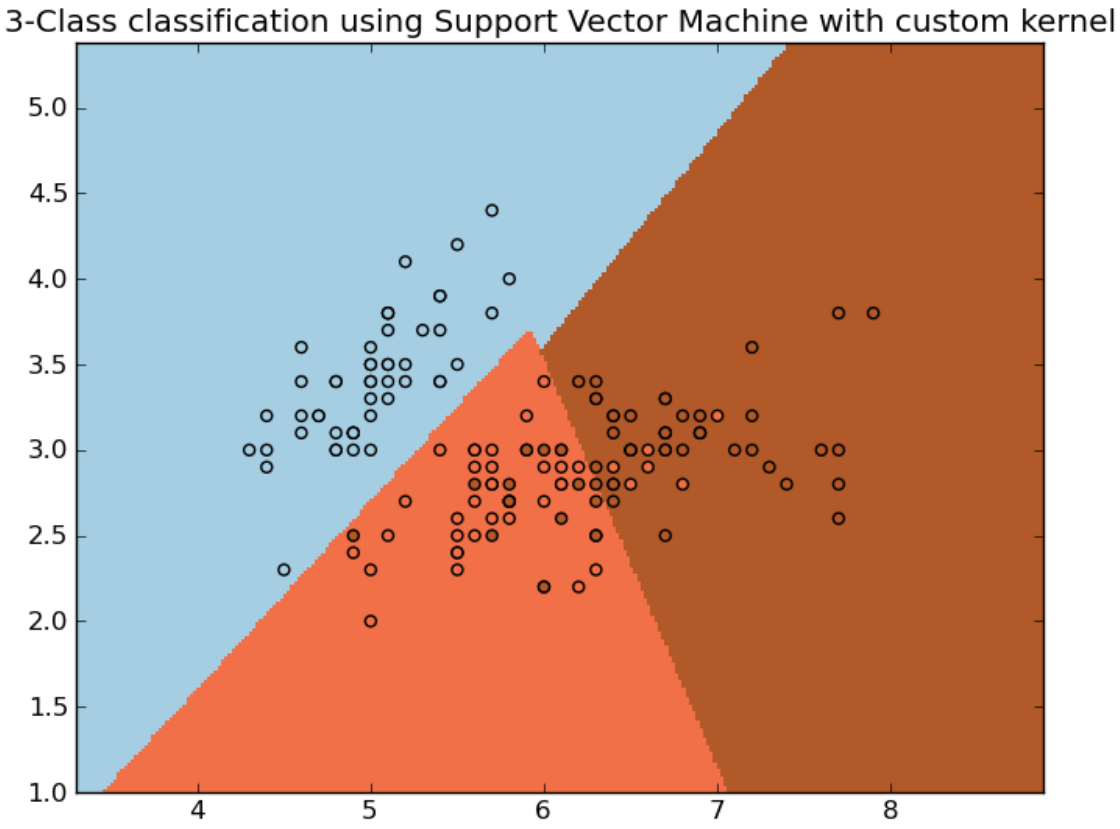
```
import numpy as np
import pylab as pl
from scikits.learn import mixture

n_samples = 300

# generate random sample, two components
np.random.seed(0)
C = np.array([[0., -0.7], [3.5, .7]])
X_train = np.r_[np.dot(np.random.randn(n_samples, 2), C),
          np.random.randn(n_samples, 2) + np.array([20, 20])]

clf = mixture.GMM(n_states=2, cvtype='full')
clf.fit(X_train)

x = np.linspace(-20.0, 30.0)
y = np.linspace(-20.0, 40.0)
X, Y = np.meshgrid(x, y)
XX = np.c_[X.ravel(), Y.ravel()]
Z =  np.log(-clf.eval(XX)[0])
Z = Z.reshape(X.shape)

CS = pl.contour(X, Y, Z)
CB = pl.colorbar(CS, shrink=0.8, extend='both')
pl.scatter(X_train[:, 0], X_train[:, 1], .8)

pl.axis('tight')
pl.show()
```

2.1.7 Support Vector Machines

Examples concerning the *scikits.learn.svm* package.

SVM with custom kernel

Simple usage of Support Vector Machines to classify a sample. It will plot the decision surface and the support vectors.



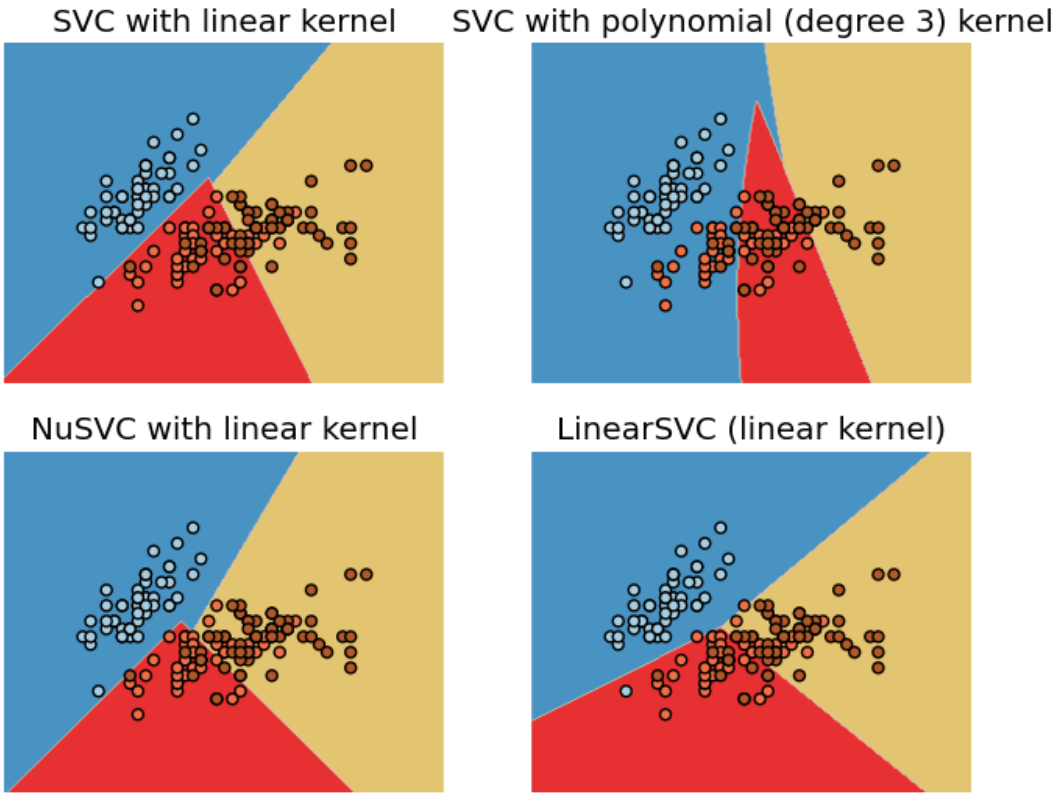
Python source code: `plot_custom_kernel.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn import svm, datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2] # we only take the first two features. We could
                     # avoid this ugly slicing by using a two-dim dataset
Y = iris.target


