1.1. 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.

$$\hat{y}(w, x) = w_0 + w_1 x_1 + ... + w_p x_p$$

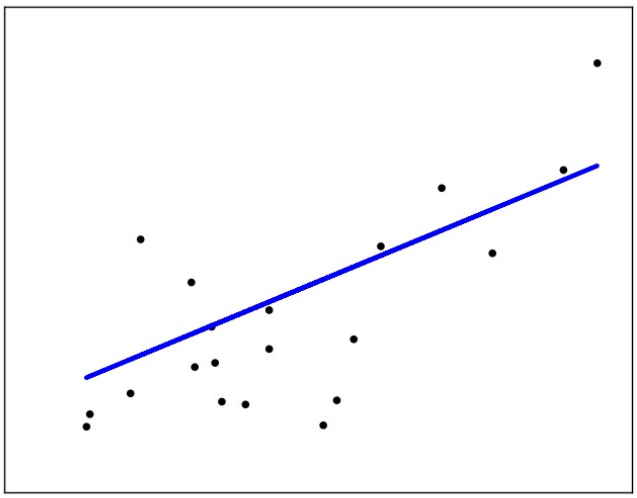
Across the module, we designate the vector $w = (w_1, ..., w_p)$ as `coef_` and w_0 as `intercept_`.

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

1.1.1. Ordinary Least Squares

`LinearRegression` fits a linear model with coefficients $w = (w_1, ..., w_p)$ to minimize the residual sum of squares between the observed responses in the dataset, and the responses predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\min_w ||Xw - y||_2^2$$



`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 sklearn import linear_model
>>> clf = linear_model.LinearRegression()
>>> clf.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
>>> 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 design matrix 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:

- *Linear Regression Example*

1.1.1.1. Ordinary Least Squares Complexity

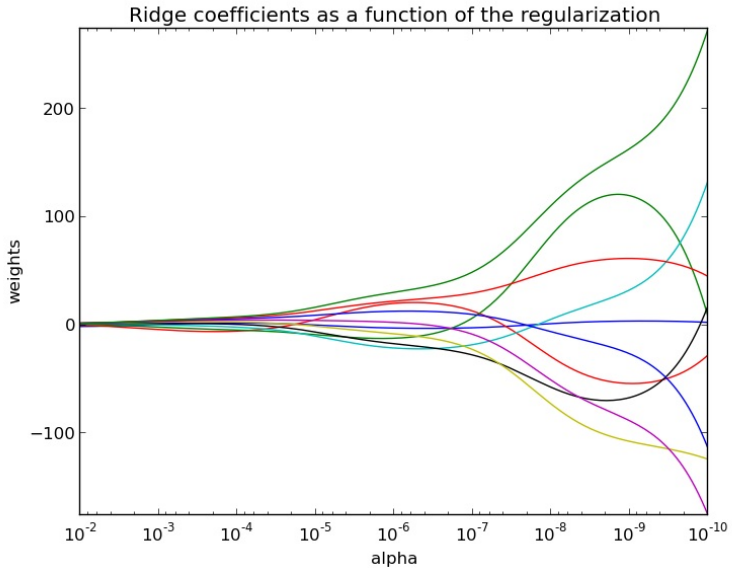
This method computes the least squares solution using a singular value decomposition of X. If X is a matrix of size (n, p) this method has a cost of $O(np^2)$, assuming that $n \geq p$.

1.1.2. Ridge Regression

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

$$\min_w ||Xw - y||_2^2 + \alpha ||w||_2^2$$

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 and thus the coefficients become more robust to collinearity.



As with other linear models, `Ridge` will take in its `fit` method arrays X, y and will store the coefficients w of the linear model in its `coef_` member:

```
>>> from sklearn import linear_model
>>> clf = linear_model.Ridge (alpha = .5)
>>> clf.fit ([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
Ridge(alpha=0.5, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, solver='auto', tol=0.001)
>>> clf.coef_
array([ 0.34545455,  0.34545455])
>>> clf.intercept_
0.13636...
```

Examples:

- *Plot Ridge coefficients as a function of the regularization*
- *Classification of text documents using sparse features*

1.1.2.1. Ridge Complexity

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

1.1.2.2. Setting the regularization parameter: 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 sklearn 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=[0.1, 1.0, 10.0], cv=None, fit_intercept=True, scoring=None,
    normalize=False)
>>> clf.alpha_
0.1
```

References

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

1.1.3. Lasso

The `Lasso` is a linear model that estimates sparse coefficients. It 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. Under certain conditions, it can recover the exact set of non-zero weights (see *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*).

Mathematically, it consists of a linear model trained with ℓ_1 prior as regularizer. The objective function to minimize is:

$$\min_w \frac{1}{2n_{samples}}||Xw - y||_2^2 + \alpha||w||_1$$

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 ℓ_1-norm of the parameter vector.

The implementation in the class `Lasso` 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, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute='auto', tol=0.0001,
   warm_start=False)
>>> clf.predict([[1, 1]])
array([ 0.8])
```

Also useful for lower-level tasks is the function `lasso_path` that computes the coefficients along the full path of possible values.

Examples:

- *Lasso and Elastic Net for Sparse Signals*
- *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*

Note: Feature selection with Lasso

As the Lasso regression yields sparse models, it can thus be used to perform feature selection, as detailed in *L1-based feature selection*.

Note: Randomized sparsity

For feature selection or sparse recovery, it may be interesting to use *Randomized sparse models*.

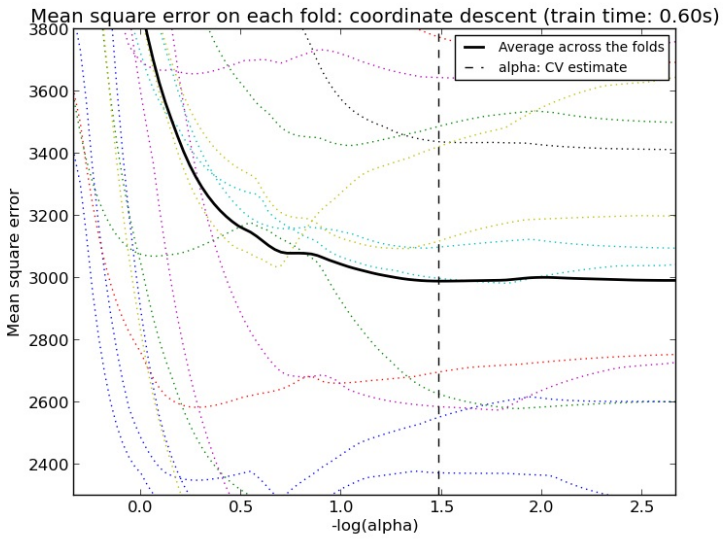
1.1.3.1. Setting regularization parameter

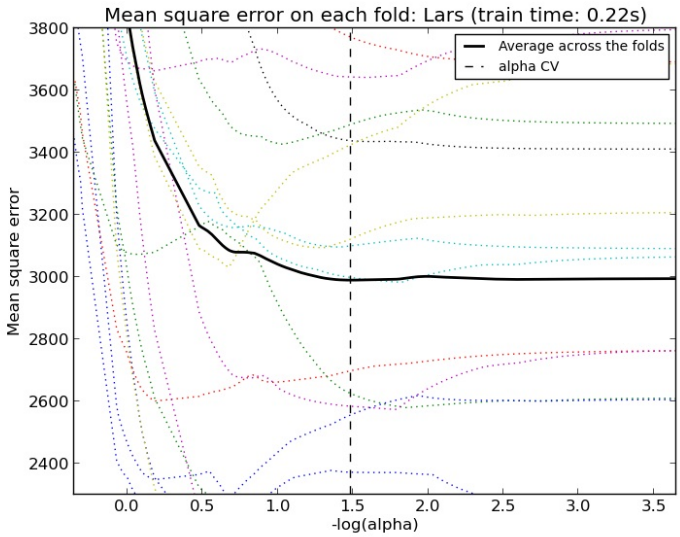
The `alpha` parameter controls the degree of sparsity of the coefficients estimated.

1.1.3.1.1. Using cross-validation

scikit-learn exposes objects that set the Lasso `alpha` parameter by cross-validation: `LassoCV` and `LassoLarsCV`. `LassoLarsCV` is based on the *Least Angle Regression* algorithm explained below.

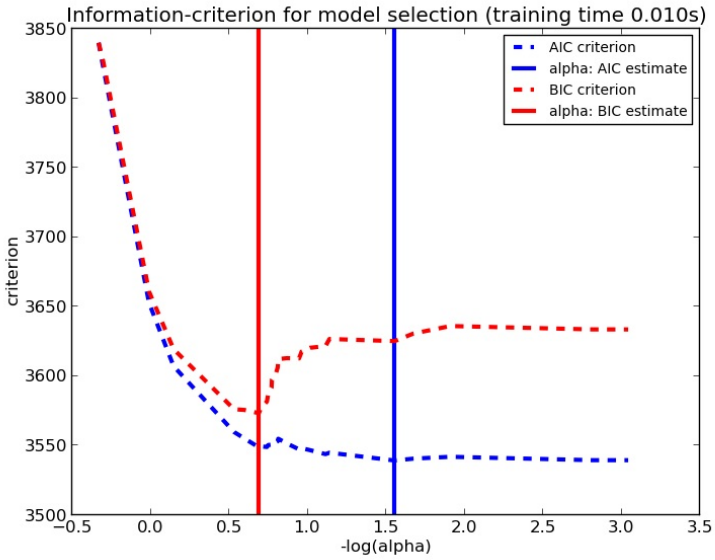
For high-dimensional datasets with many collinear regressors, `LassoCV` is most often preferable. How, `LassoLarsCV` has the advantage of exploring more relevant values of *alpha* parameter, and if the number of samples is very small compared to the number of observations, it is often faster than `LassoCV`.





1.1.3.1.2. Information-criteria based model selection

Alternatively, the estimator **LassoLarsIC** proposes to use the Akaike information criterion (AIC) and the Bayes Information criterion (BIC). It is a computationally cheaper alternative to find the optimal value of alpha as the regularization path is computed only once instead of k+1 times when using k-fold cross-validation. However, such criteria needs a proper estimation of the degrees of freedom of the solution, are derived for large samples (asymptotic results) and assume the model is correct, i.e. that the data are actually generated by this model. They also tend to break when the problem is badly conditioned (more features than samples).



Examples:

- *Lasso model selection: Cross-Validation / AIC / BIC*

1.1.4. Elastic Net

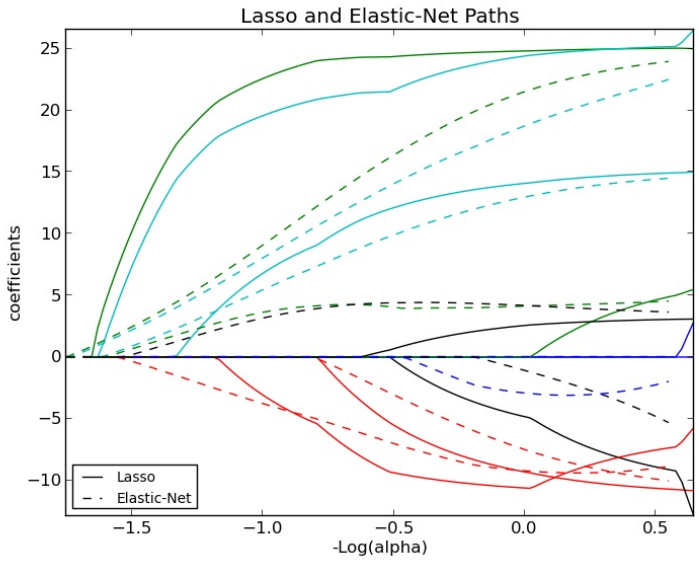
ElasticNet is a linear regression model trained with L1 and L2 prior as regularizer. This combination allows for learning a sparse model where few of the weights are non-zero like **Lasso**, while still maintaining the regularization properties of **Ridge**. We control the convex combination of L1 and L2 using the `l1_ratio` parameter.

Elastic-net is useful when there are multiple features which are correlated with one another. Lasso is likely to pick one of these at random, while elastic-net is likely to pick both.

A practical advantage of trading-off between Lasso and Ridge is it allows Elastic-Net to inherit some of Ridge's stability under rotation.

The objective function to minimize is in this case

$$\min_w \frac{1}{2n_{samples}}||Xw - y||_2^2 + \alpha\rho||w||_1 + \frac{\alpha(1-\rho)}{2}||w||_2^2$$



The class `ElasticNetCV` can be used to set the parameters `alpha` (α) and `l1_ratio` (ρ) by cross-validation.

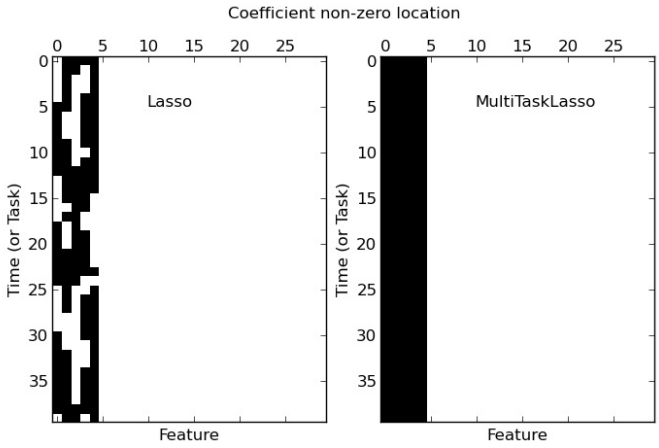
Examples:

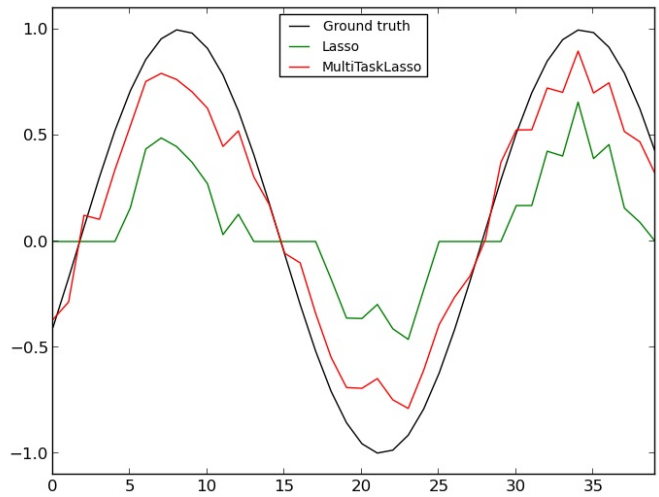
- *Lasso and Elastic Net for Sparse Signals*
- *Lasso and Elastic Net*

1.1.5. Multi-task Lasso

The `MultiTaskLasso` is a linear model that estimates sparse coefficients for multiple regression problems jointly: `y` is a 2D array, of shape (n_samples, n_tasks). The constraint is that the selected features are the same for all the regression problems, also called tasks.

The following figure compares the location of the non-zeros in W obtained with a simple Lasso or a MultiTaskLasso. The Lasso estimates yields scattered non-zeros while the non-zeros of the MultiTaskLasso are full columns.





Fitting a time-series model, imposing that any active feature be active at all times.

Examples:

- *Joint feature selection with multi-task Lasso*

Mathematically, it consists of a linear model trained with a mixed ℓ_1 ℓ_2 prior as regularizer. The objective function to minimize is:

$$\min_w \frac{1}{2n_{samples}}||XW - Y||_2^2 + \alpha||W||_{21}$$

where;

$$||W||_2 1 = \sum_i \sqrt{\sum_j w_{ij}^2}$$

The implementation in the class `MultiTaskLasso` uses coordinate descent as the algorithm to fit the coefficients.

1.1.6. 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 numerically efficient in contexts where p >> n (i.e., when the number of dimensions is significantly greater than the number of points)
- 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.

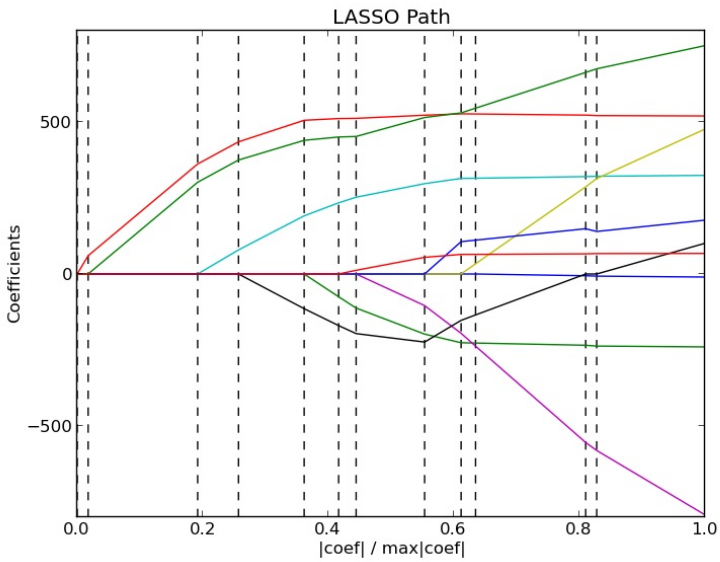
The disadvantages of the LARS method include:

- 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`**.

1.1.7. 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.




```
>>> from sklearn import linear_model
>>> clf = linear_model.LassoLars(alpha=.1)
>>> clf.fit([[0, 0], [1, 1]], [0, 1])
LassoLars(alpha=0.1, copy_X=True, eps=..., fit_intercept=True,
     fit_path=True, max_iter=500, normalize=True, precompute='auto',
     verbose=False)
>>> clf.coef_
array([ 0.717157...,  0.        ])
```

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`

1.1.7.1. 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 coefficients 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.

1.1.8. Orthogonal Matching Pursuit (OMP)

`OrthogonalMatchingPursuit` and `orthogonal_mp` implements the OMP algorithm for approximating the fit of a linear model with constraints imposed on the number of non-zero coefficients (ie. the L $_0$ pseudo-norm).

Being a forward feature selection method like *Least Angle Regression*, orthogonal matching pursuit can approximate the optimum solution vector with a fixed number of non-zero elements:

$$\arg\min ||y - X\gamma||_2^2 \text{ subject to } ||\gamma||_0 \leq n_{nonzero\_coefs}$$

Alternatively, orthogonal matching pursuit can target a specific error instead of a specific number of non-zero coefficients. This can be expressed as:

$$\arg\min ||\gamma||_0 \text{ subject to } ||y - X\gamma||_2^2 \leq \text{tol}$$

OMP is based on a greedy algorithm that includes at each step the atom most highly correlated with the current residual. It is similar to the simpler matching pursuit (MP) method, but better in that at each iteration, the residual is recomputed using an orthogonal projection on the space of the previously chosen dictionary elements.

Examples:

- *Orthogonal Matching Pursuit*

References:

- http://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf
- Matching pursuits with time-frequency dictionaries, S. G. Mallat, Z. Zhang,

1.1.9. Bayesian Regression

Bayesian regression techniques can be used to include regularization parameters in the estimation procedure: the regularization parameter is not set in a hard sense but tuned to the data at hand.

This can be done by introducing uninformative priors over the hyper parameters of the model. The ℓ_2 regularization used in Ridge Regression is equivalent to finding a maximum a-postiori solution under a Gaussian prior over the parameters w with precision $\lambda^{-}1$. Instead of setting *lambda* manually, it is possible to treat it as a random variable to be estimated from the data.

To obtain a fully probabilistic model, the output y is assumed to be Gaussian distributed around Xw:

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

Alpha is again treated as a random variable that is to be estimated from the data.

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.

References

- A good introduction to Bayesian methods is given in C. Bishop: Pattern Recognition and Machine learning
- Original Algorithm is detailed in the book *Bayesian learning for neural networks* by Radford M. Neal

1.1.9.1. Bayesian Ridge Regression

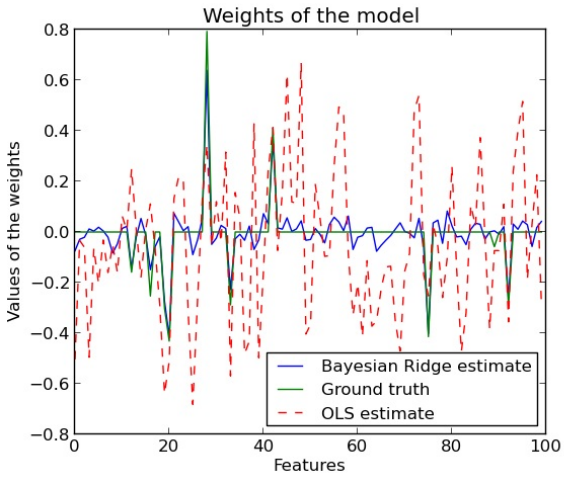
`BayesianRidge` estimates a probabilistic model of the regression problem as described above. The prior for the parameter w is given by a spherical Gaussian:

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

The priors over α and λ are chosen to be gamma distributions, the conjugate prior for the precision of the Gaussian.

The resulting model is called *Bayesian Ridge Regression*, and is similar to the classical `Ridge`. The parameters w, α and λ are estimated jointly during the fit of the model. The remaining hyperparameters are the parameters of the gamma priors over α and λ. These are usually chosen to be *non-informative*. The parameters are estimated by maximizing the *marginal log likelihood*.

By default $\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 1.e^{-6}$.



Bayesian Ridge Regression is used for regression:

```
>>> from sklearn import linear_model
>>> X = [[0., 0.], [1., 1.], [2., 2.], [3., 3.]]
>>> Y = [0., 1., 2., 3.]
>>> clf = linear_model.BayesianRidge()
>>> clf.fit(X, Y)
BayesianRidge(alpha_1=1e-06, alpha_2=1e-06, compute_score=False, copy_X=True,
       fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06, n_iter=300,
       normalize=False, tol=0.001, verbose=False)
```

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

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

The weights w 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*. 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.

1.1.9.2. Automatic Relevance Determination - ARD

`ARDRegression` is very similar to Bayesian Ridge Regression, but can lead to sparser weights w [1] [2]. `ARDRegression` poses a different prior over w, by dropping the assumption of the Gaussian being spherical.

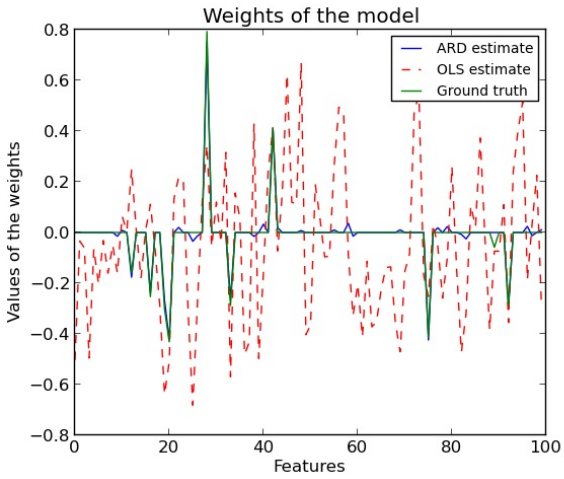
Instead, the distribution over w is assumed to be an axis-parallel, elliptical Gaussian distribution.

This means each weight w_i is drawn from a Gaussian distribution, centered on zero and with a precision λ_i:

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

with $diag\ (A) = \lambda = \{\lambda_1, ..., \lambda_p\}$.

In contrast to Bayesian Ridge Regression, each coordinate of w_i has its own standard deviation λ_i. The prior over all λ_i is chosen to be the same gamma distribution given by hyperparameters λ_1 and λ_2.



Examples:

- *Automatic Relevance Determination Regression (ARD)*

References:

[1] Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 7.2.1

[2] David Wipf and Srikantan Nagarajan: A new view of automatic relevance determination.

1.1.10. Logistic regression

Logistic regression, despite its name, is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

The implementation of logistic regression in scikit-learn can be accessed from class `LogisticRegression`. This implementation can fit a multiclass (one-vs-rest) logistic regression with optional L2 or L1 regularization.

As an optimization problem, binary class L2 penalized logistic regression minimizes the following cost function:

$$\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1).$$

Similarly, L1 regularized logistic regression solves the following optimization problem

$$\min_{w,c} \|w\|_1 + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1).$$

L1 penalization yields sparse predicting weights. For L1 penalization `sklearn.svm.l1_min_c` allows to calculate the lower bound for C in order to get a non “null” (all feature weights to zero) model.

The implementation of Logistic Regression relies on the excellent LIBLINEAR library, which is shipped with scikit-learn.

Examples:

- *L1 Penalty and Sparsity in Logistic Regression*
- *Path with L1- Logistic Regression*

Note: Feature selection with sparse logistic regression

A logistic regression with L1 penalty yields sparse models, and can thus be used to perform feature selection, as detailed in *L1-based feature selection*.

1.1.11. Stochastic Gradient Descent - SGD

Stochastic gradient descent is a simple yet very efficient approach to fit linear models. It is particularly useful when the number of samples (and the number of features) is very large. The `partial_fit` method allows only/out-of-core learning.

The classes **`SGDClassifier`** and **`SGDRegressor`** provide functionality to fit linear models for classification and regression using different (convex) loss functions and different penalties. E.g., with `loss="log"`, **`SGDClassifier`** fits a logistic regression model, while with `loss="hinge"` it fits a linear support vector machine (SVM).

References

- *Stochastic Gradient Descent*

1.1.12. Perceptron

The **`Perceptron`** is another simple algorithm suitable for large scale learning. By default:

- It does not require a learning rate.
- It is not regularized (penalized).
- It updates its model only on mistakes.

The last characteristic implies that the Perceptron is slightly faster to train than SGD with the hinge loss and that the resulting models are sparser.

1.1.13. Passive Aggressive Algorithms

The passive-aggressive algorithms are a family of algorithms for large-scale learning. They are similar to the Perceptron in that they do not require a learning rate. However, contrary to the Perceptron, they include a regularization parameter `C`.

For classification, **`PassiveAggressiveClassifier`** can be used with `loss='hinge'` (PA-I) or

`loss='squared_hinge'` (PA-II). For regression, **`PassiveAggressiveRegressor`** can be used with `loss='epsilon_insensitive'` (PA-I) or `loss='squared_epsilon_insensitive'` (PA-II).

References:

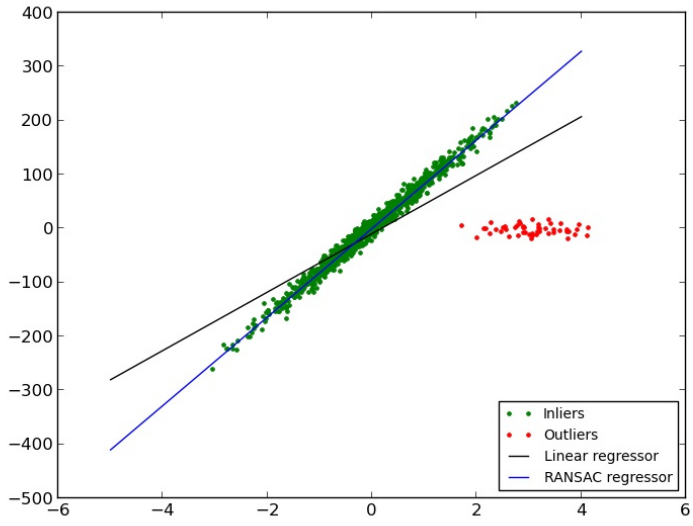
- "Online Passive-Aggressive Algorithms" K. Crammer, O. Dekel, J. Keshat, S. Shalev-Shwartz, Y. Singer - JMLR 7 (2006)

1.1.14. Robustness to outliers: RANSAC

RANSAC (RANdom SAmple Consensus) is an iterative algorithm for the robust estimation of parameters from a subset of inliers from the complete data set.

It is an iterative method to estimate the parameters of a mathematical model. RANSAC is a non-deterministic algorithm producing only a reasonable result with a certain probability, which is dependent on the number of iterations (see *max_trials* parameter). It is typically used for linear and non-linear regression problems and is especially popular in the fields of photogrammetric computer vision.

The algorithm splits the complete input sample data into a set of inliers, which may be subject to noise, and outliers, which are e.g. caused by erroneous measurements or invalid hypotheses about the data. The resulting model is then estimated only from the determined inliers.



Each iteration performs the following steps:

1. Select `min_samples` random samples from the original data and check whether the set of data is valid (see `is_data_valid`).
2. Fit a model to the random subset (`base_estimator.fit`) and check whether the estimated model is valid (see `is_model_valid`).
3. Classify all data as inliers or outliers by calculating the residuals to the estimated model (`base_estimator.predict(X) - y`) - all data samples with absolute residuals smaller than the `residual_threshold` are considered as inliers.
4. Save fitted model as best model if number of inlier samples is maximal. In case the current estimated model has the same number of inliers, it is only considered as the best model if it has better score.

These steps are performed either a maximum number of times (`max_trials`) or until one of the special stop criteria are met (see `stop_n_inliers` and `stop_score`). The final model is estimated using all inlier samples (consensus set) of the previously determined best model.

The `is_data_valid` and `is_model_valid` functions allow to identify and reject degenerate combinations of random sub-samples. If the estimated model is not needed for identifying degenerate cases, `is_data_valid` should be used as it is called prior to fitting the model and thus leading to better computational performance.

Examples:

- *Robust linear model estimation using RANSAC*

References:

- http://en.wikipedia.org/wiki/RANSAC
- "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography" Martin A. Fischler and Robert C. Bolles - SRI International (1981)
- "Performance Evaluation of RANSAC Family" Sunglok Choi, Taemin Kim and Wonpil Yu - BMVC (2009)

1.1.15. Polynomial regression: extending linear models with basis functions

One common pattern within machine learning is to use linear models trained on nonlinear functions of the data. This approach maintains the generally fast performance of linear methods, while allowing them to fit a much wider range of data.

For example, a simple linear regression can be extended by constructing **polynomial features** from the coefficients. In the standard linear regression case, you might have a model that looks like this for two-dimensional data:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2$$

If we want to fit a paraboloid to the data instead of a plane, we can combine the features in second-order polynomials, so that the model looks like this:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$$

The (sometimes surprising) observation is that this is *still a linear model*: to see this, imagine creating a new variable

$$z = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]$$

With this re-labeling of the data, our problem can be written

$$\hat{y}(w, x) = w_0 + w_1 z_1 + w_2 z_2 + w_3 z_3 + w_4 z_4 + w_5 z_5$$

We see that the resulting *polynomial regression* is in the same class of linear models we'd considered above

(i.e. the model is linear in w) and can be solved by the same techniques. By considering linear fits within a higher-dimensional space built with these basis functions, the model has the flexibility to fit a much broader range of data.

Here is an example of applying this idea to one-dimensional data, using polynomial features of varying degrees:

This figure is created using the `PolynomialFeatures` preprocessor. This preprocessor transforms an input data matrix into a new data matrix of a given degree. It can be used as follows:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> import numpy as np
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> poly = PolynomialFeatures(degree=2)
>>> poly.fit_transform(X)
array([[ 1,  0,  1,  0,  0,  1],
       [ 1,  2,  3,  4,  6,  9],
       [ 1,  4,  5, 16, 20, 25]])
```

The features of `X` have been transformed from $[x_1, x_2]$ to $[1, x_1, x_2, x_1^2, x_1 x_2, x_2^2]$, and can now be used within any linear model.

This sort of preprocessing can be streamlined with the *Pipeline* tools. A single object representing a simple polynomial regression can be created and used as follows:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.pipeline import Pipeline
>>> model = Pipeline([('poly', PolynomialFeatures(degree=3)),
...                   ('linear', LinearRegression(fit_intercept=False))])
>>> # fit to an order-3 polynomial data
>>> x = np.arange(5)
>>> y = 3 - 2 * x + x ** 2 - x ** 3
>>> model = model.fit(x[:, np.newaxis], y)
>>> model.named_steps['linear'].coef_
array([ 3., -2.,  1., -1.])
```

The linear model trained on polynomial features is able to exactly recover the input polynomial coefficients.

In some cases it's not necessary to include higher powers of any single feature, but only the so-called *interaction features* that multiply together at most d distinct features. These can be gotten from `PolynomialFeatures` with the setting `interaction_only=True`.

For example, when dealing with boolean features, $x_i^n = x_i$ for all n and is therefore useless; but $x_i x_j$ represents the conjunction of two booleans. This way, we can solve the XOR problem with a linear classifier:

```
>>> from sklearn.linear_model import Perceptron
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
>>> y = X[:, 0] ^ X[:, 1]
>>> X = PolynomialFeatures(interaction_only=True).fit_transform(X)
>>> X
array([[1, 0, 0, 0],
       [1, 0, 1, 0],
       [1, 1, 0, 0],
       [1, 1, 1, 1]])
>>> clf = Perceptron(fit_intercept=False, n_iter=10).fit(X, y)
>>> clf.score(X, y)
```

```
1.0
```

Previous Next

1.2. 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.

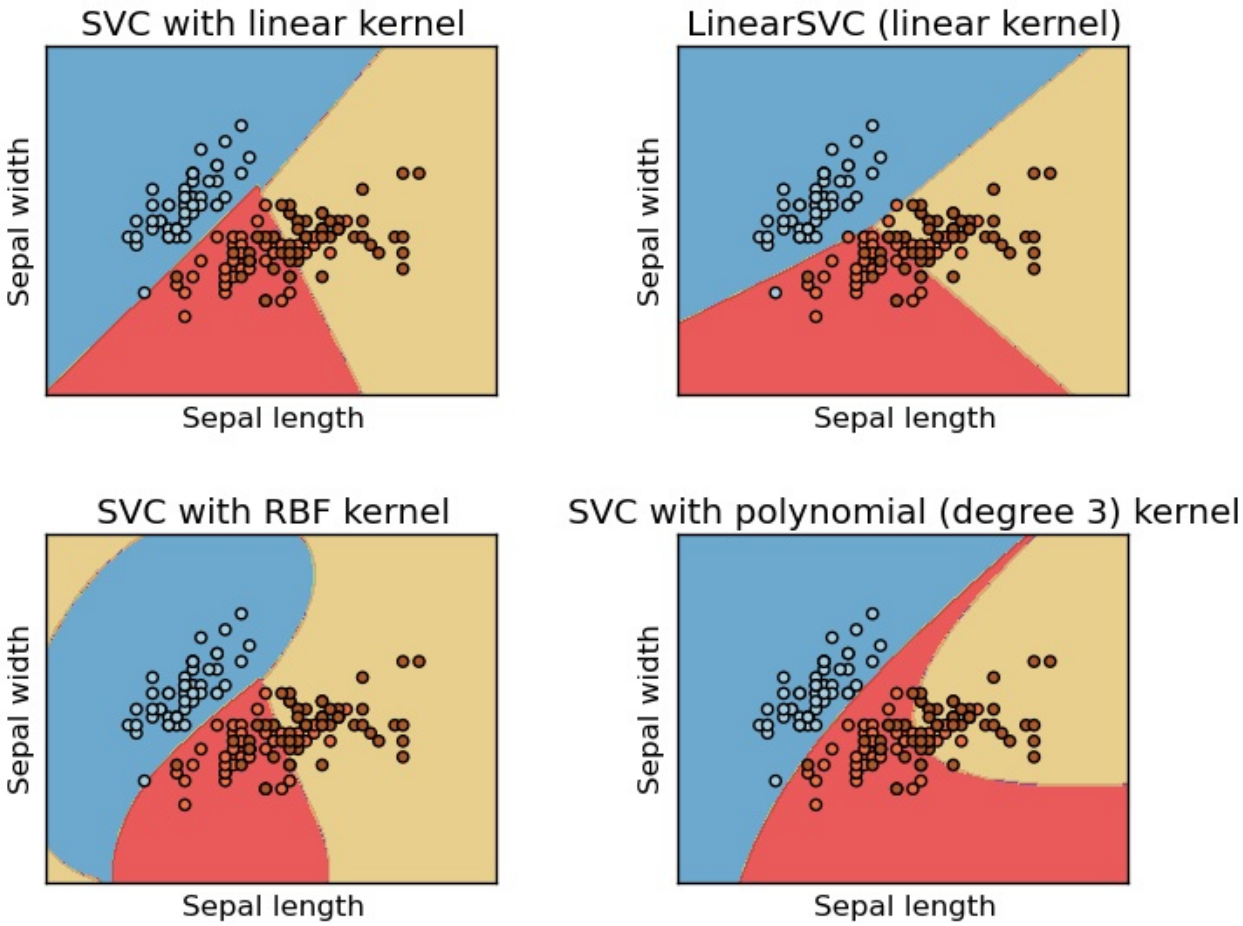
The disadvantages of support vector machines include:

- 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 an expensive five-fold cross-validation (see *Scores and probabilities*, below).