def my_kernel(x, y):
    """
    We create a custom kernel:

                 (2  0)
    k(x, y) = x  (    ) y.T
                 (0  1)
    """
    M = np.array([[2, 0], [0, 1.0]])
    return np.dot(np.dot(x, M), y.T)
```

```
h=.02 # step size in the mesh

# we create an instance of SVM and fit out data.
clf = svm.SVC(kernel=my_kernel)
clf.fit(X, Y)

# Plot the decision boundary. For that, we will asign a color to each
# point in the mesh [x_min, m_max]x[y_min, y_max].
x_min, x_max = X[:,0].min()-1, X[:,0].max()+1
y_min, y_max = X[:,1].min()-1, X[:,1].max()+1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
pl.set_cmap(pl.cm.Paired)
pl.pcolormesh(xx, yy, Z)

# Plot also the training points
pl.scatter(X[:,0], X[:,1], c=Y)
pl.title('3-Class classification using Support Vector Machine with custom kernel')
pl.axis('tight')
pl.show()
```

Plot different SVM classifiers in the iris dataset

Comparison of different linear SVM classifiers on the iris dataset. It will plot the decision surface for four different SVM classifiers.



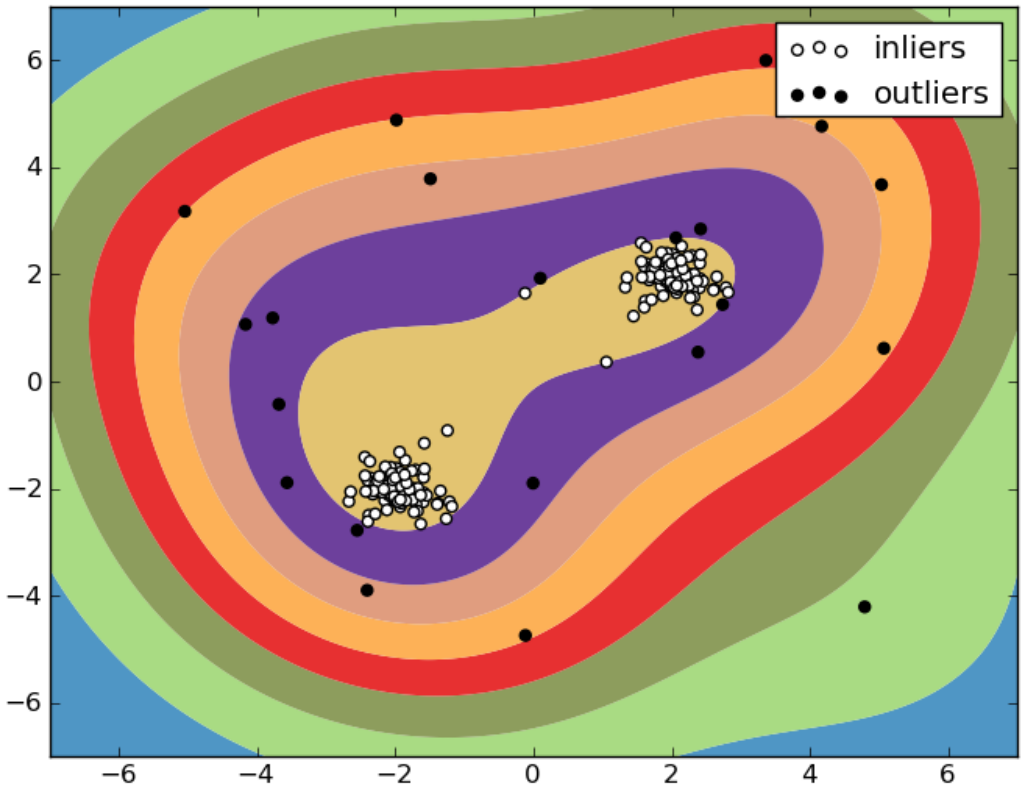
Python source code: `plot_iris.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn import svm, datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2] # we only take the first two features. We could
                     # avoid this ugly slicing by using a two-dim dataset
Y = iris.target

h=.02 # step size in the mesh

# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
svc     = svm.SVC(kernel='linear').fit(X, Y)
rbf_svc = svm.SVC(kernel='poly').fit(X, Y)
nu_svc  = svm.NuSVC(kernel='linear').fit(X,Y)
lin_svc = svm.LinearSVC().fit(X, Y)

# create a mesh to plot in
x_min, x_max = X[:,0].min()-1, X[:,0].max()+1
y_min, y_max = X[:,1].min()-1, X[:,1].max()+1
```

```
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

# title for the plots
titles = ['SVC with linear kernel',
          'SVC with polynomial (degree 3) kernel',
          'NuSVC with linear kernel',
          'LinearSVC (linear kernel)']


pl.set_cmap(pl.cm.Paired)

for i, clf in enumerate((svc, rbf_svc, nu_svc, lin_svc)):
    # Plot the decision boundary. For that, we will asign a color to each
    # point in the mesh [x_min, m_max]x[y_min, y_max].
    pl.subplot(2, 2, i+1)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    pl.set_cmap(pl.cm.Paired)
    pl.contourf(xx, yy, Z)
    pl.axis('off')

    # Plot also the training points
    pl.scatter(X[:,0], X[:,1], c=Y)

    pl.title(titles[i])

pl.show()
```

One-class SVM with non-linear kernel (RBF)

One-class SVM is an unsupervised algorithm that estimates outliers in a dataset.