The support vector machines in scikit-learn support both dense (`numpy.ndarray` and convertible to that by `numpy.asarray`) and sparse (any `scipy.sparse`) sample vectors as input. However, to use an SVM to make predictions for sparse data, it must have been fit on such data. For optimal performance, use C-ordered `numpy.ndarray` (dense) or `scipy.sparse.csr_matrix` (sparse) with `dtype=float64`.

1.2.1. 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 class labels (strings or integers), size `[n_samples]`:

```
>>> from sklearn import svm
>>> X = [[0, 0], [1, 1]]
>>> y = [0, 1]
>>> clf = svm.SVC()
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.0, kernel='rbf', max_iter=-1, probability=False, random_state=None,
shrinking=True, tol=0.001, verbose=False)
```

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`:

```
>>> # 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]...)
```

1.2.1.1. 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(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.0, kernel='rbf', max_iter=-1, probability=False, random_state=None,
shrinking=True, tol=0.001, verbose=False)
>>> 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(C=1.0, class_weight=None, dual=True, fit_intercept=True,
intercept_scaling=1, loss='l2', multi_class='ovr', penalty='l2',
random_state=None, tol=0.0001, verbose=0)
>>> dec = lin_clf.decision_function([[1]])
>>> dec.shape[1]
4
```

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

Note that the **LinearSVC** also implements an alternative multi-class strategy, the so-called multi-class SVM formulated by Crammer and Singer, by using the option `multi_class='crammer_singer'`. This method is consistent, which is not true for one-vs-rest classification. In practice, one-vs-rest classification is usually preferred, since the results are mostly similar, but the runtime is significantly less.

For "one-vs-rest" **LinearSVC** the attributes `coef_` and `intercept_` have the shape `[n_class, n_features]` and `[n_class]` respectively. Each row of the coefficients corresponds to one of the `n_class` many "one-vs-rest" classifiers and similar for the intercepts, in the order of the "one" class.

In the case of "one-vs-one" **SVC**, the layout of the attributes is a little more involved. In the case of having a linear kernel, The layout of `coef_` and `intercept_` is similar to the one described for **LinearSVC** described above, except that the shape of `coef_` is `[n_class * (n_class - 1) / 2, n_features]`, corresponding to as many binary classifiers. The order for classes 0 to n is "0 vs 1", "0 vs 2" , ... "0 vs n", "1 vs 2", "1 vs 3", "1 vs n", . . . "n-1 vs n".

The shape of `dual_coef_` is `[n_class-1, n_SV]` with a somewhat hard to grasp layout. The columns correspond to the support vectors involved in any of the `n_class * (n_class - 1) / 2` "one-vs-one" classifiers. Each of the support vectors is used in `n_class - 1` classifiers. The `n_class - 1` entries in each

row correspond to the dual coefficients for these classifiers.

This might be made more clear by an example:

Consider a three class problem with with class 0 having three support vectors v_0^0, v_0^1, v_0^2 and class 1 and 2 having two support vectors v_1^0, v_1^1 and v_1^0, v_1^1 respectively. For each support vector v_i^j, there are two dual coefficients. Let's call the coefficient of support vector v_i^j in the classifier between classes i and k $\alpha_{i,k}^j$. Then `dual_coef_` looks like this:

$\alpha_{0,1}^0$	$\alpha_{0,2}^0$	Coefficients for SVs of class 0
$\alpha_{0,1}^1$	$\alpha_{0,2}^1$	
$\alpha_{0,1}^2$	$\alpha_{0,2}^2$	
$\alpha_{1,0}^0$	$\alpha_{1,2}^0$	Coefficients for SVs of class 1
$\alpha_{1,0}^1$	$\alpha_{1,2}^1$	
$\alpha_{2,0}^0$	$\alpha_{2,1}^0$	Coefficients for SVs of class 2
$\alpha_{2,0}^1$	$\alpha_{2,1}^1$	

1.2.1.2. Scores and probabilities

The **SVC** method `decision_function` gives per-class scores for each sample (or a single score per sample in the binary case). When the constructor option `probability` is set to `True`, class membership probability estimates (from the methods `predict_proba` and `predict_log_proba`) are enabled. In the binary case, the probabilities are calibrated using Platt scaling: logistic regression on the SVM's scores, fit by an additional cross-validation on the training data. In the multiclass case, this is extended as per Wu et al. (2004).

Needless to say, the cross-validation involved in Platt scaling is an expensive operation for large datasets. In addition, the probability estimates may be inconsistent with the scores, in the sense that the "argmax" of the scores may not be the argmax of the probabilities. (E.g., in binary classification, a sample may be labeled by `predict` as belonging to a class that has probability <½ according to `predict_proba`.) Platt's method is also known to have theoretical issues. If confidence scores are required, but these do not have to be probabilities, then it is advisable to set `probability=False` and use `decision_function` instead of `predict_proba`.

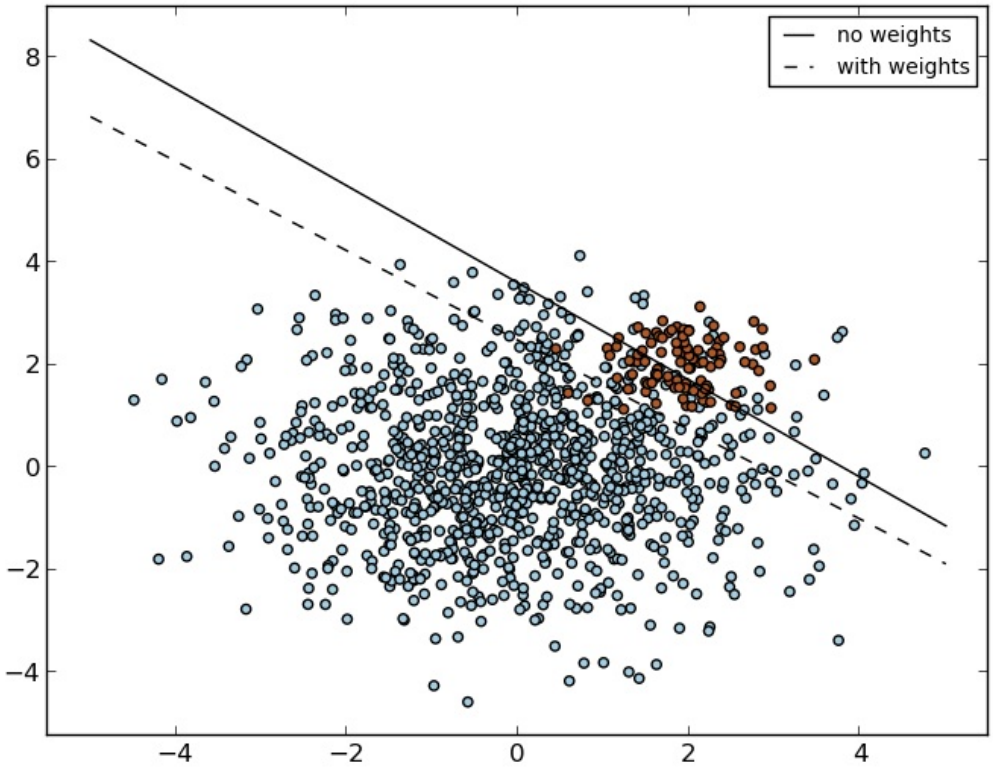
References:

- Wu, Lin and Weng, "Probability estimates for multi-class classification by pairwise coupling". JMLR 5:975-1005, 2004.

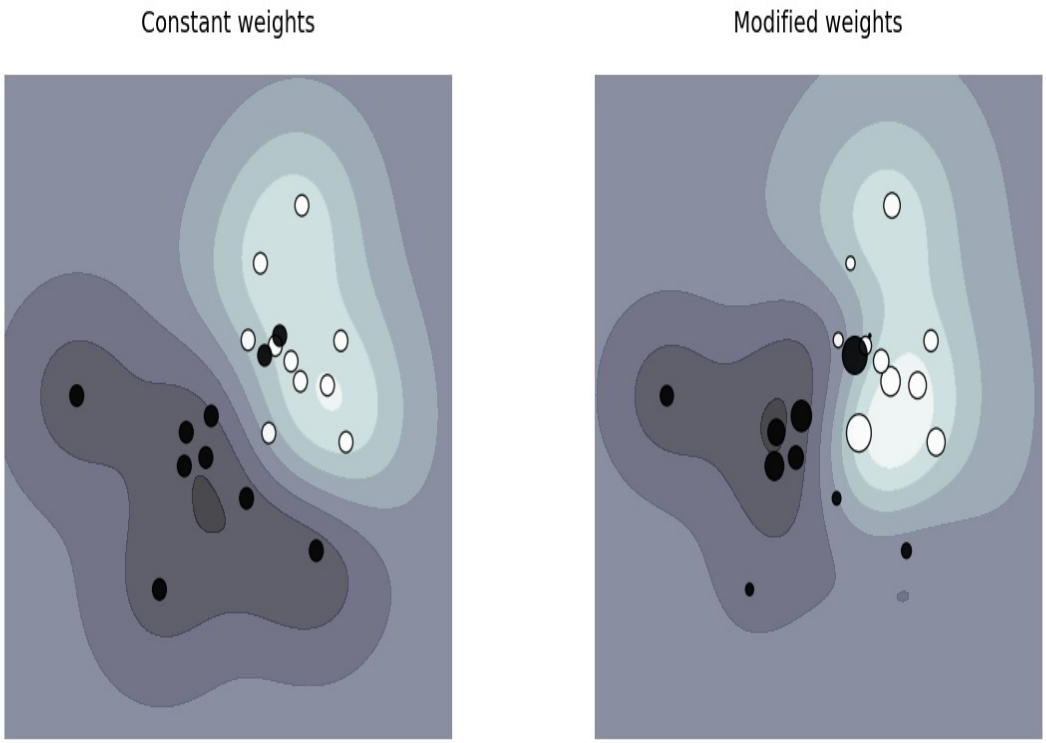
1.2.1.3. Unbalanced problems

In problems where it is desired to give more importance to certain classes or certain individual samples keywords `class_weight` and `sample_weight` can be used.

SVC (but not **NuSVC**) implement a keyword `class_weight` in the `fit` method. It's a dictionary of the form `{class_label : value}`, where value is a floating point number > 0 that sets the parameter `C` of class `class_label` to `C * value`.



SVC, **NuSVC**, **SVR**, **NuSVR** and **OneClassSVM** implement also weights for individual samples in method `fit` through keyword `sample_weight`. Similar to `class_weight`, these set the parameter `C` for the i-th example to `C * sample_weight[i]`.



Examples:

- *Plot different SVM classifiers in the iris dataset*,
- *SVM: Maximum margin separating hyperplane*,
- *SVM: Separating hyperplane for unbalanced classes*
- *SVM-Anova: SVM with univariate feature selection*,
- *Non-linear SVM*
- *SVM: Weighted samples*,

1.2.2. 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 sklearn import svm
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> clf = svm.SVR()
>>> clf.fit(X, y)
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3,
epsilon=0.1, gamma=0.0, kernel='rbf', max_iter=-1, probability=False,
random_state=None, shrinking=True, tol=0.001, verbose=False)
>>> clf.predict([[1, 1]])
array([ 1.5])
```

Examples:

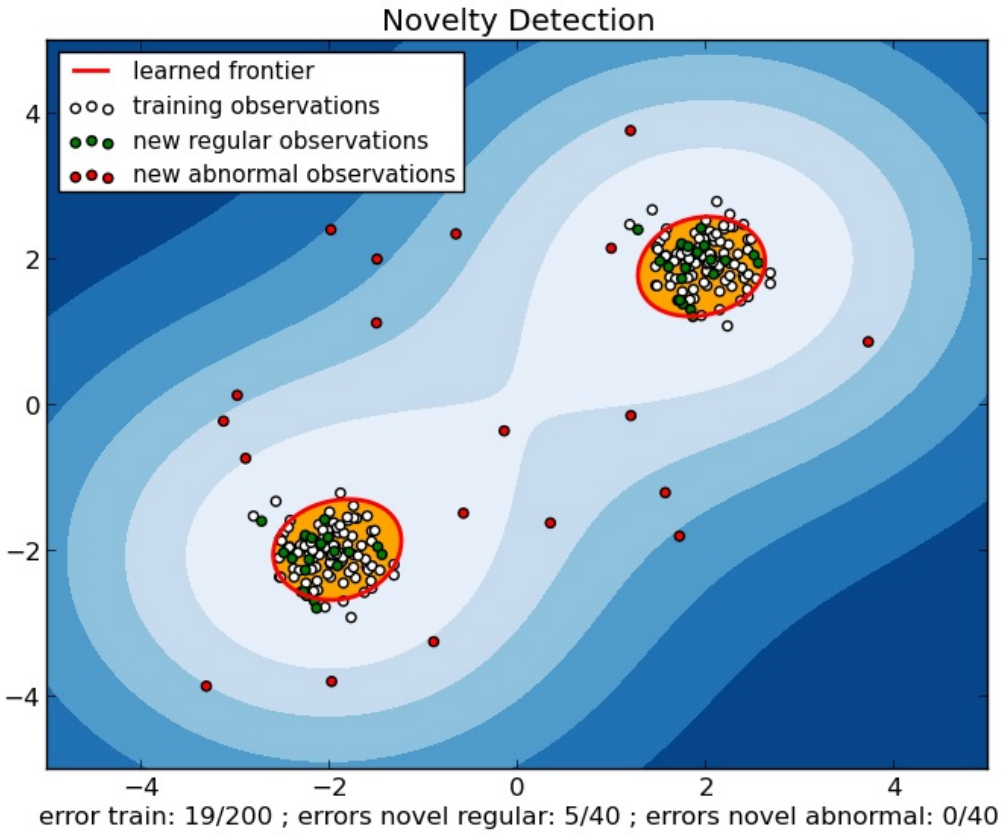
- *Support Vector Regression (SVR) using linear and non-linear kernels*

1.2.3. Density estimation, novelty detection

One-class SVM is used for novelty 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.

See, section *Novelty and Outlier Detection* for more details on this usage.



Examples:

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

1.2.4. 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.

1.2.5. Tips on Practical Use

- **Avoiding data copy**: For **`SVC`**, **`SVR`**, **`NuSVC`** and **`NuSVR`**, if the data passed to certain methods is not C-ordered contiguous, and double precision, it will be copied before calling the underlying C implementation. You can check whether a give numpy array is C-contiguous by inspecting its `flags` attribute.

For **LinearSVC** (and **LogisticRegression**) any input passed as a numpy array will be copied and converted to the liblinear internal sparse data representation (double precision floats and int32 indices of non-zero components). If you want to fit a large-scale linear classifier without copying a dense numpy C-contiguous double precision array as input we suggest to use the **SGDClassifier** class instead. The objective function can be configured to be almost the same as the **LinearSVC** model.

- **Kernel cache size**: For **SVC**, **SVR**, **nuSVC** and **NuSVR**, the size of the kernel cache has a strong impact on run times for larger problems. If you have enough RAM available, it is recommended to set `cache_size` to a higher value than the default of 200(MB), such as 500(MB) or 1000(MB).
- **Setting C**: `C` is `1` by default and it's a reasonable default choice. If you have a lot of noisy observations you should decrease it. It corresponds to regularize more the estimation.
- 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 section *Preprocessing data* for more details on scaling and normalization.
- Parameter `nu` in **NuSVC**/**OneClassSVM**/**NuSVR** approximates the fraction of training errors and support vectors.
- In **SVC**, if data for classification are unbalanced (e.g. many positive and few negative), set `class_weight='auto'` and/or try different penalty parameters `C`.
- 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 tol parameter.
- Using L1 penalization as provided by `LinearSVC(loss='l2', penalty='l1', dual=False)` yields a sparse solution, i.e. only a subset of feature weights is different from zero and contribute to the decision function. Increasing `C` yields a more complex model (more feature are selected). The `C` value that yields a "null" model (all weights equal to zero) can be calculated using **l1_min_c**.

1.2.6. Kernel functions

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

- linear: $\langle x, x' \rangle$.
- polynomial: $(\gamma \langle x, x' \rangle + r)^d$. d is specified by keyword `degree`, r by `coef0`.
- rbf: $\exp(-\gamma |x - x'|^2)$. γ is specified by keyword `gamma`, must be greater than 0.
- sigmoid ($\tanh(\gamma \langle x, x' \rangle + r)$), where r is specified by `coef0`.

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'
```

1.2.6.1. 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.

1.2.6.1.1. 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 sklearn import svm
>>> def my_kernel(x, y):
...     return np.dot(x, y.T)
...
>>> clf = svm.SVC(kernel=my_kernel)
```

Examples:

- *SVM with custom kernel*.

1.2.6.1.2. Using the Gram matrix

Set `kernel='precomputed'` and pass the Gram matrix instead of X in the fit method. At the moment, the kernel values between *all* training vectors and the test vectors must be provided.

```
>>> import numpy as np
>>> from sklearn import svm
>>> X = np.array([[0, 0], [1, 1]])
>>> y = [0, 1]
>>> clf = svm.SVC(kernel='precomputed')
>>> # linear kernel computation
>>> gram = np.dot(X, X.T)
>>> clf.fit(gram, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
gamma=0.0, kernel='precomputed', max_iter=-1, probability=False,
random_state=None, shrinking=True, tol=0.001, verbose=False)
>>> # predict on training examples
>>> clf.predict(gram)
array([0, 1])
```

1.2.6.1.3. Parameters of the RBF Kernel

When training an SVM with the *Radial Basis Function* (RBF) kernel, two parameters must be considered: `C` and `gamma`. The parameter `C`, common to all SVM kernels, trades off misclassification of training examples against simplicity of the decision surface. A low `C` makes the decision surface smooth, while a high `C` aims at classifying all training examples correctly. `gamma` defines how much influence a single training example has. The larger `gamma` is, the closer other examples must be to be affected.

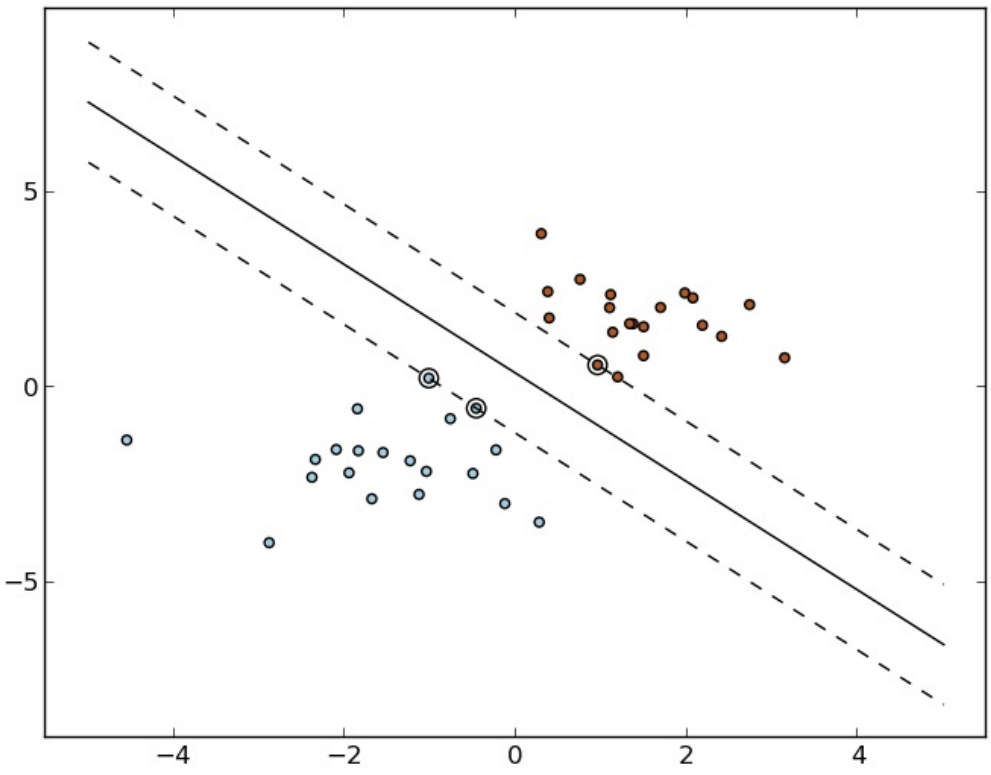
Proper choice of `C` and `gamma` is critical to the SVM's performance. One is advised to use **`GridSearchCV`** with `C` and `gamma` spaced exponentially far apart to choose good values.

Examples:

- *RBF SVM parameters*

1.2.7. 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.



1.2.7.1. SVC

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

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1,n} \zeta_i$$
$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$
$$\zeta_i \geq 0, i = 1, ..., n$$

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 n by n positive semidefinite matrix, $Q_{ij} \equiv K(x_i, x_j)$ 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:

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

Note: While SVM models derived from libsvm and liblinear use `C` as regularization parameter, most other estimators use `alpha`. The relation between both is $C = \frac{n\_samples}{alpha}$.

This parameters can be accessed through the members `dual_coef_` `which holds the product :math:`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)

1.2.7.2. 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.

It can be shown that the ν-SVC formulation is a reparametrization of the C-SVC and therefore mathematically equivalent.

1.2.8. 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. 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 10^5 features.

The advantages of Stochastic Gradient Descent are:

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

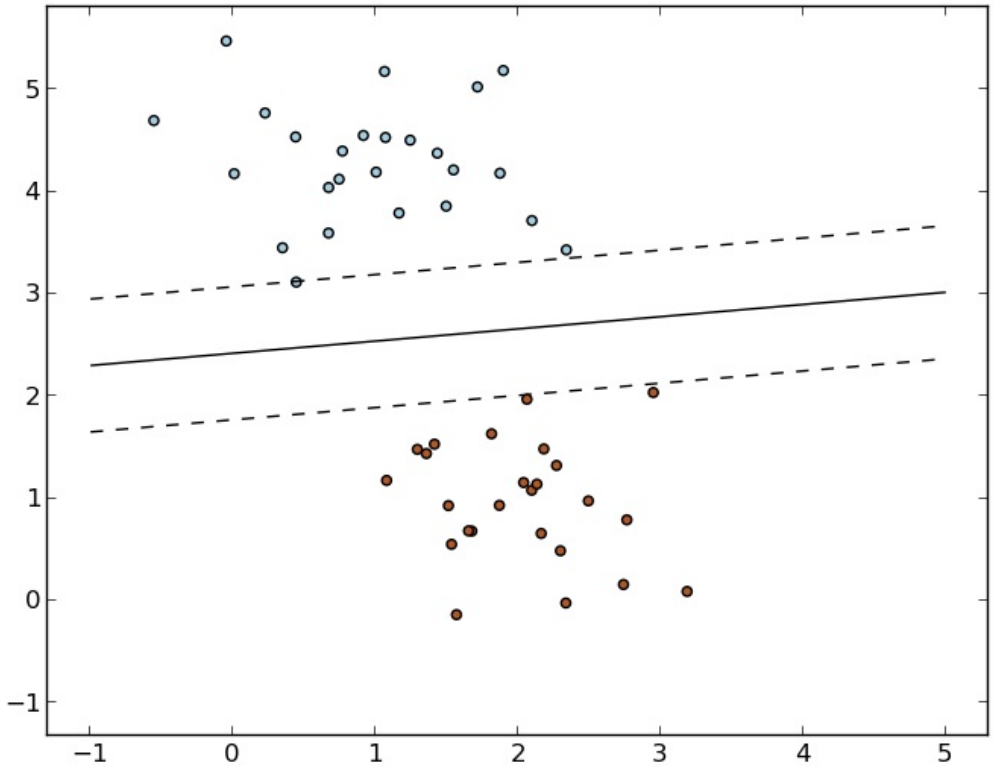
The disadvantages of Stochastic Gradient Descent include:

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

1.3.1. 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 sklearn.linear_model import SGDClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = SGDClassifier(loss="hinge", penalty="l2")
>>> clf.fit(X, y)
SGDClassifier(alpha=0.0001, class_weight=None, epsilon=0.1, eta0=0.0,
       fit_intercept=True, l1_ratio=0.15, learning_rate='optimal',
       loss='hinge', n_iter=5, n_jobs=1, penalty='l2', power_t=0.5,
       random_state=None, shuffle=False, verbose=0, warm_start=False)
```

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([[ 9.91080278,  9.91080278]])
```

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

```
>>> clf.intercept_
array([-9.990...])
```

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 **`SGDClassifier.decision_function`**:

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

```
array([ 29.65318117])
```

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,
- and all regression losses below.

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 and may result in sparser models, even when L2 penalty is used.

Using `loss="log"` or `loss="modified_huber"` enables the `predict_proba` method, which gives a vector of probability estimates $P(y|x)$ per sample x:

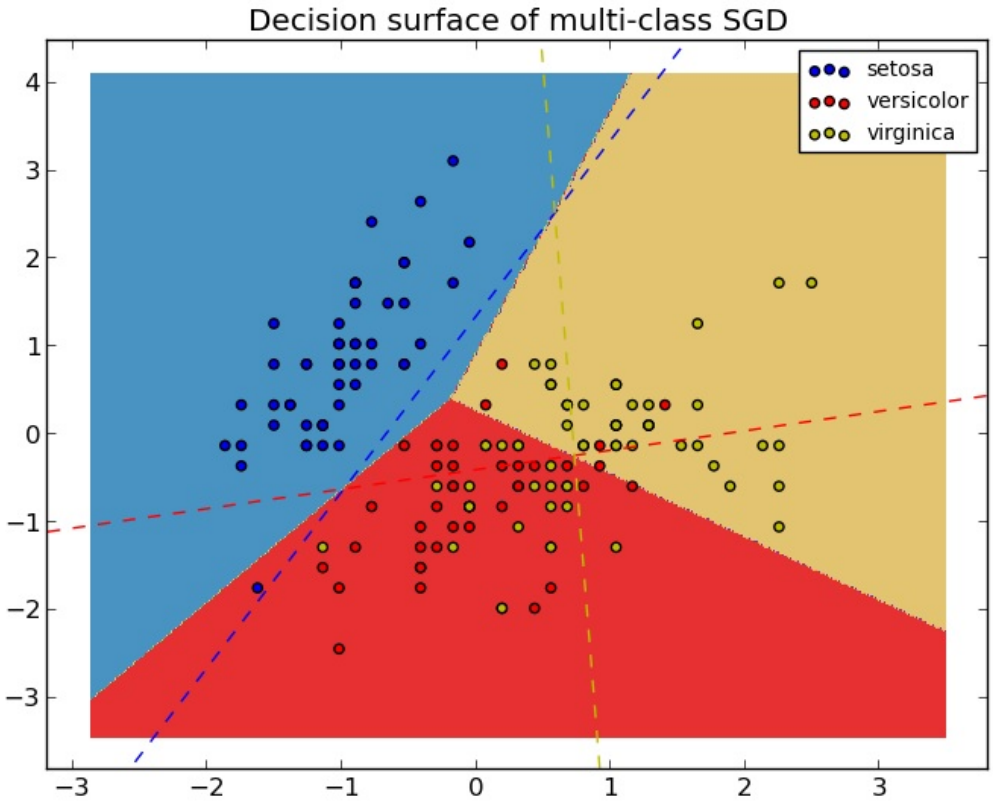
```
>>> clf = SGDClassifier(loss="log").fit(X, y)
>>> clf.predict_proba([[1., 1.]])
array([[ 0.0000005,  0.9999995]])
```

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; `(1 - l1_ratio) * L2 + l1_ratio * 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 `l1_ratio` controls the convex combination of L1 and L2 penalty.

`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-dimensionally 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 attribute `classes_`). Note that, in principle, since they allow to create a probability model, `loss="log"` and `loss="modified_huber"` are more suitable for one-vs-all classification.

`SGDClassifier` supports both weighted classes and weighted instances via the fit parameters `class_weight` and `sample_weight`. See the examples below and the doc string of **`SGDClassifier.fit`** for further information.

Examples:

- *SGD: Maximum margin separating hyperplane*,
- *Plot multi-class SGD on the iris dataset*
- *SGD: Weighted samples*
- *SVM: Separating hyperplane for unbalanced classes* (See the *Note*)

1.3.2. Regression

The class **`SGDRegressor`** implements a plain stochastic gradient descent learning routine which supports different loss functions and penalties to fit linear regression models. **`SGDRegressor`** is well suited for regression problems with a large number of training samples (> 10.000), for other problems we recommend **`Ridge`**, **`Lasso`**, or **`ElasticNet`**.

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,
- `loss="epsilon_insensitive"`: linear Support Vector Regression.

The Huber and epsilon-insensitive loss functions can be used for robust regression. The width of the insensitive region has to be specified via the parameter `epsilon`. This parameter depends on the scale of the target variables.

1.3.3. 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 built-in support for sparse data given in any matrix in a format supported by scipy.sparse. For maximum efficiency, however, use the CSR matrix format as defined in scipy.sparse.csr_matrix.

Examples:

- *Classification of text documents using sparse features*

1.3.4. 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 $O(knp)$, where k is the number of iterations (epochs) and 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.

1.3.5. 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. This can be easily done using **`StandardScaler`**:

  ```
  from sklearn.preprocessing import StandardScaler
  scaler = StandardScaler()
  scaler.fit(X_train)  # Don't cheat - fit only on training data
  X_train = scaler.transform(X_train)
  X_test = scaler.transform(X_test)  # apply same transformation to test data
  ```

  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 **`GridSearchCV`**, usually in

the range `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.
- If you apply SGD to features extracted using PCA we found that it is often wise to scale the feature values by some constant *c* such that the average L2 norm of the training data equals one.

References:

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

1.3.6. 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

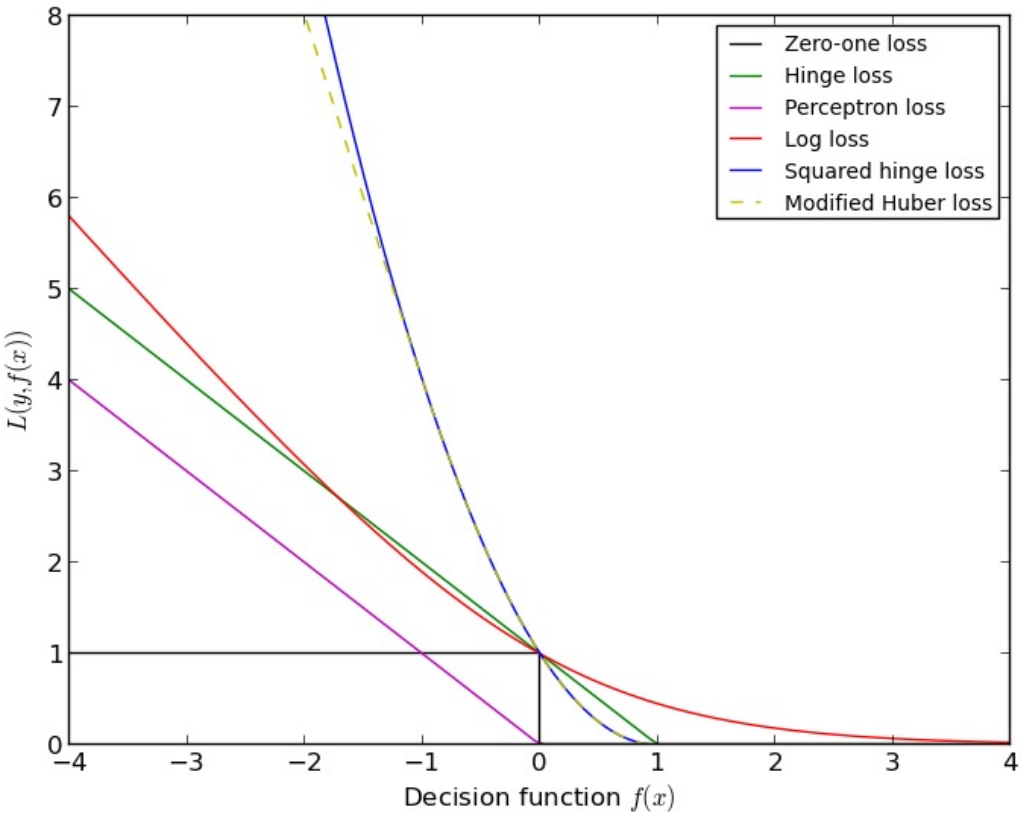
$$E(w, b) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i)) + \alpha R(w)$$

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.
- Epsilon-Insensitive: (soft-margin) Support Vector Regression.

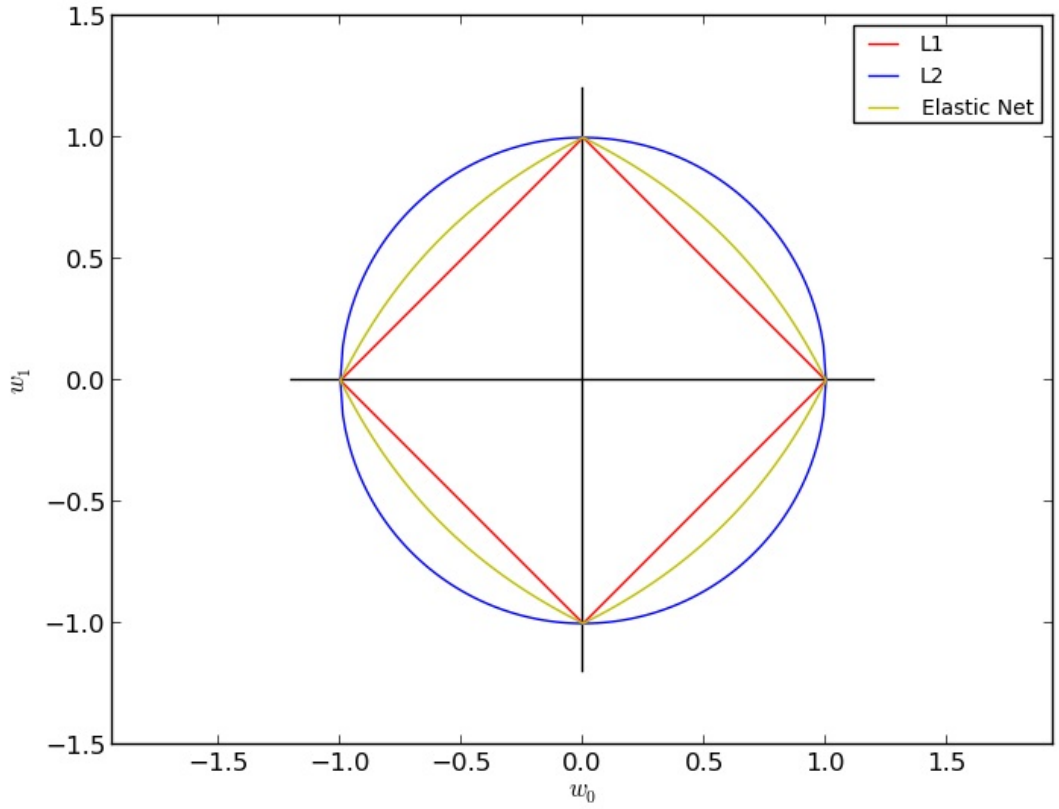
All of the above loss functions can be regarded as an upper bound on the misclassification error (Zero-one loss) as shown in the Figure below.