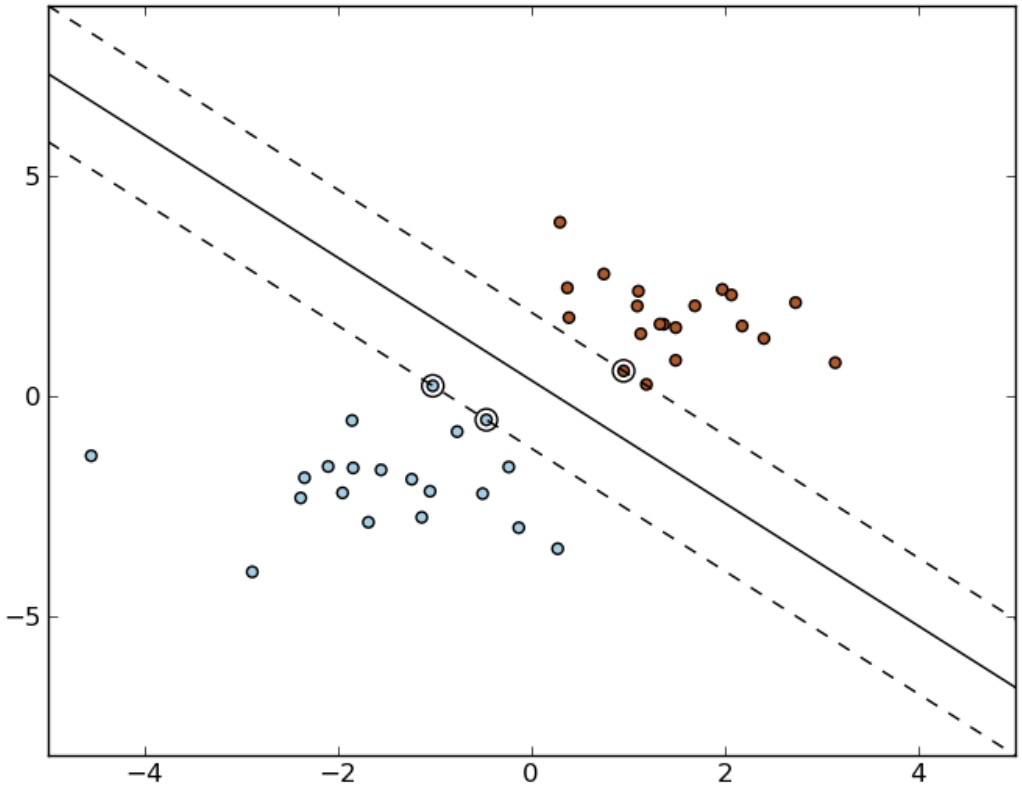
Python source code: `plot_oneclass.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn import svm

xx, yy = np.meshgrid(np.linspace(-7, 7, 500), np.linspace(-7, 7, 500))
X = 0.3 * np.random.randn(100, 2)
X = np.r_[X + 2, X - 2]

# Add 10 % of outliers (leads to nu=0.1)
X = np.r_[X, np.random.uniform(low=-6, high=6, size=(20, 2))]

# fit the model
clf = svm.OneClassSVM(nu=0.1, kernel="rbf", gamma=0.1)
clf.fit(X)

# plot the line, the points, and the nearest vectors to the plane
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
y_pred = clf.predict(X)

pl.set_cmap(pl.cm.Paired)
pl.contourf(xx, yy, Z)
```

```
pl.scatter(X[y_pred>0,0], X[y_pred>0,1], c='white', label='inliers')
pl.scatter(X[y_pred<=0,0], X[y_pred<=0,1], c='black', label='outliers')
pl.axis('tight')
pl.legend()
pl.show()
```

SVM: Maximum margin separating hyperplane

Plot the maximum margin separating hyperplane within a two-class separable dataset using a Support Vector Machines classifier with linear kernel.



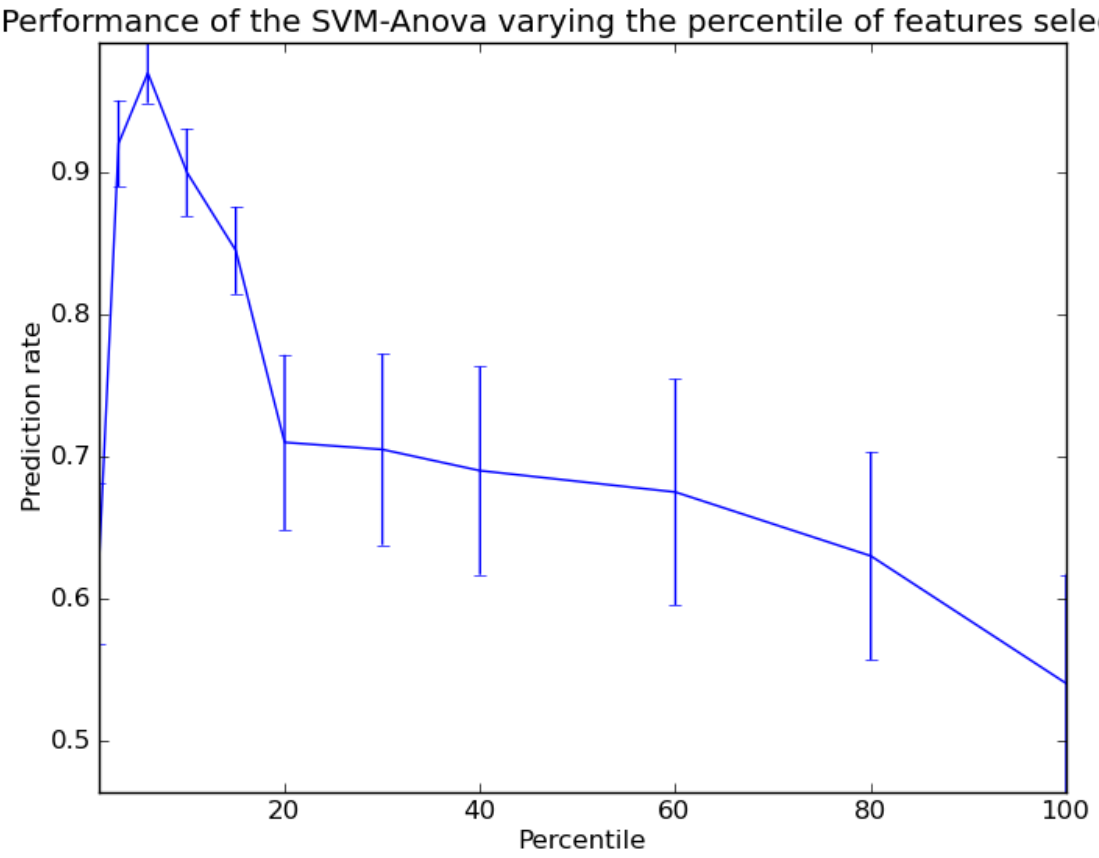
Python source code: `plot_separating_hyperplane.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn import svm

# we create 40 separable points
np.random.seed(0)
X = np.r_[np.random.randn(20, 2) - [2,2], np.random.randn(20, 2) + [2, 2]]
Y = [0]*20 + [1]*20

# fit the model
```

```
clf = svm.SVC(kernel='linear')
clf.fit(X, Y)

# get the separating hyperplane
w =  clf.coef_[0]
a = -w[0]/w[1]
xx = np.linspace(-5, 5)
yy = a*xx - (clf.intercept_[0])/w[1]

# plot the parallels to the separating hyperplane that pass through the
# support vectors
b = clf.support_vectors_[0]
yy_down = a*xx + (b[1] - a*b[0])
b = clf.support_vectors_[-1]
yy_up = a*xx + (b[1] - a*b[0])

# plot the line, the points, and the nearest vectors to the plane
pl.set_cmap(pl.cm.Paired)
pl.plot(xx, yy, 'k-')
pl.plot(xx, yy_down, 'k--')
pl.plot(xx, yy_up, 'k--')

pl.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
           s=80, facecolors='none')
pl.scatter(X[:,0], X[:,1], c=Y)

pl.axis('tight')
pl.show()
```