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: $R(w) := \frac{\rho}{2}\sum_{i=1}^{n} w_i^2 + (1-\rho)\sum_{i=1}^{n} |w_i|$, a convex combination of L2 and L1, where ρ is given by `1 - l1_ratio`.

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



1.3.6.1. 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 learning rate η can be either constant or gradually decaying. For classification, the default learning rate schedule (`learning_rate='optimal'`) is given by

$$\eta^{(t)} = \frac{1}{\alpha(t_0 + t)}$$

where t is the time step (there are a total of *n_samples* * *n_iter* time steps), t_0 is determined based on a heuristic proposed by Léon Bottou such that the expected initial updates are comparable with the expected size of the weights (this assuming that the norm of the training samples is approx. 1). The exact definition can be found in `_init_t` in **`BaseSGD`**.

For regression the default learning rate schedule is inverse scaling (`learning_rate='invscaling'`), given

by

$$\eta^{(t)} = \frac{eta_0}{t^{power\_t}}$$

where eta_0 and $power\_t$ are hyperparameters chosen by the user via `eta0` and `power_t`, resp.

For a constant learning rate use `learning_rate='constant'` and use `eta0` to specify the learning rate.

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.

1.3.7. 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.
- "The Tradeoffs of Large Scale Machine Learning" L. Bottou - Website, 2011.
- "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.

1.4. Nearest Neighbors

`sklearn.neighbors` provides functionality for unsupervised and supervised neighbors-based learning methods. Unsupervised nearest neighbors is the foundation of many other learning methods, notably manifold learning and spectral clustering. Supervised neighbors-based learning comes in two flavors: classification for data with discrete labels, and regression for data with continuous labels.

The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. Neighbors-based methods are known as *non-generalizing* machine learning methods, since they simply "remember" all of its training data (possibly transformed into a fast indexing structure such as a *Ball Tree* or *KD Tree*.).

Despite its simplicity, nearest neighbors has been successful in a large number of classification and regression problems, including handwritten digits or satellite image scenes. Being a non-parametric method, it is often successful in classification situations where the decision boundary is very irregular.

The classes in `sklearn.neighbors` can handle either Numpy arrays or *scipy.sparse* matrices as input. For dense matrices, a large number of possible distance metrics are supported. For sparse matrices, arbitrary Minkowski metrics are supported for searches.

There are many learning routines which rely on nearest neighbors at their core. One example is *kernel density estimation*, discussed in the *density estimation* section.

1.4.1. Unsupervised Nearest Neighbors

`NearestNeighbors` implements unsupervised nearest neighbors learning. It acts as a uniform interface to three different nearest neighbors algorithms: `BallTree`, `KDTree`, and a brute-force algorithm based on routines in `sklearn.metrics.pairwise`. The choice of neighbors search algorithm is controlled through the keyword `'algorithm'`, which must be one of `['auto', 'ball_tree', 'kd_tree', 'brute']`. When the default value `'auto'` is passed, the algorithm attempts to determine the best approach from the training data. For a discussion of the strengths and weaknesses of each option, see Nearest Neighbor Algorithms.

> **Warning:** Regarding the Nearest Neighbors algorithms, if two neighbors, neighbor $k+1$ and k, have identical distances but different labels, the results will depend on the ordering of the training data.

1.4.1.1. Finding the Nearest Neighbors

For the simple task of finding the nearest neighbors between two sets of data, the unsupervised algorithms within `sklearn.neighbors` can be used:

```
>>> from sklearn.neighbors import NearestNeighbors
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> nbrs = NearestNeighbors(n_neighbors=2, algorithm='ball_tree').fit(X)
>>> distances, indices = nbrs.kneighbors(X)
>>> indices
array([[0, 1],
       [1, 0],
       [2, 1],
       [3, 4],
       [4, 3],
       [5, 4]]...)
>>> distances
array([[ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.41421356],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.41421356]])
```

Because the query set matches the training set, the nearest neighbor of each point is the point itself, at a distance of zero.

It is also possible to efficiently produce a sparse graph showing the connections between neighboring points:

```
>>> nbrs.kneighbors_graph(X).toarray()
array([[ 1.,  1.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  1.,  0.],
       [ 0.,  0.,  0.,  1.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  1.]])
```

Our dataset is structured such that points nearby in index order are nearby in parameter space, leading to an approximately block-diagonal matrix of K-nearest neighbors. Such a sparse graph is useful in a variety of circumstances which make use of spatial relationships between points for unsupervised learning: in particular, see **`sklearn.manifold.Isomap`**, **`sklearn.manifold.LocallyLinearEmbedding`**, and **`sklearn.cluster.SpectralClustering`**.

1.4.1.2. KDTree and BallTree Classes

Alternatively, one can use the **`KDTree`** or **`BallTree`** classes directly to find nearest neighbors. This is the functionality wrapped by the **`NearestNeighbors`** class used above. The Ball Tree and KD Tree have the same interface; we'll show an example of using the KD Tree here:

```
>>> from sklearn.neighbors import KDTree
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> kdt = KDTree(X, leaf_size=30, metric='euclidean')
>>> kdt.query(X, k=2, return_distance=False)
array([[0, 1],
       [1, 0],
       [2, 1],
       [3, 4],
       [4, 3],
       [5, 4]]...)
```

Refer to the **`KDTree`** and **`BallTree`** class documentation for more information on the options available for neighbors searches, including specification of query strategies, of various distance metrics, etc. For a list of available metrics, see the documentation of the **`DistanceMetric`** class.

1.4.2. Nearest Neighbors Classification

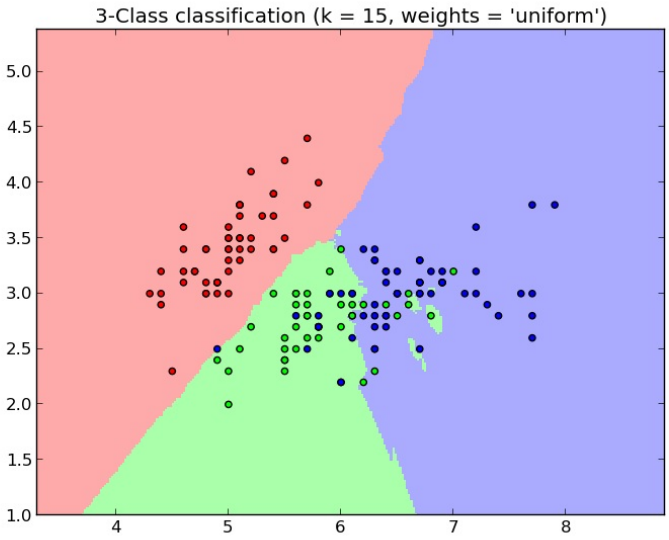
Neighbors-based classification is a type of *instance-based learning* or *non-generalizing learning*: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.

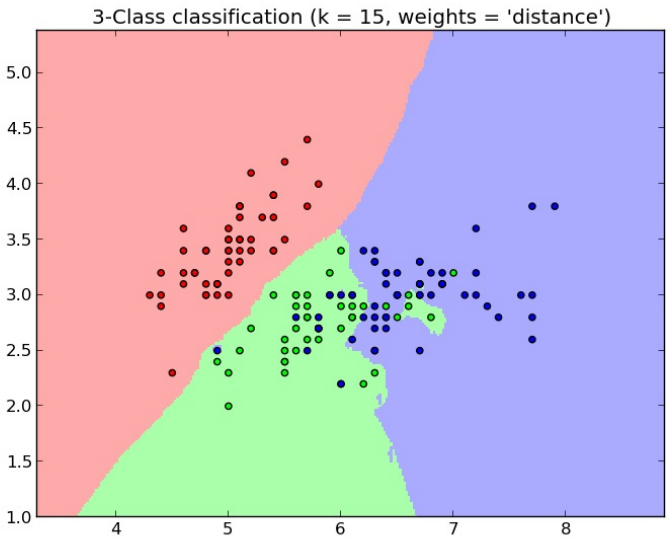
scikit-learn implements two different nearest neighbors classifiers: `KNeighborsClassifier` implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user. `RadiusNeighborsClassifier` implements learning based on the number of neighbors within a fixed radius r of each training point, where r is a floating-point value specified by the user.

The k-neighbors classification in `KNeighborsClassifier` is the more commonly used of the two techniques. The optimal choice of the value k is highly data-dependent: in general a larger k suppresses the effects of noise, but makes the classification boundaries less distinct.

In cases where the data is not uniformly sampled, radius-based neighbors classification in `RadiusNeighborsClassifier` can be a better choice. The user specifies a fixed radius r, such that points in sparser neighborhoods use fewer nearest neighbors for the classification. For high-dimensional parameter spaces, this method becomes less effective due to the so-called “curse of dimensionality”.

The basic nearest neighbors classification uses uniform weights: that is, the value assigned to a query point is computed from a simple majority vote of the nearest neighbors. Under some circumstances, it is better to weight the neighbors such that nearer neighbors contribute more to the fit. This can be accomplished through the `weights` keyword. The default value, `weights = 'uniform'`, assigns uniform weights to each neighbor. `weights = 'distance'` assigns weights proportional to the inverse of the distance from the query point. Alternatively, a user-defined function of the distance can be supplied which is used to compute the weights.





Examples:

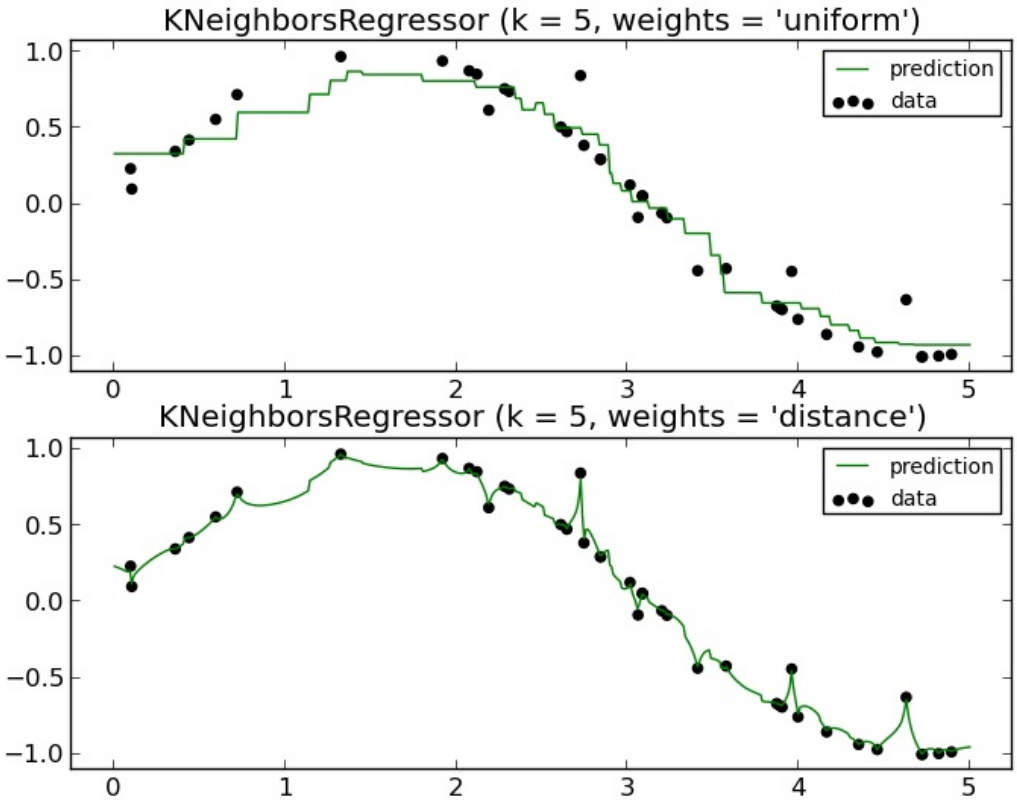
- *Nearest Neighbors Classification*: an example of classification using nearest neighbors.

1.4.3. Nearest Neighbors Regression

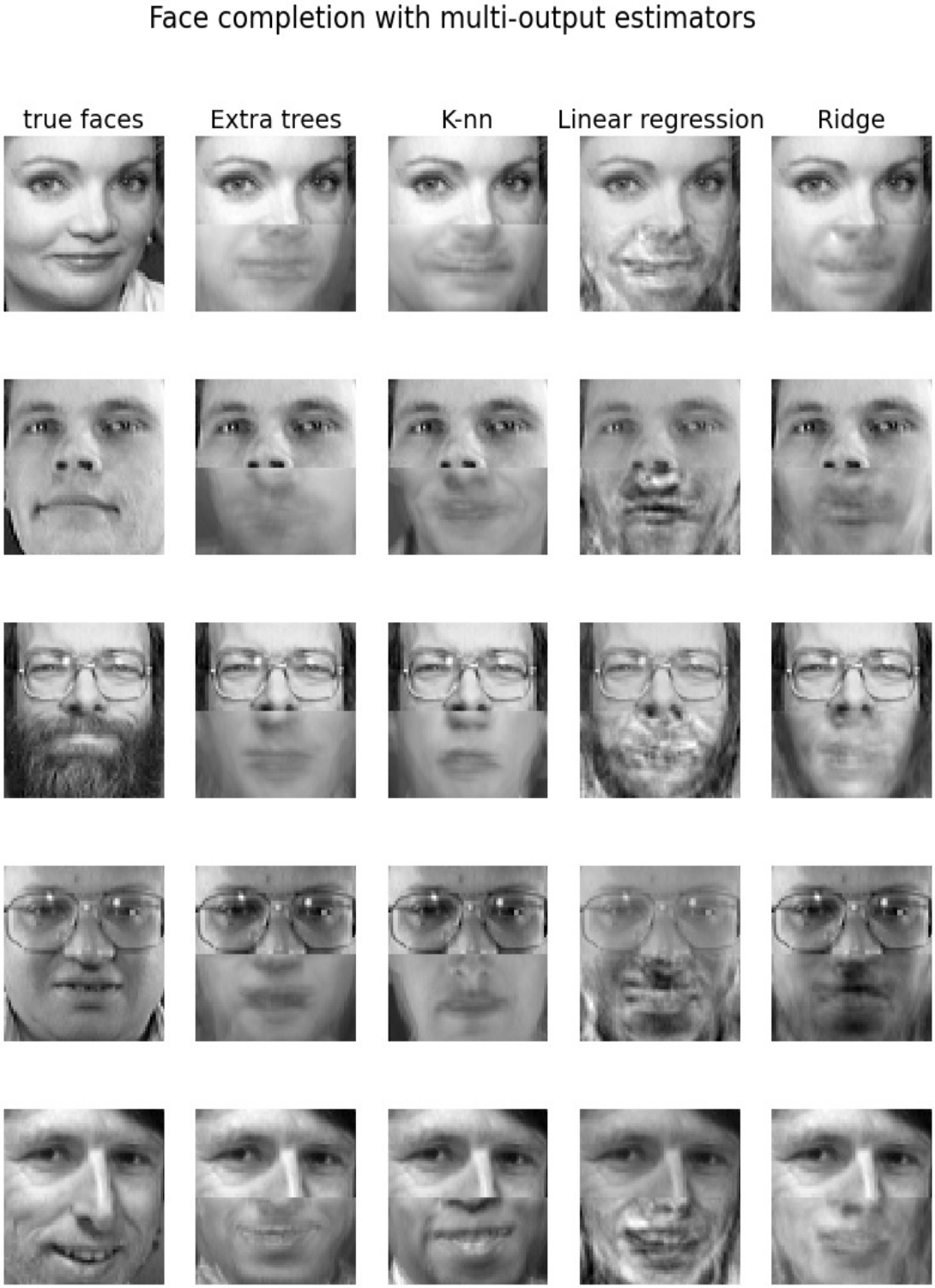
Neighbors-based regression can be used in cases where the data labels are continuous rather than discrete variables. The label assigned to a query point is computed based the mean of the labels of its nearest neighbors.

scikit-learn implements two different neighbors regressors: `KNeighborsRegressor` implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user. `RadiusNeighborsRegressor` implements learning based on the neighbors within a fixed radius r of the query point, where r is a floating-point value specified by the user.

The basic nearest neighbors regression uses uniform weights: that is, each point in the local neighborhood contributes uniformly to the classification of a query point. Under some circumstances, it can be advantageous to weight points such that nearby points contribute more to the regression than faraway points. This can be accomplished through the `weights` keyword. The default value, `weights = 'uniform'`, assigns equal weights to all points. `weights = 'distance'` assigns weights proportional to the inverse of the distance from the query point. Alternatively, a user-defined function of the distance can be supplied, which will be used to compute the weights.



The use of multi-output nearest neighbors for regression is demonstrated in *Face completion with a multi-output estimators*. In this example, the inputs X are the pixels of the upper half of faces and the outputs Y are the pixels of the lower half of those faces.



Examples:

- *Nearest Neighbors regression*: an example of regression using nearest neighbors.
- *Face completion with a multi-output estimators*: an example of multi-output regression using nearest neighbors.

1.4.4. Nearest Neighbor Algorithms

1.4.4.1. Brute Force

Fast computation of nearest neighbors is an active area of research in machine learning. The most naive neighbor search implementation involves the brute-force computation of distances between all pairs of points in the dataset: for N samples in D dimensions, this approach scales as $O[DN^2]$. Efficient brute-force neighbors searches can be very competitive for small data samples. However, as the number of samples N grows, the brute-force approach quickly becomes infeasible. In the classes within **`sklearn.neighbors`**, brute-force neighbors searches are specified using the keyword `algorithm = 'brute'`, and are computed using the routines available in **`sklearn.metrics.pairwise`**.

1.4.4.2. K-D Tree

To address the computational inefficiencies of the brute-force approach, a variety of tree-based data structures have been invented. In general, these structures attempt to reduce the required number of distance calculations by efficiently encoding aggregate distance information for the sample. The basic idea is that if point A is very distant from point B, and point B is very close to point C, then we know that points A and C are very distant, *without having to explicitly calculate their distance*. In this way, the computational cost of a nearest neighbors search can be reduced to $O[DN\log(N)]$ or better. This is a significant improvement over brute-force for large N.