SVM-Anova: SVM with univariate feature selection

This example shows how to perform univariate feature before running a SVC (support vector classifier) to improve the classification scores.



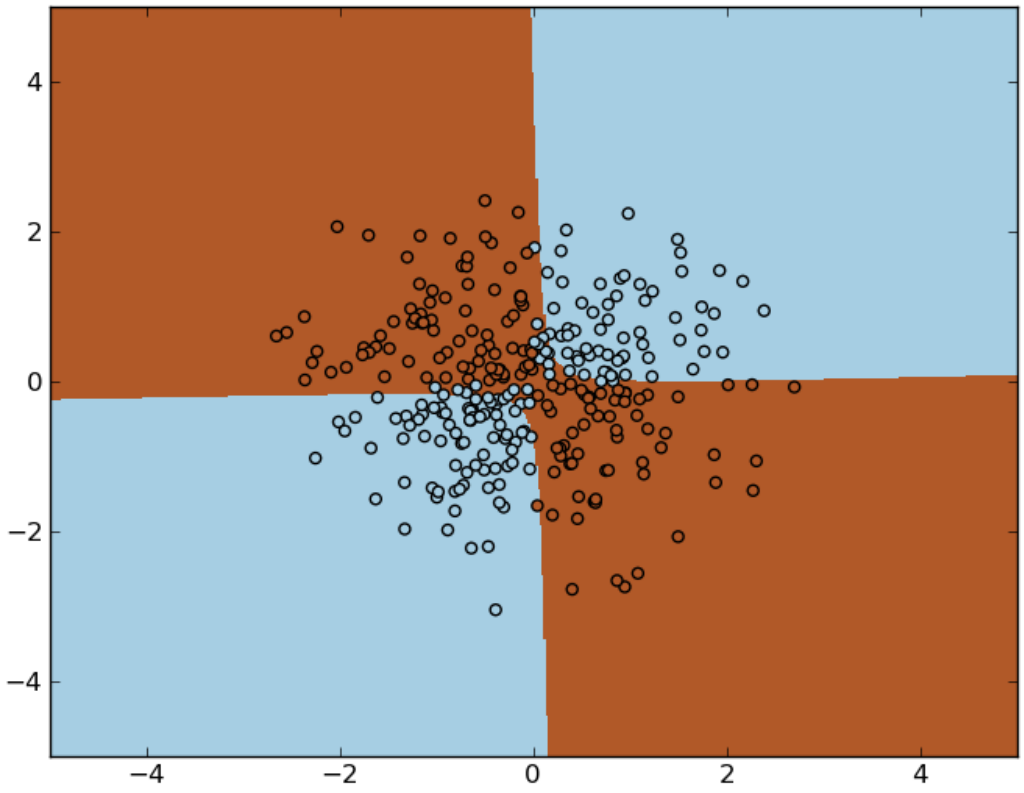
Python source code: `plot_svm_anova.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn import svm, datasets, feature_selection, cross_val
from scikits.learn.pipeline import Pipeline

###############################################################################
# Import some data to play with
digits = datasets.load_digits()
y = digits.target
# Throw away data, to be in the curse of dimension settings
y = y[:200]
X = digits.data[:200]
n_samples = len(y)
X = X.reshape((n_samples, -1))
# add 200 non-informative features
X = np.hstack((X, 2*np.random.random((n_samples, 200))))

###############################################################################
# Create a feature-selection transform and an instance of SVM that we
# combine together to have an full-blown estimator

transform = feature_selection.SelectPercentile(feature_selection.f_classif)
```

```
clf = Pipeline([('anova', transform), ('svc', svm.SVC())])