An early approach to taking advantage of this aggregate information was the *KD tree* data structure (short for *K-dimensional tree*), which generalizes two-dimensional *Quad-trees* and 3-dimensional *Oct-trees* to an arbitrary number of dimensions. The KD tree is a binary tree structure which recursively partitions the parameter space along the data axes, dividing it into nested orthotopic regions into which data points are filed. The construction of a KD tree is very fast: because partitioning is performed only along the data axes, no D-dimensional distances need to be computed. Once constructed, the nearest neighbor of a query point can be determined with only $O[\log(N)]$ distance computations. Though the KD tree approach is very fast for low-dimensional ($D < 20$) neighbors searches, it becomes inefficient as D grows very large: this is one manifestation of the so-called "curse of dimensionality". In scikit-learn, KD tree neighbors searches are specified using the keyword `algorithm = 'kd_tree'`, and are computed using the class **`KDTree`**.

References:

- "Multidimensional binary search trees used for associative searching", Bentley, J.L., Communications of the ACM (1975)

1.4.4.3. Ball Tree

To address the inefficiencies of KD Trees in higher dimensions, the *ball tree* data structure was developed. Where KD trees partition data along Cartesian axes, ball trees partition data in a series of nesting hyper-spheres. This makes tree construction more costly than that of the KD tree, but results in a data structure which can be very efficient on highly-structured data, even in very high dimensions.

A ball tree recursively divides the data into nodes defined by a centroid C and radius r, such that each point in the node lies within the hyper-sphere defined by r and C. The number of candidate points for a neighbor search is reduced through use of the *triangle inequality*:

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

With this setup, a single distance calculation between a test point and the centroid is sufficient to determine a lower and upper bound on the distance to all points within the node. Because of the spherical geometry of the ball tree nodes, it can out-perform a *KD-tree* in high dimensions, though the actual performance is highly dependent on the structure of the training data. In scikit-learn, ball-tree-based neighbors searches are specified using the keyword `algorithm = 'ball_tree'`, and are computed using the class **`sklearn.neighbors.BallTree`**. Alternatively, the user can work with the **`BallTree`** class directly.

References:

- "Five balltree construction algorithms", Omohundro, S.M., International Computer Science Institute Technical Report (1989)

1.4.4.4. Choice of Nearest Neighbors Algorithm

The optimal algorithm for a given dataset is a complicated choice, and depends on a number of factors:

- number of samples N (i.e. `n_samples`) and dimensionality D (i.e. `n_features`).
  - *Brute force* query time grows as $O[DN]$
  - *Ball tree* query time grows as approximately $O[D \log(N)]$
  - *KD tree* query time changes with D in a way that is difficult to precisely characterise. For small D (less than 20 or so) the cost is approximately $O[D \log(N)]$, and the KD tree query can be very efficient. For larger D, the cost increases to nearly $O[DN]$, and the overhead due to the tree structure can lead to queries which are slower than brute force.

  For small data sets (N less than 30 or so), $\log(N)$ is comparable to N, and brute force algorithms can be more efficient than a tree-based approach. Both **`KDTree`** and **`BallTree`** address this through providing a *leaf size* parameter: this controls the number of samples at which a query switches to brute-force. This allows both algorithms to approach the efficiency of a brute-force computation for small N.
- data structure: *intrinsic dimensionality* of the data and/or *sparsity* of the data. Intrinsic dimensionality refers to the dimension $d \leq D$ of a manifold on which the data lies, which can be linearly or non-linearly embedded in the parameter space. Sparsity refers to the degree to which the data fills the parameter space (this is to be distinguished from the concept as used in "sparse" matrices. The data matrix may have no zero entries, but the **structure** can still be "sparse" in this sense).
  - *Brute force* query time is unchanged by data structure.
  - *Ball tree* and *KD tree* query times can be greatly influenced by data structure. In general, sparser data with a smaller intrinsic dimensionality leads to faster query times. Because the KD tree internal representation is aligned with the parameter axes, it will not generally show as much improvement as ball tree for arbitrarily structured data.

  Datasets used in machine learning tend to be very structured, and are very well-suited for tree-based queries.
- number of neighbors k requested for a query point.
  - *Brute force* query time is largely unaffected by the value of k
  - *Ball tree* and *KD tree* query time will become slower as k increases. This is due to two effects: first, a larger k leads to the necessity to search a larger portion of the parameter space. Second, using $k > 1$ requires internal queueing of results as the tree is traversed.

As k becomes large compared to N, the ability to prune branches in a tree-based query is reduced. In this situation, Brute force queries can be more efficient.

- number of query points. Both the ball tree and the KD Tree require a construction phase. The cost of this construction becomes negligible when amortized over many queries. If only a small number of queries will be performed, however, the construction can make up a significant fraction of the total cost. If very few query points will be required, brute force is better than a tree-based method.

Currently, `algorithm = 'auto'` selects `'kd_tree'` if $k < N/2$ and the `'effective_metric_'` is in the `'VALID_METRICS'` list of `'kd_tree'`. It selects `'ball_tree'` if $k < N/2$ and the `'effective_metric_'` is not in the `'VALID_METRICS'` list of `'kd_tree'`. It selects `'brute'` if $k >= N/2$. This choice is based on the assumption that the number of query points is at least the same order as the number of training points, and that `leaf_size` is close to its default value of `30`.

1.4.4.5. Effect of `leaf_size`

As noted above, for small sample sizes a brute force search can be more efficient than a tree-based query. This fact is accounted for in the ball tree and KD tree by internally switching to brute force searches within leaf nodes. The level of this switch can be specified with the parameter `leaf_size`. This parameter choice has many effects:

construction time
A larger `leaf_size` leads to a faster tree construction time, because fewer nodes need to be created

query time
Both a large or small `leaf_size` can lead to suboptimal query cost. For `leaf_size` approaching 1, the overhead involved in traversing nodes can significantly slow query times. For `leaf_size` approaching the size of the training set, queries become essentially brute force. A good compromise between these is `leaf_size = 30`, the default value of the parameter.

memory
As `leaf_size` increases, the memory required to store a tree structure decreases. This is especially important in the case of ball tree, which stores a D-dimensional centroid for each node. The required storage space for **`BallTree`** is approximately `1 / leaf_size` times the size of the training set.

`leaf_size` is not referenced for brute force queries.

1.4.5. Nearest Centroid Classifier

The **`NearestCentroid`** classifier is a simple algorithm that represents each class by the centroid of its members. In effect, this makes it similar to the label updating phase of the **`sklearn.KMeans`** algorithm. It also has no parameters to choose, making it a good baseline classifier. It does, however, suffer on non-convex classes, as well as when classes have drastically different variances, as equal variance in all dimensions is assumed. See Linear Discriminant Analysis (**`sklearn.lda.LDA`**) and Quadratic Discriminant Analysis (**`sklearn.qda.QDA`**) for more complex methods that do not make this assumption. Usage of the default **`NearestCentroid`** is simple:

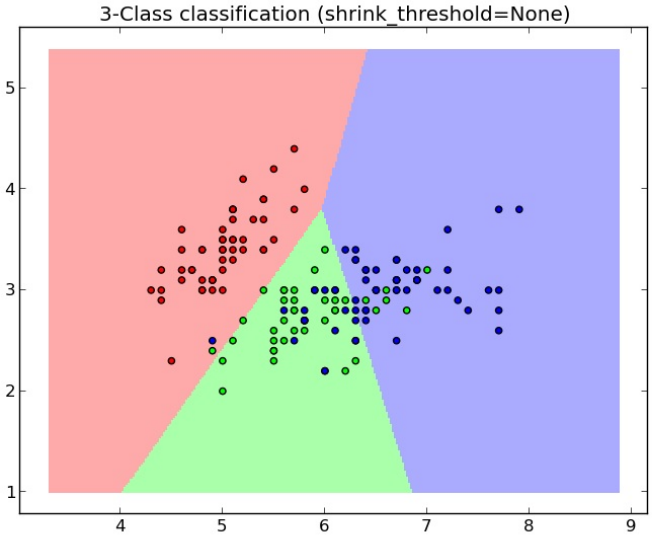
```
>>> from sklearn.neighbors.nearest_centroid import NearestCentroid
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = NearestCentroid()
>>> clf.fit(X, y)
```

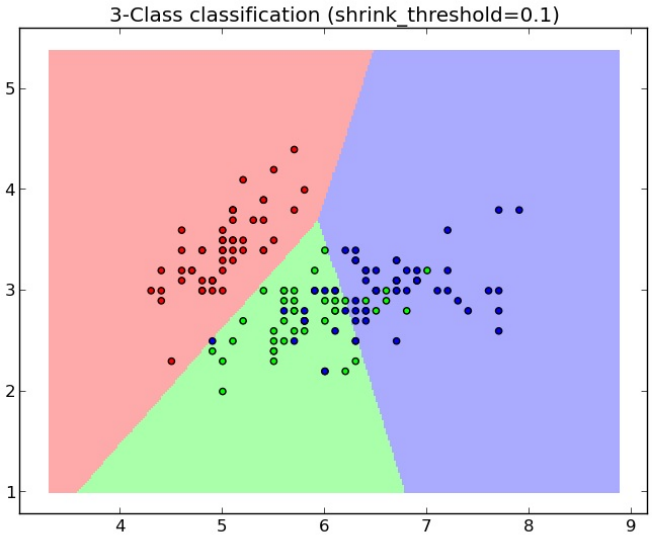
```
NearestCentroid(metric='euclidean', shrink_threshold=None)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

1.4.5.1. Nearest Shrunken Centroid

The **NearestCentroid** classifier has a `shrink_threshold` parameter, which implements the nearest shrunken centroid classifier. In effect, the value of each feature for each centroid is divided by the within-class variance of that feature. The feature values are then reduced by `shrink_threshold`. Most notably, if a particular feature value crosses zero, it is set to zero. In effect, this removes the feature from affecting the classification. This is useful, for example, for removing noisy features.

In the example below, using a small shrink threshold increases the accuracy of the model from 0.81 to 0.82.





Examples:

- *Nearest Centroid Classification*: an example of classification using nearest centroid with different shrink thresholds.

1.5. 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 exceedance 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:

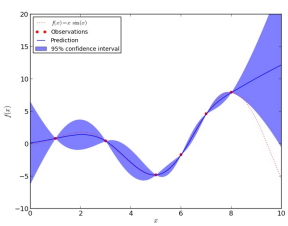
- It is not sparse. It uses the whole samples/features information to perform the prediction.
- It loses efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens. It might indeed give poor performance and it loses computational efficiency.
- Classification is only 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.

1.5.1. Examples

1.5.1.1. 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 instantiation, the fitting procedure may recourse to maximum likelihood estimation for the parameters or alternatively it uses the given parameters.



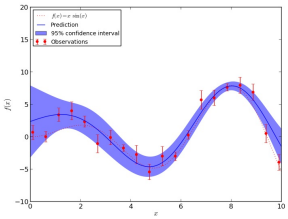
```
>>> import numpy as np
>>> from sklearn import gaussian_process
>>> def f(x):
...     return x * np.sin(x)
>>> X = np.atleast_2d([1., 3., 5., 6., 7., 8.]).T
>>> y = f(X).ravel()
>>> x = np.atleast_2d(np.linspace(0, 10, 1000)).T
>>> gp = gaussian_process.GaussianProcess(theta0=1e-2, thetaL=1e-4, thetaU=1e-1)
>>> gp.fit(X, y)
GaussianProcess(beta0=None, corr=<function squared_exponential at 0x...>,
        normalize=True, nugget=array(2.22...-15),
        optimizer='fmin_cobyla', random_start=1, random_state=...
        regr=<function constant at 0x...>, storage_mode='full',
        theta0=array([[ 0.01]]), thetaL=array([[ 0.0001]]),
        thetaU=array([[ 0.1]]), verbose=False)
>>> y_pred, sigma2_pred = gp.predict(x, eval_MSE=True)
```

1.5.1.2. Fitting Noisy Data

When the data to be fit includes noise, the Gaussian process model can be used by specifying the variance of the noise for each point. `GaussianProcess` takes a parameter `nugget` which is added to the diagonal of the correlation matrix between training points: in general this is a type of Tikhonov regularization. In the special case of a squared-exponential correlation function, this normalization is equivalent to specifying a fractional variance in the input. That is

$$\text{nugget}_i = \left[\frac{\sigma_i}{y_i}\right]^2$$

With `nugget` and `corr` properly set, Gaussian Processes can be used to robustly recover an underlying function from noisy data:



Other examples

- *Gaussian Processes classification example: exploiting the probabilistic output*

1.5.2. Mathematical formulation

1.5.2.1. 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 additionally 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.

1.5.2.2. 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:

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

where we have introduced:

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

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

- 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:

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

- 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.

1.5.2.3. 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 Diettrich)
- The design and analysis of computer experiments TJ Santner, BJ Williams, W Notz Springer, 2003

1.5.3. 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 additionally 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.

1.5.4. 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*.

1.5.5. Implementation details

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

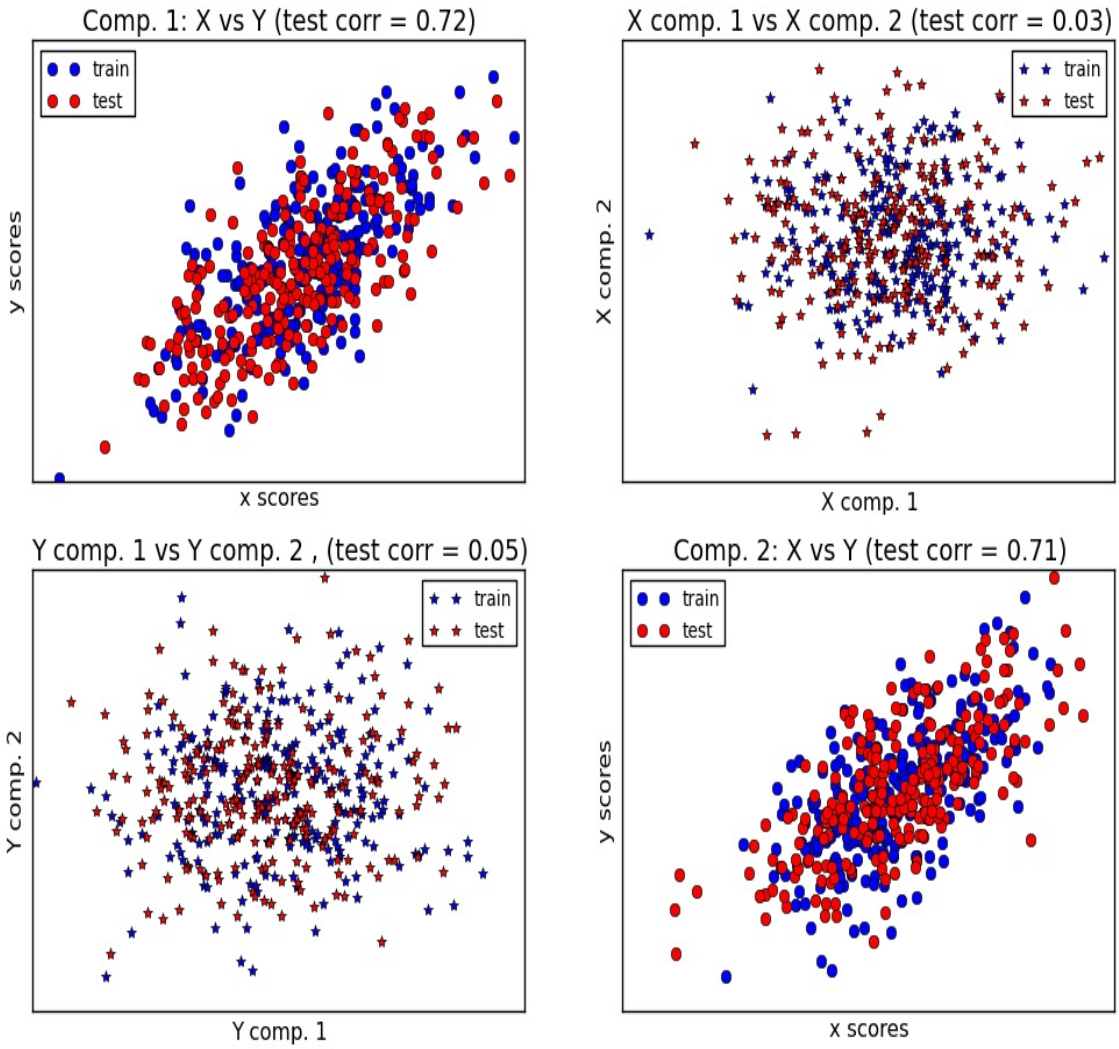
References:

- DACE, A Matlab Kriging Toolbox S Lophaven, HB Nielsen, J Sondergaard 2002,
- W.J. Welch, R.J. Buck, J. Sacks, H.P. Wynn, T.J. Mitchell, and M.D. Morris (1992). Screening, predicting, and computer experiments. Technometrics, 34(1) 15–25.

1.6. Cross decomposition

The cross decomposition module contains two main families of algorithms: the partial least squares (PLS) and the canonical correlation analysis (CCA).

These families of algorithms are useful to find linear relations between two multivariate datasets: the `X` and `Y` arguments of the `fit` method are 2D arrays.



Cross decomposition algorithms find the fundamental relations between two matrices (X and Y). They are latent variable approaches to modeling the covariance structures in these two spaces. They will try to find the multidimensional direction in the X space that explains the maximum multidimensional variance direction in the Y space. PLS-regression is particularly suited when the matrix of predictors has more variables than observations, and when there is multicollinearity among X values. By contrast, standard regression will fail in these cases.

Classes included in this module are `PLSRegression` `PLSCanonical`, `CCA` and `PLSSVD`

Reference:

- JA Wegelin A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case

Examples:

- *Compare cross decomposition methods*

Previous Next

1.7. Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of independence between every pair of features. Given a class variable y and a dependent feature vector x_1 through x_n, Bayes' theorem states the following relationship:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots x_n \mid y)}{P(x_1, \dots, x_n)}$$

Using the naive independence assumption that

$$P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i|y),$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

Since $P(x_1, \dots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$; the former is then the relative frequency of class y in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$.

In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters. (For theoretical reasons why naive Bayes works well, and on which types of data it does, see the references below.)

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality.

On the flip side, although naive Bayes is known as a decent classifier, it is known to be a bad estimator, so the probability outputs from `predict_proba` are not to be taken too seriously.

References:

- H. Zhang (2004). The optimality of Naive Bayes. Proc. FLAIRS.

1.7.1. Gaussian Naive Bayes

`GaussianNB` implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

The parameters σ_y and μ_y are estimated using maximum likelihood.

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
>>> y_pred = gnb.fit(iris.data, iris.target).predict(iris.data)
>>> print("Number of mislabeled points out of a total %d points : %d"
...       % (iris.data.shape[0],(iris.target != y_pred).sum()))
Number of mislabeled points out of a total 150 points : 6
```

1.7.2. Multinomial Naive Bayes

`MultinomialNB` implements the naive Bayes algorithm for multinomially distributed data, and is one of the two classic naive Bayes variants used in text classification (where the data are typically represented as word vector counts, although tf-idf vectors are also known to work well in practice). The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ for each class y, where n is the number of features (in text classification, the size of the vocabulary) and θ_{yi} is the probability $P(x_i \mid y)$ of feature i appearing in a sample belonging to class y.

The parameters θ_y is estimated by a smoothed version of maximum likelihood, i.e. relative frequency counting:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T, and $N_y = \sum_{i=1}^{|T|} N_{yi}$ is the total count of all features for class y.

The smoothing priors $\alpha \geq 0$ accounts for features not present in the learning samples and prevents zero probabilities in further computations. Setting $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing.

1.7.3. Bernoulli Naive Bayes

`BernoulliNB` implements the naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions; i.e., there may be multiple features but each one is assumed

to be a binary-valued (Bernoulli, boolean) variable. Therefore, this class requires samples to be represented as binary-valued feature vectors; if handed any other kind of data, a `BernoulliNB` instance may binarize its input (depending on the `binarize` parameter).

The decision rule for Bernoulli naive Bayes is based on

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i)$$

which differs from multinomial NB's rule in that it explicitly penalizes the non-occurrence of a feature i that is an indicator for class y, where the multinomial variant would simply ignore a non-occurring feature.

In the case of text classification, word occurrence vectors (rather than word count vectors) may be used to train and use this classifier. `BernoulliNB` might perform better on some datasets, especially those with shorter documents. It is advisable to evaluate both models, if time permits.

References:

- C.D. Manning, P. Raghavan and H. Schütze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234-265.
- A. McCallum and K. Nigam (1998). A comparison of event models for Naive Bayes text classification. Proc. AAAI/ICML-98 Workshop on Learning for Text Categorization, pp. 41-48.
- V. Metsis, I. Androutsopoulos and G. Paliouras (2006). Spam filtering with Naive Bayes – Which Naive Bayes? 3rd Conf. on Email and Anti-Spam (CEAS).

1.7.4. Out-of-core naive Bayes model fitting

Discrete naive Bayes models can be used to tackle large scale text classification problems for which the full training set might not fit in memory. To handle this case both **`MultinomialNB`** and **`BernoulliNB`** expose a `partial_fit` method that can be used incrementally as done with other classifiers as demonstrated in *Out-of-core classification of text documents*.

Contrary to the `fit` method, the first call to `partial_fit` needs to be passed the list of all the expected class labels.

For an overview of available strategies in scikit-learn, see also the *out-of-core learning* documentation.

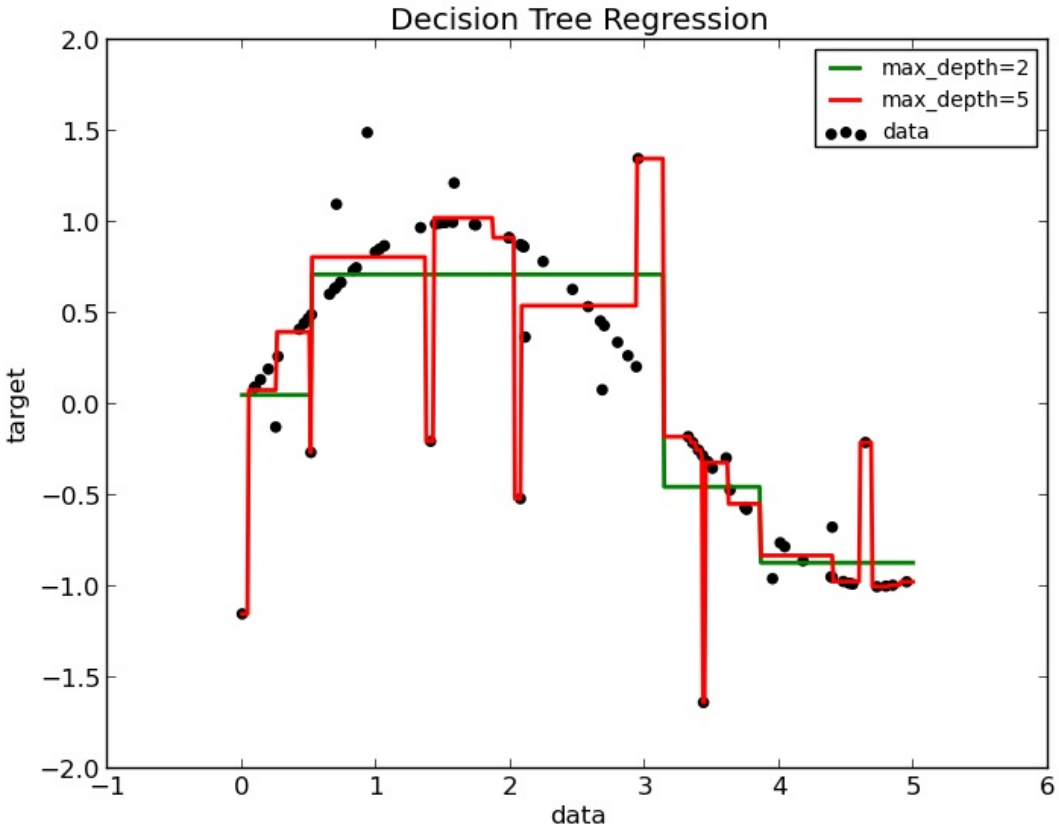
note:

```
The ``partial_fit`` method call of naive Bayes models introduces some
computational overhead. It is recommended to use data chunk sizes that are as
large as possible, that is as the available RAM allows.
```

1.8. Decision Trees

Decision Trees (DTs) are a non-parametric supervised learning method used for *classification* and *regression*. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

For instance, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.



Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualised.
- Requires little data preparation. Other techniques often require data normalisation, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. Other techniques are usually specialised in analysing datasets that have only one type of variable. See *algorithms* for more information.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g.,

in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting. Mechanisms such as pruning (not currently supported), setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

1.8.1. Classification

`DecisionTreeClassifier` is a class capable of performing multi-class classification on a dataset.

As other classifiers, `DecisionTreeClassifier` 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 sklearn import tree
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, Y)
```

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

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

`DecisionTreeClassifier` is capable of both binary (where the labels are [-1, 1]) classification and multiclass (where the labels are [0, ..., K-1]) classification.

Using the Iris dataset, we can construct a tree as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
>>> iris = load_iris()
```

```
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(iris.data, iris.target)
```

Once trained, we can export the tree in Graphviz format using the **`export_graphviz`** exporter. Below is an example export of a tree trained on the entire iris dataset:

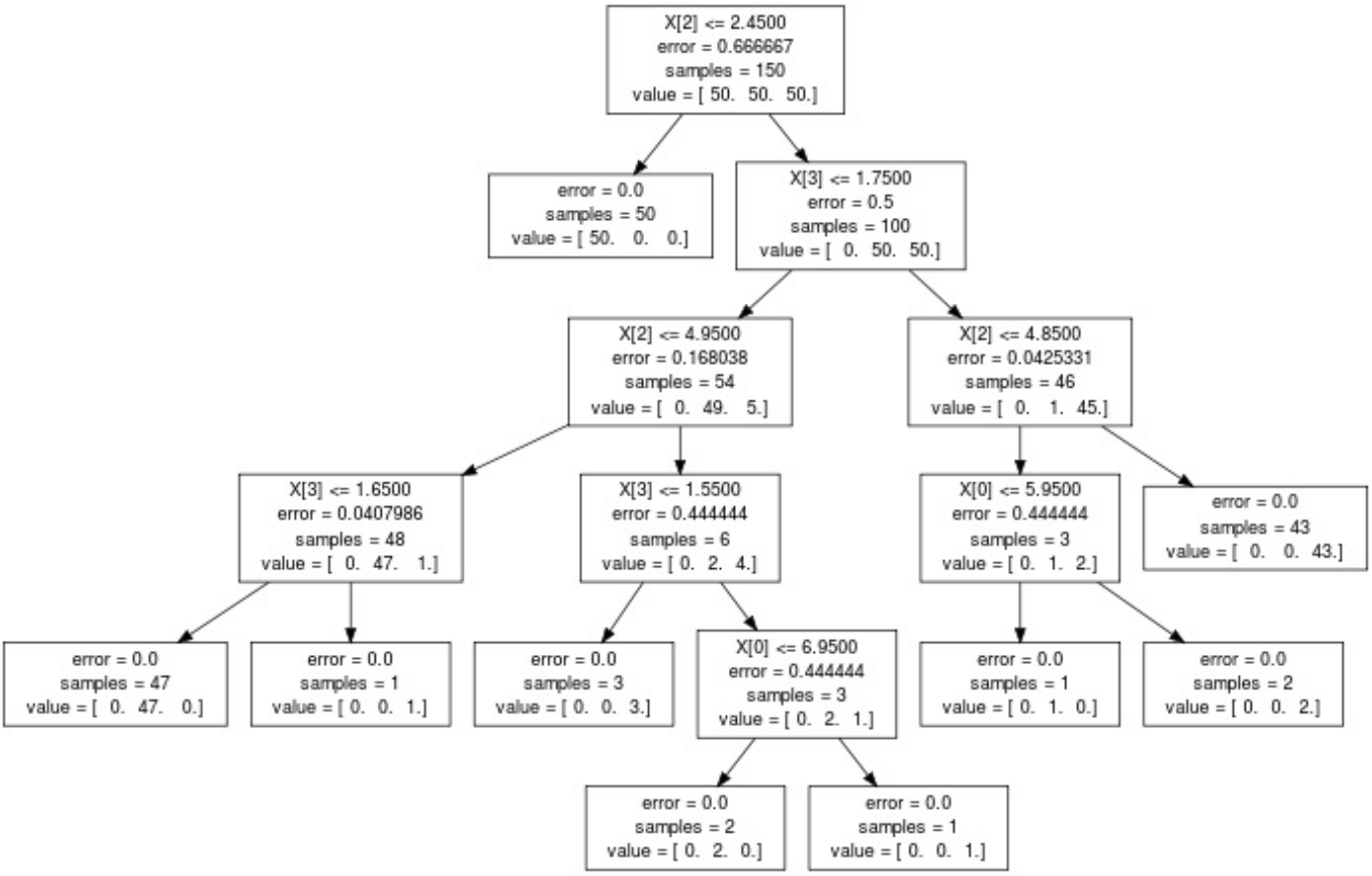
```
>>> from sklearn.externals.six import StringIO
>>> with open("iris.dot", 'w') as f:
...     f = tree.export_graphviz(clf, out_file=f)
```

Then we can use Graphviz's `dot` tool to create a PDF file (or any other supported file type): `dot -Tpdf iris.dot -o iris.pdf`.

```
>>> import os
>>> os.unlink('iris.dot')
```

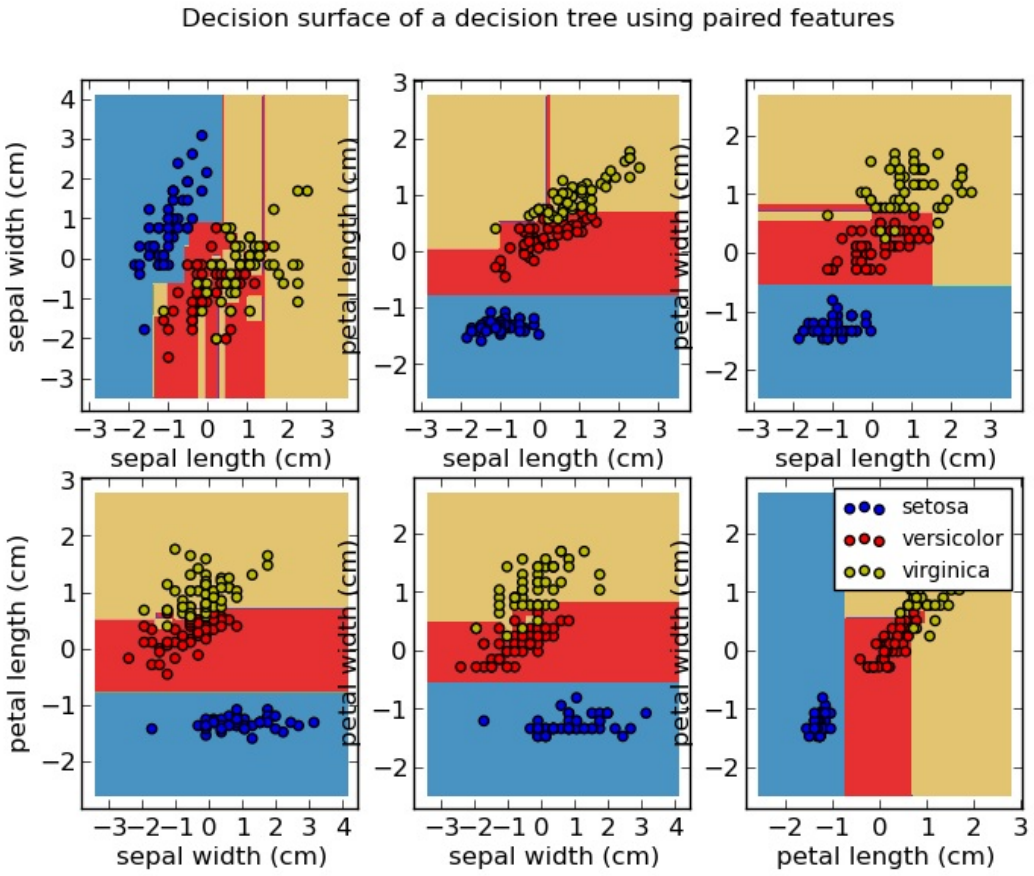
Alternatively, if we have Python module `pydot` installed, we can generate a PDF file (or any other supported file type) directly in Python:

```
>>> from sklearn.externals.six import StringIO
>>> import pydot
>>> dot_data = StringIO()
>>> tree.export_graphviz(clf, out_file=dot_data)
>>> graph = pydot.graph_from_dot_data(dot_data.getvalue())
>>> graph.write_pdf("iris.pdf")
```



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

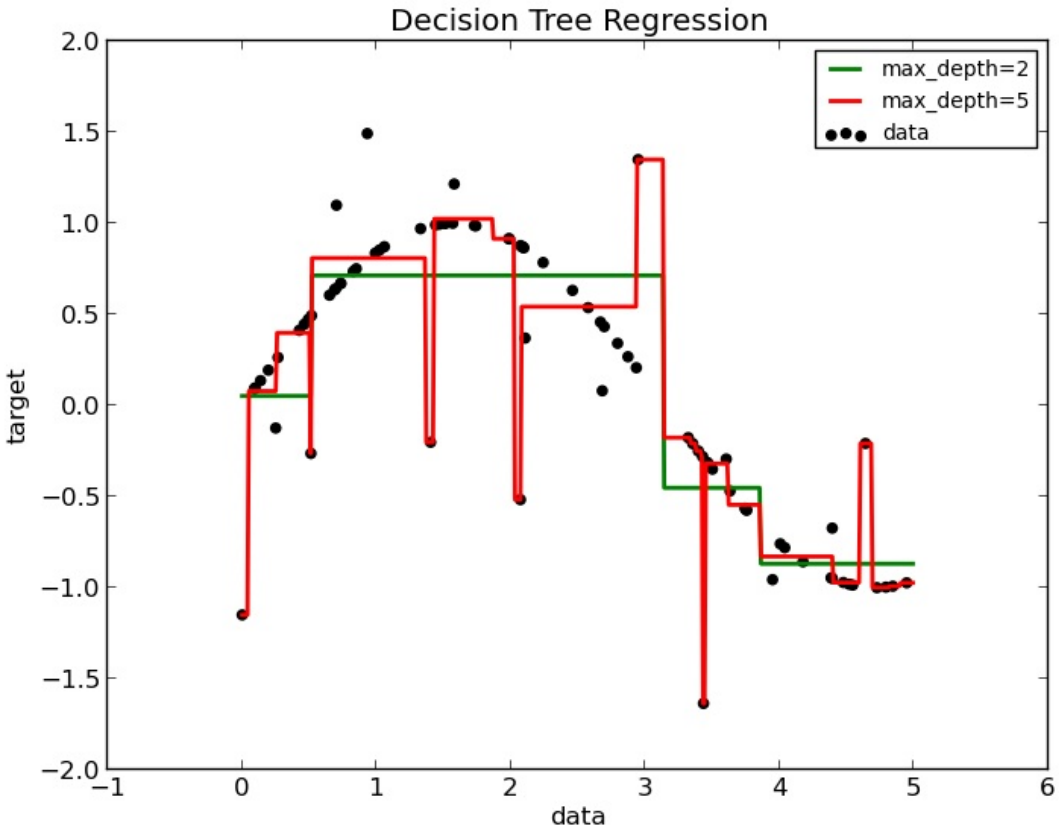
```
>>> clf.predict(iris.data[0, :])
array([0])
```



Examples:

- *Plot the decision surface of a decision tree on the iris dataset*

1.8.2. Regression



Decision trees can also be applied to regression problems, using the `DecisionTreeRegressor` class.