###############################################################################
# Plot the cross-validation score as a function of percentile of features
score_means = list()
score_stds  = list()
percentiles = (1, 3, 6, 10, 15, 20, 30, 40, 60, 80, 100)

for percentile in percentiles:
    clf._set_params(anova__percentile=percentile)
    # Compute cross-validation score using all CPUs
    this_scores = cross_val.cross_val_score(clf, X, y, n_jobs=1)
    score_means.append(this_scores.mean())
    score_stds.append(this_scores.std())

pl.errorbar(percentiles, score_means, np.array(score_stds))

pl.title(
    'Performance of the SVM-Anova varying the percentile of features selected')
pl.xlabel('Percentile')
pl.ylabel('Prediction rate')

pl.axis('tight')
pl.show()
```

Non-linear SVM

Perform binary classification using non-linear SVC with RBF kernel. The target to predict is a XOR of the inputs.



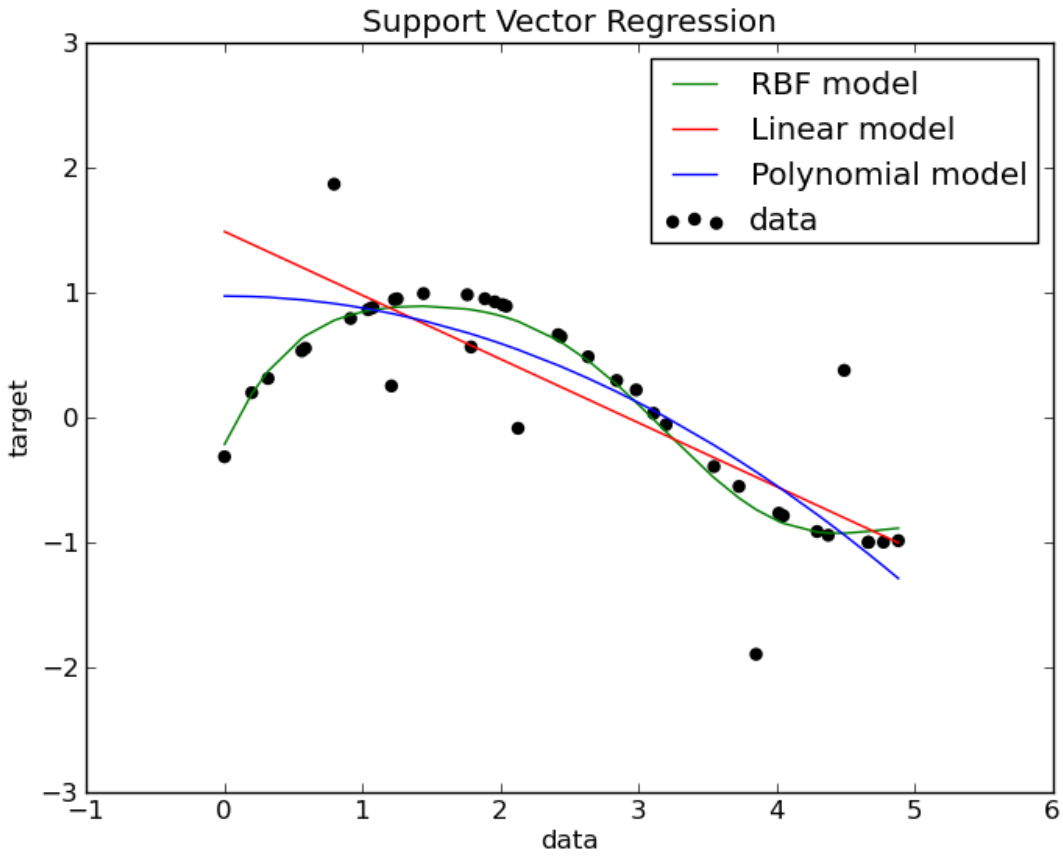
Python source code: `plot_svm_nonlinear.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn import svm

xx, yy = np.meshgrid(np.linspace(-5, 5, 500), np.linspace(-5, 5, 500))
np.random.seed(0)
X = np.random.randn(300, 2)
Y = np.logical_xor(X[:,0]>0, X[:,1]>0)

# fit the model
clf = svm.NuSVC()
clf.fit(X, Y)

# plot the line, the points, and the nearest vectors to the plane
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

pl.set_cmap(pl.cm.Paired)
pl.pcolormesh(xx, yy, Z)
pl.scatter(X[:,0], X[:,1], c=Y)

pl.axis('tight')
```

```
pl.show()
```

Support Vector Regression (SVR) using linear and non-linear kernels

Toy example of 1D regression using linear, polynominial and RBF kernels.



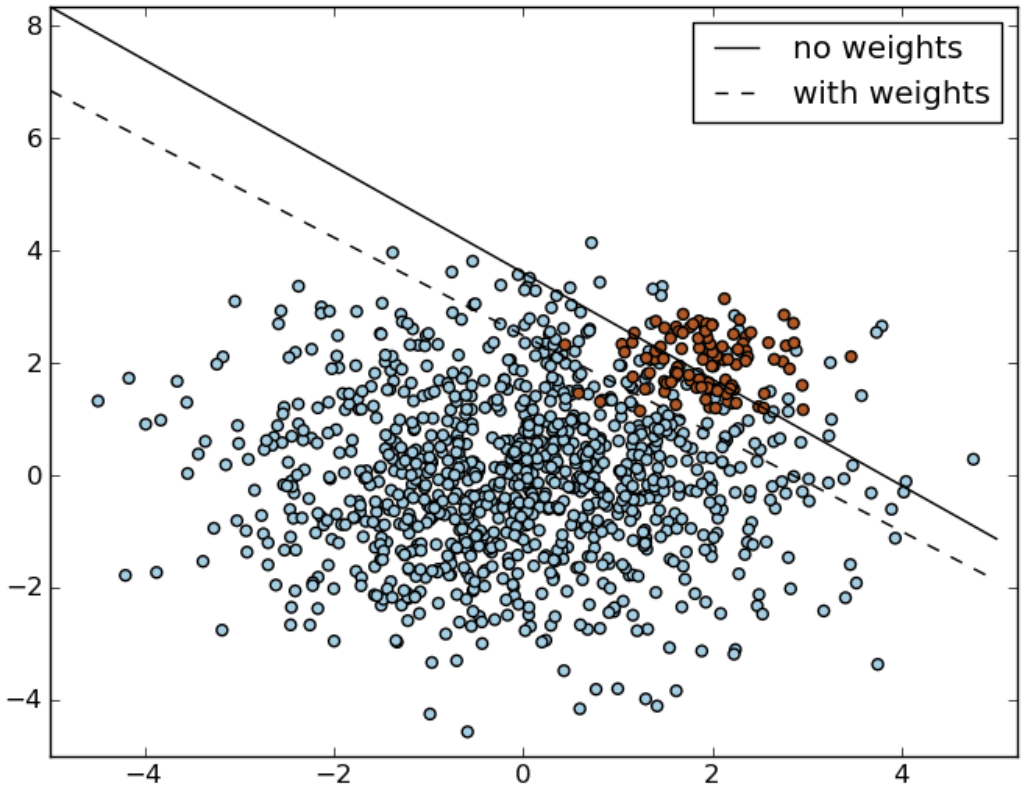
Python source code: `plot_svm_regression.py`

```
print __doc__

###############################################################################
# Generate sample data
import numpy as np

X = np.sort(5*np.random.rand(40, 1), axis=0)
y = np.sin(X).ravel()

###############################################################################
# Add noise to targets
y[::5] += 3*(0.5 - np.random.rand(8))

###############################################################################
# Fit regression model
from scikits.learn.svm import SVR
```

```
svr_rbf = SVR(kernel='rbf', C=1e4, gamma=0.1)
svr_lin = SVR(kernel='linear', C=1e4)
svr_poly = SVR(kernel='poly', C=1e4, degree=2)
y_rbf = svr_rbf.fit(X, y).predict(X)
y_lin = svr_lin.fit(X, y).predict(X)
y_poly = svr_poly.fit(X, y).predict(X)

##############################################################################
# look at the results
import pylab as pl
pl.scatter(X, y, c='k', label='data')
pl.hold('on')
pl.plot(X, y_rbf, c='g', label='RBF model')
pl.plot(X, y_lin, c='r', label='Linear model')
pl.plot(X, y_poly, c='b', label='Polynomial model')
pl.xlabel('data')
pl.ylabel('target')
pl.title('Support Vector Regression')
pl.legend()
pl.show()
```

SVM: Separating hyperplane with weighted classes

Fit linear SVMs with and without class weighting. Allows to handle problems with unbalanced classes.

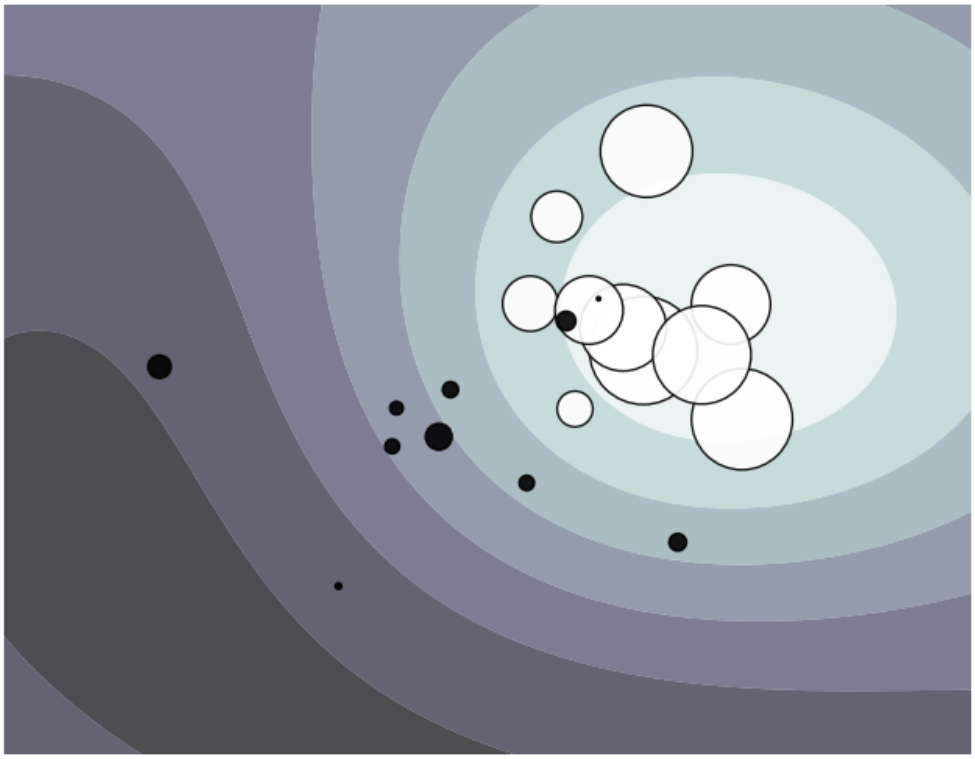


Python source code: `plot_weighted_classes.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn import svm

# we create 40 separable points
np.random.seed(0)
n_samples_1 = 1000
n_samples_2 = 100
X = np.r_[1.5*np.random.randn(n_samples_1, 2),
          0.5*np.random.randn(n_samples_2, 2) + [2, 2]]
y = [0]*(n_samples_1) + [1]*(n_samples_2)

# fit the model and get the separating hyperplane
clf = svm.SVC(kernel='linear')
clf.fit(X, y)

w = clf.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(-5, 5)
yy = a * xx - clf.intercept_[0] / w[1]


# get the separating hyperplane using weighted classes
wclf = svm.SVC(kernel='linear')
wclf.fit(X, y, class_weight={1: 10})

ww = wclf.coef_[0]
wa = -ww[0] / ww[1]
wyy = wa * xx - wclf.intercept_[0] / ww[1]

# plot separating hyperplanes and samples
pl.set_cmap(pl.cm.Paired)
h0 = pl.plot(xx, yy, 'k-')
h1 = pl.plot(xx, wyy, 'k--')
pl.scatter(X[:,0], X[:,1], c=y)
pl.legend((h0, h1), ('no weights', 'with weights'))

pl.axis('tight')
pl.show()
```

SVM: Weighted samples

Plot decision function of a weighted dataset, where the size of points is proportional to its weight.



Python source code: `plot_weighted_samples.py`

```
print __doc__

import numpy as np
import pylab as pl
from scikits.learn import svm

# we create 20 points
np.random.seed(0)
X = np.r_[np.random.randn(10, 2) + [1, 1], np.random.randn(10, 2)]
Y = [1]*10 + [-1]*10
sample_weight = 100 * np.abs(np.random.randn(20))
# and assign a bigger weight to the last 10 samples
sample_weight[:10] *= 10

# # fit the model
clf = svm.SVC()
clf.fit(X, Y, sample_weight=sample_weight)

# plot the decision function
xx, yy = np.meshgrid(np.linspace(-4, 5, 500), np.linspace(-4, 5, 500))

Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
```

```
# plot the line, the points, and the nearest vectors to the plane
pl.set_cmap(pl.cm.bone)
pl.contourf(xx, yy, Z, alpha=0.75)
pl.scatter(X[:, 0], X[:, 1], c=Y, s=sample_weight, alpha=0.9)

pl.axis('off')
pl.show()
```

CHAPTER

THREE

DEVELOPMENT

3.1 Contributing

This project is a community effort, and everyone is welcomed to contribute.

3.1.1 Submitting a bug report

In case you experience difficulties using the package, do not hesitate to submit a ticket to the Bug Tracker.

You are also welcomed to post there feature requests and patches.

3.1.2 Retrieving the latest code

You can check the latest sources with the command:

```
git clone git://github.com/scikit-learn/scikit-learn.git
```

or if you have write privileges:

```
git clone git@github.com:scikit-learn/scikit-learn.git
```

You can also check out the sources online in the web page http://github.com/scikit-learn/scikit-learn

If you run the development version, it is cumbersome to re-install the package each time you update the sources. It is thus preferred that you add the scikit-directory to your PYTHONPATH and build the extension in place:

```
python setup.py build_ext --inplace
```

3.1.3 Contributing code

How to contribute

The prefered way to contribute to *scikit-learn* is to fork the main repository on github:

1. Create an account on github if you don't have one already.
2. Fork the scikit-learn repo: click on the 'Fork' button, at the top, center of the page. This creates a copy of the code on the github server where you can work.
3. Clone this copy to your local disk (you need the *git* program to do this):

```
$ git clone git@github.com:YourLogin/scikit-learn.git
```

4. Work on this copy, on your computer, using git to do the version control:

```
$ git add modified_files
$ git commit
$ git push origin master
```

   and so on.

When you are ready, and you have pushed your changes on your github repo, go the web page of the repo, and click on 'Pull request' to send us a pull request. Send us a mail with your pull request, and we can look at your changes, and integrate them.

Before asking for a pull or a review, be sure to read the coding-guidelines (below).

Also, make sure that your code is tested, and that all the tests for the scikit pass.

EasyFix Issues

The best way to get your feet wet is to pick up an issue from the issue tracker that are labeled as EasyFix. This means that the knowledge needed to solve the issue is low, but still you are helping the project and letting more experienced developers concentrate on other issues.

Roadmap

Here you will find a detailed roadmap, with a description on what's planned to be implemented in the following releases.

Documentation

We are glad to accept any sort of documentation: function docstrings, rst docs (like this one), tutorials, etc. Rst docs live in the source code repository, under directory doc/.

You can edit them using any text editor and generate the html docs by typing from the doc/ directory `make html` (or `make html-noplot`, see README in that directory for more info). That should create a directory _build/html/ with html files that are viewable in a web browser.

Developers web site

More information can be found at the developer's wiki.

3.1.4 Other ways to contribute

Code is not the only way to contribute to this project. For instance, documentation is also a very important part of the project and ofter doesn't get as much attention as it deserves. If you find a typo in the documentation, or have made improvements, don't hesitate to send an email to the mailing list or a github pull request. Full documentation can be found under directory doc/.

It also helps us if you spread the word: reference it from your blog, articles, link to us from your website, or simply by saying "I use it":

3.1.5 Coding guidelines

The following are some guidelines on how new code should be written. Of course, there are special cases and there will be exceptions to these rules. However, following these rules when submitting new code makes the review easier so new code can be integrated in less time.

Uniformly formated code makes it easier to share code ownership. The scikit learn tries to follow closely the officiel Python guidelines detailed in PEP8 that details how code should be formatted, and indented. Please read it and follow it.

In addition, we add the following guidelines:

- Use underscores to separate words in non class names: *n_samples* rather than *nsamples*.
- Avoid multiple statements on one line. Prefer a line return after a control flow statement (*if*/*for*).
- Use relative imports for references inside scikits.learn.
- **Please don't use 'import *' in any case**. It is considered harmful by the official Python recommandations. It makes the code harder to read as the origin of symbols is no longer explicitely referenced, but most important, it prevents using a static analysis tool like pyflakes to automatically find bugs in the scikit.

A good example of code that we like can be found here.

3.1.6 APIs of scikit learn objects

To have a uniform API, we try to have a common basic API for all the objects. In addition, to avoid the proliferation of framework code, we try to adopt simple conventions and limit to a minimum the number of methods an object has to implement.

Different objects

The main objects of the scikit learn are (one class can implement multiple interfaces):

Estimator The base object, implements:

```
estimator = obj.fit(data)
```

Predictor For suppervised learning, or some unsupervised problems, implements:

```
prediction = obj.predict(data)
```

Transformer For filtering or modifying the data, in a supervised or unsupervised way, implements:

```
new_data = obj.transform(data)
```

When fitting and transforming can be performed much more efficiently together than separately, implements:

new_data = obj.fit_transform(data)

Model A model that can give a goodness of fit or a likelihood of unseen data, implements (higher is better):

```
score = obj.score(data)
```

Estimators

The API has one predominant object: the estimator. A estimator is an object that fits a model based on some training data and is capable of inferring some properties on new data. It can be for instance a classifier or a regressor. All estimators implement the fit method:

```
estimator.fit(X, y)
```

Instantiation

This concerns the object creation. The object's __init__ method might accept as arguments constants that determine the estimator behavior (like the C constant in SVMs).

It should not, however, take the actual training data as argument, as this is left to the `fit()` method:

```
clf2 = SVC(C=2.3)
clf3 = SVC([[1, 2], [2, 3]], [-1, 1]) # WRONG!
```

The arguments that go in the *__init__* should all be keyword arguments with a default value. In other words, a user should be able to instanciate an estimator without passing to it any arguments.

The arguments in given at instanciation of an estimator should all correspond to hyper parameters describing the model or the optimisation problem that estimator tries to solve. They should however not be parameters of the estimation routine: these are passed directly to the *fit* method.

In addition, **every keyword argument given to the '__init__' should correspond to an attribute on the instance**. The scikit relies on this to find what are the relevent attributes to set on an estimator when doing model selection.

All estimators should inherit from *scikit.learn.base.BaseEstimator*

Fitting

The next thing you'll probably want to do is to estimate some parameters in the model. This is implemented in the .fit() method.

The fit method takes as argument the training data, which can be one array in the case of unsupervised learning, or two arrays in the case of supervised learning.

Note that the model is fitted using X and y but the object holds no reference to X, y. There are however some exceptions to this, as in the case of precomputed kernels where you need to store access these data in the predict method.

Parameters

- X : array-like, with shape = [N, D], where N is the number of samples and D is the number of features.
- Y : array, with shape = [N], where N is the number of samples.
- args, kwargs. Parameters can also be set in the fit method.

X.shape[0] should be the same as Y.shape[0]. If this requisite is not met, an exception should be raised.

Y might be dropped in the case of unsupervised learning.

The method should return the object (self).

Python tuples

In addition to numpy arrays, all methods should be able to accept python tuples as arguments. In practice, this means you should call numpy.asanyarray at the beginning at each public method that accepts arrays.

Optional Arguments

In iterative algorithms, number of iterations should be specified by an int called `n_iter`.

Unresolved API issues

Some things are must still be decided:

- what should happen when predict is called before than fit() ?
- which exception should be raised when arrays' shape do not match in fit() ?

Specific models

In linear models, coefficients are stored in an array called `coef_`, and independent term is stored in `intercept_`.

3.2 scikits.learn.neighbors working notes

3.2.1 barycenter

Function `barycenter()` tries to find appropriate weights to reconstruct the point x from a subset (y1, y2, ..., yn), where weights sum to one.

This is just a simple case of Equality Constrained Least Squares [1] with constrain dot(np.ones(n), x) = 1. In particular, the Q matrix from the QR decomposition of B is the Householder reflection of np.ones(n).

Purpose

This method was added to ease some computations in the future manifold module, namely in LLE. However, it is still to be shown that it is useful and efficient in that context.

Performance

The algorithm has to iterate over n_samples, which is the main bottleneck. It would be great to vectorize this loop. Also, the rank updates could probably be moved outside the loop.

Also, least squares solution could be computed more efficiently by a QR factorization, since probably we don't care about a minimum norm solution for the undertermined case.

The paper 'An introduction to Locally Linear Embeddings', Saul & Roweis solves the problem by the normal equation method over the covariance matrix. However, it does not degrade grathefully when the covariance is singular, requiring to explicitly add regularization.

[1] Section 12.1.4 ('Equality Constrained Least Squares'), 'Matrix Computations' by Golub & Van Loan

Stability

Should be good as it uses SVD to solve the LS problem. TODO: explicit bounds.

API

The API is convenient to use from NeighborsBarycenter and kneighbors_graph, but might not be very easy to use directly due to the fact that Y must be a 3-D array.

It should be checked that it is usable in other contexts.

TODO

3.3 About us

This is a community effort, and as such many people have contributed to it over the years.

3.3.1 History

This project was started in 2007 as a Google Summer of Code project by David Cournapeau. Later that year, Matthieu Brucher started work on this project as part of his thesis.

In 2010 Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort and Vincent Michel took leadership of the project and made the first public release, February the 1st 2010. Since then, several releases have appeard following a ~3 month cycle.

3.3.2 People

- David Cournapeau, 2007-2009
- Fred Mailhot, 2008, Artificial Neural Networks (ann) module.
- David Cooke
- David Huard
- Dave Morrill
- Ed Schofield
- Eric Jones, 2008, Genetic Algorithms (ga) module. No longer part of scikits.learn.
- Jarrod Millman
- Matthieu Brucher contributed the manifold module. It is not currently part of the scikit, althouth it is planned to be newly included in the upcoming 0.7 release.
- Travis Oliphant
- Pearu Peterson
- Fabian Pedregosa joined the project in January 2010 and is the current maintainer.
- Gael Varoquaux
- Jake VanderPlas contributed the BallTree module in February 2010.
- Alexandre Gramfort

- Olivier Grisel
- Vincent Michel.
- Chris Filo Gorgolewski
- Angel Soler Gollonet contributed the official logo and web page layout.
- Yaroslav Halchenko is the maintainer for Debian OS and has contributed several fixes.
- Ron Weiss joined the project in July 2010 and contributed both the mixture and hmm module.
- Virgile Fritsch. Bug fixes.
- Mathieu Blondel joined the project in September 2010 and has worked since on the sparse matrix support, Ridge generalized crossval, text feature extraction and general bug fixes.
- Peter Prettenhofer joined the project in October 2010 and contributed the *Stochastic Gradient Descent* module as well as several examples and fixes.
- Vincent Dubourg joined the project in November 2010 and contributed the *Gaussian Processes* module.
- Alexandre Passos joined the project in November 2010 contributed the fast SVD variant.

If I forgot anyone, do not hesitate to send me an email to fabian.pedregosa@inria.fr and I'll include you in the list.

3.3.3 Funding

INRIA actively supports this project. It has providen funding for Fabian Pedregosa to work on this project full time in the period 2010-2012. It also hosts coding sprints and other events.

Google sponsored David Cournapeau with a Summer of Code Scolarship in the summer of 2007. If you would like to participate in the next Google Summer of code program, please see this page

The NeuroDebian project providing Debian packaging and contributions is supported by Dr. James V. Haxby (Dartmouth College).

INDEX

Symbols

A

B

C

D

E

F

G

I

K

Q

R

S

T

W

Z