As in the classification setting, the fit method will take as argument arrays X and y, only that in this case y is expected to have floating point values instead of integer values:

```
>>> from sklearn import tree
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> clf = tree.DecisionTreeRegressor()
>>> clf = clf.fit(X, y)
>>> clf.predict([[1, 1]])
array([ 0.5])
```

Examples:

- *Decision Tree Regression*

1.8.3. Multi-output problems

A multi-output problem is a supervised learning problem with several outputs to predict, that is when Y is a 2d array of size `[n_samples, n_outputs]`.

When there is no correlation between the outputs, a very simple way to solve this kind of problem is to build n independent models, i.e. one for each output, and then to use those models to independently predict each one of the n outputs. However, because it is likely that the output values related to the same input are themselves correlated, an often better way is to build a single model capable of predicting simultaneously all n outputs. First, it requires lower training time since only a single estimator is built. Second, the generalization accuracy of the resulting estimator may often be increased.

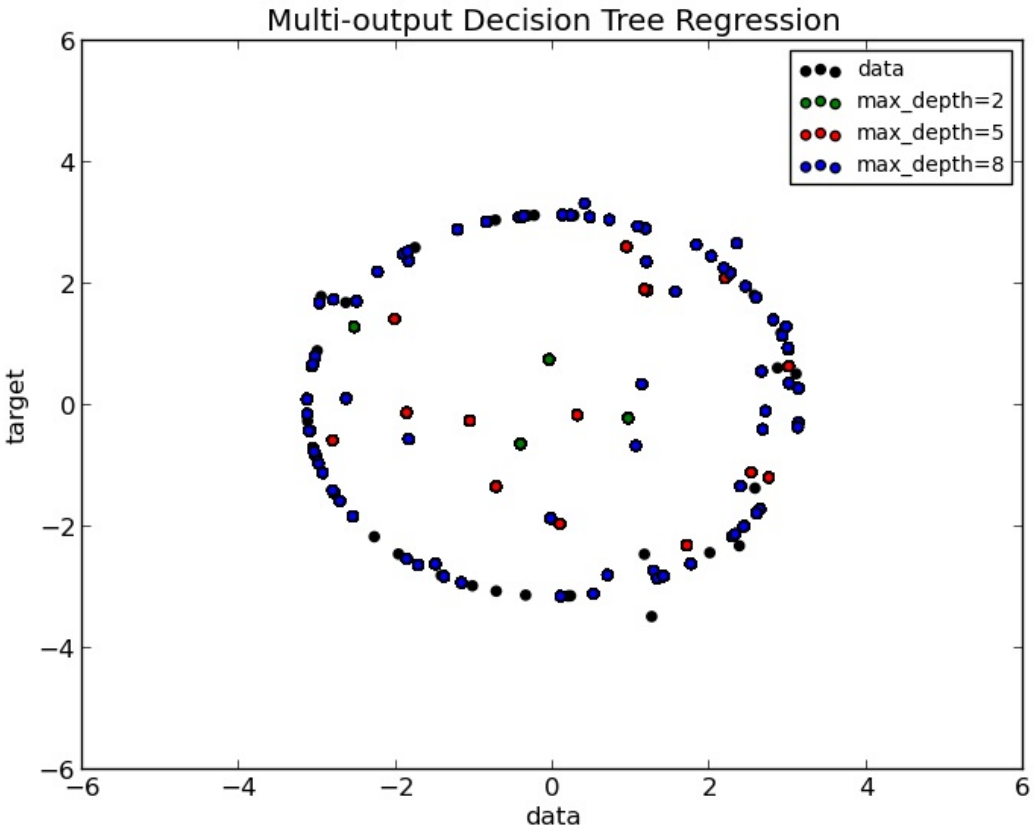
With regard to decision trees, this strategy can readily be used to support multi-output problems. This requires the following changes:

- Store n output values in leaves, instead of 1;
- Use splitting criteria that compute the average reduction across all n outputs.

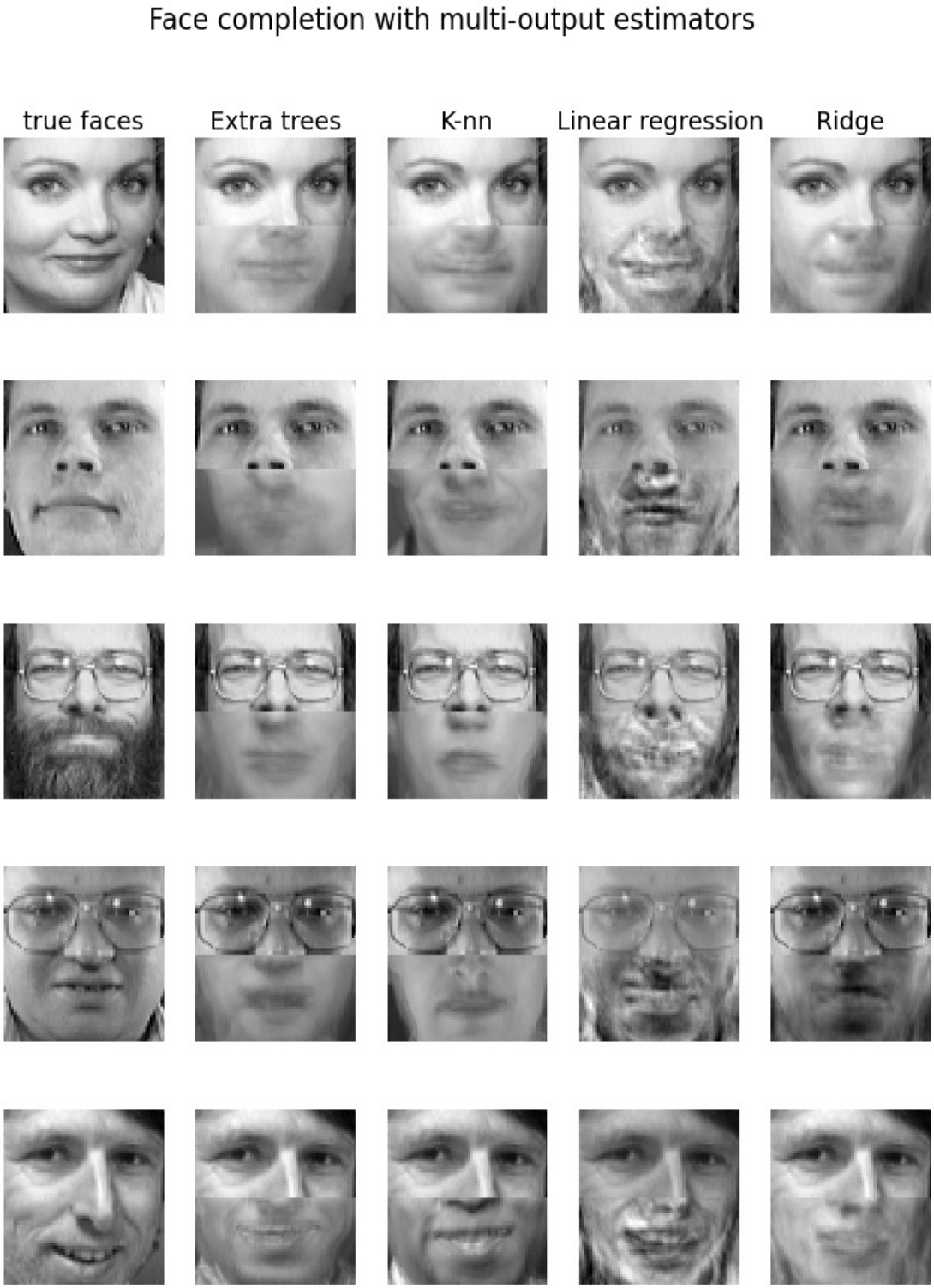
This module offers support for multi-output problems by implementing this strategy in both **`DecisionTreeClassifier`** and **`DecisionTreeRegressor`**. If a decision tree is fit on an output array Y of size `[n_samples, n_outputs]` then the resulting estimator will:

- Output n_output values upon `predict`;
- Output a list of n_output arrays of class probabilities upon `predict_proba`.

The use of multi-output trees for regression is demonstrated in *Multi-output Decision Tree Regression*. In this example, the input X is a single real value and the outputs Y are the sine and cosine of X.



The use of multi-output trees for classification is demonstrated in *Face completion with a multi-output estimators*. In this example, the inputs X are the pixels of the upper half of faces and the outputs Y are the pixels of the lower half of those faces.



Examples:

- *Multi-output Decision Tree Regression*
- *Face completion with a multi-output estimators*

References:

- M. Dumont et al, Fast multi-class image annotation with random subwindows and multiple output

randomized trees, International Conference on Computer Vision Theory and Applications 2009

1.8.4. Complexity

In general, the run time cost to construct a balanced binary tree is $O(n_{samples}n_{features}\log(n_{samples}))$ and query time $O(\log(n_{samples}))$. Although the tree construction algorithm attempts to generate balanced trees, they will not always be balanced. Assuming that the subtrees remain approximately balanced, the cost at each node consists of searching through $O(n_{features})$ to find the feature that offers the largest reduction in entropy. This has a cost of $O(n_{features}n_{samples}\log(n_{samples}))$ at each node, leading to a total cost over the entire trees (by summing the cost at each node) of $O(n_{features}n^2_{samples}\log(n_{samples}))$.

Scikit-learn offers a more efficient implementation for the construction of decision trees. A naive implementation (as above) would recompute the class label histograms (for classification) or the means (for regression) at for each new split point along a given feature. By presorting the feature over all relevant samples, and retaining a running label count, we reduce the complexity at each node to $O(n_{features}\log(n_{samples}))$, which results in a total cost of $O(n_{features}n_{samples}\log(n_{samples}))$.

1.8.5. Tips on practical use

- Decision trees tend to overfit on data with a large number of features. Getting the right ratio of samples to number of features is important, since a tree with few samples in high dimensional space is very likely to overfit.
- Consider performing dimensionality reduction (*PCA*, *ICA*, or *Feature selection*) beforehand to give your tree a better chance of finding features that are discriminative.
- Visualise your tree as you are training by using the `export` function. Use `max_depth=3` as an initial tree depth to get a feel for how the tree is fitting to your data, and then increase the depth.
- Remember that the number of samples required to populate the tree doubles for each additional level the tree grows to. Use `max_depth` to control the size of the tree to prevent overfitting.
- Use `min_samples_split` or `min_samples_leaf` to control the number of samples at a leaf node. A very small number will usually mean the tree will overfit, whereas a large number will prevent the tree from learning the data. Try `min_samples_leaf=5` as an initial value. The main difference between the two is that `min_samples_leaf` guarantees a minimum number of samples in a leaf, while `min_samples_split` can create arbitrary small leaves, though `min_samples_split` is more common in the literature.
- Balance your dataset before training to prevent the tree from creating a tree biased toward the classes that are dominant.
- All decision trees use `np.float32` arrays internally. If training data is not in this format, a copy of the dataset will be made.

1.8.6. Tree algorithms: ID3, C4.5, C5.0 and CART

What are all the various decision tree algorithms and how do they differ from each other? Which one is implemented in scikit-learn?

ID3 (Iterative Dichotomiser 3) was developed in 1986 by Ross Quinlan. The algorithm creates a multiway tree, finding for each node (i.e. in a greedy manner) the categorical feature that will yield the largest information gain for categorical targets. Trees are grown to their maximum size and then a pruning step is usually applied to improve the ability of the tree to generalise to unseen data.

C4.5 is the successor to ID3 and removed the restriction that features must be categorical by dynamically defining a discrete attribute (based on numerical variables) that partitions the continuous attribute value into a discrete set of intervals. C4.5 converts the trained trees (i.e. the output of the ID3 algorithm) into sets of if-then rules. These accuracy of each rule is then evaluated to determine the order in which they should be applied. Pruning is done by removing a rule's precondition if the accuracy of the rule improves without it.

C5.0 is Quinlan's latest version release under a proprietary license. It uses less memory and builds smaller rulesets than C4.5 while being more accurate.

CART (Classification and Regression Trees) is very similar to C4.5, but it differs in that it supports numerical target variables (regression) and does not compute rule sets. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node.

scikit-learn uses an optimised version of the CART algorithm.

1.8.7. Mathematical formulation

Given training vectors $x_i \in R^n$, i=1,..., l and a label vector $y \in R^l$, a decision tree recursively partitions the space such that the samples with the same labels are grouped together.

Let the data at node m be represented by Q. For each candidate split $\theta = (j, t_m)$ consisting of a feature j and threshold t_m, partition the data into $Q_{left}(\theta)$ and $Q_{right}(\theta)$ subsets

$$Q_{left}(\theta) = (x, y)|x_j <= t_m$$
$$Q_{right}(\theta) = Q \setminus Q_{left}(\theta)$$

The impurity at m is computed using an impurity function $H()$, the choice of which depends on the task being solved (classification or regression)

$$G(Q, \theta) = \frac{n_{left}}{N_m} H(Q_{left}(\theta)) + \frac{n_{right}}{N_m} H(Q_{right}(\theta))$$

Select the parameters that minimises the impurity

$$\theta^* = \operatorname{argmin}_\theta G(Q, \theta)$$

Recurse for subsets $Q_{left}(\theta^*)$ and $Q_{right}(\theta^*)$ until the maximum allowable depth is reached, $N_m < \min_{samples}$ or $N_m = 1$.

1.8.7.1. Classification criteria

If a target is a classification outcome taking on values 0,1,...,K-1, for node m, representing a region R_m with N_m observations, let

$$p_{mk} = 1/N_m \sum_{x_i \in R_m} I(y_i = k)$$

be the proportion of class k observations in node m

Common measures of impurity are Gini

$$H(X_m) = \sum_k p_{mk}(1 - p_{mk})$$

Cross-Entropy

$$H(X_m) = \sum_k p_{mk} \log(p_{mk})$$

and Misclassification

$$H(X_m) = 1 - \max(p_{mk})$$

1.8.7.2. Regression criteria

If the target is a continuous value, then for node m, representing a region R_m with N_m observations, a common criterion to minimise is the Mean Squared Error

$$c_m = \frac{1}{N_m} \sum_{i \in N_m} y_i$$

$$H(X_m) = \frac{1}{N_m} \sum_{i \in N_m} (y_i - c_m)^2$$

References:

- http://en.wikipedia.org/wiki/Decision_tree_learning
- http://en.wikipedia.org/wiki/Predictive_analytics
- L. Breiman, J. Friedman, R. Olshen, and C. Stone. Classification and Regression Trees. Wadsworth, Belmont, CA, 1984.
- J.R. Quinlan. C4. 5: programs for machine learning. Morgan Kaufmann, 1993.
- T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning, Springer, 2009.

Previous Next

1.9. Ensemble methods

The goal of **ensemble methods** is to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator.

Two families of ensemble methods are usually distinguished:

- In **averaging methods**, the driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.

  Examples: *Bagging methods*, *Forests of randomized trees*, ...
- By contrast, in **boosting methods**, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble.

  Examples: *AdaBoost*, *Gradient Tree Boosting*, ...

1.9.1. Bagging meta-estimator

In ensemble algorithms, bagging methods form a class of algorithms which build several instances of a black-box estimator on random subsets of the original training set and then aggregate their individual predictions to form a final prediction. These methods are used as a way to reduce the variance of a base estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it. In many cases, bagging methods constitute a very simple way to improve with respect to a single model, without making it necessary to adapt the underlying base algorithm. As they provide a way to reduce overfitting, bagging methods work best with strong and complex models (e.g., fully developed decision trees), in contrast with boosting methods which usually work best with weak models (e.g., shallow decision trees).

Bagging methods come in many flavours but mostly differ from each other by the way they draw random subsets of the training set:

- When random subsets of the dataset are drawn as random subsets of the samples, then this algorithm is known as Pasting [B1999].
- When samples are drawn with replacement, then the method is known as Bagging [B1996].
- When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces [H1998].
- Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches [LG2012].

In scikit-learn, bagging methods are offered as a unified `BaggingClassifier` meta-estimator (resp. `BaggingRegressor`), taking as input a user-specified base estimator along with parameters specifying the

strategy to draw random subsets. In particular, `max_samples` and `max_features` control the size of the subsets (in terms of samples and features), while `bootstrap` and `bootstrap_features` control whether samples and features are drawn with or without replacement. As an example, the snippet below illustrates how to instantiate a bagging ensemble of **`KNeighborsClassifier`** base estimators, each built on random subsets of 50% of the samples and 50% of the features.

```
>>> from sklearn.ensemble import BaggingClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> bagging = BaggingClassifier(KNeighborsClassifier(),
...                             max_samples=0.5, max_features=0.5)
```

Examples:

- *Single estimator versus bagging: bias-variance decomposition*

References

[B1999] L. Breiman, “Pasting small votes for classification in large databases and on-line”, Machine Learning, 36(1), 85-103, 1999.

[B1996] L. Breiman, “Bagging predictors”, Machine Learning, 24(2), 123-140, 1996.

[H1998] T. Ho, “The random subspace method for constructing decision forests”, Pattern Analysis and Machine Intelligence, 20(8), 832-844, 1998.

[LG2012] G. Louppe and P. Geurts, “Ensembles on Random Patches”, Machine Learning and Knowledge Discovery in Databases, 346-361, 2012.

1.9.2. Forests of randomized trees

The **`sklearn.ensemble`** module includes two averaging algorithms based on randomized *decision trees*: the RandomForest algorithm and the Extra-Trees method. Both algorithms are perturb-and-combine techniques [B1998] specifically designed for trees. This means a diverse set of classifiers is created by introducing randomness in the classifier construction. The prediction of the ensemble is given as the averaged prediction of the individual classifiers.

As other classifiers, forest classifiers have 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 sklearn.ensemble import RandomForestClassifier
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = RandomForestClassifier(n_estimators=10)
>>> clf = clf.fit(X, Y)
```

Like *decision trees*, forests of trees also extend to *multi-output problems* (if Y is an array of size `[n_samples, n_outputs]`).

1.9.2.1. Random Forests

In random forests (see **`RandomForestClassifier`** and **`RandomForestRegressor`** classes), each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set. In addition, when splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a random subset of the features. As a result of this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random tree) but, due to averaging, its variance also decreases, usually more than

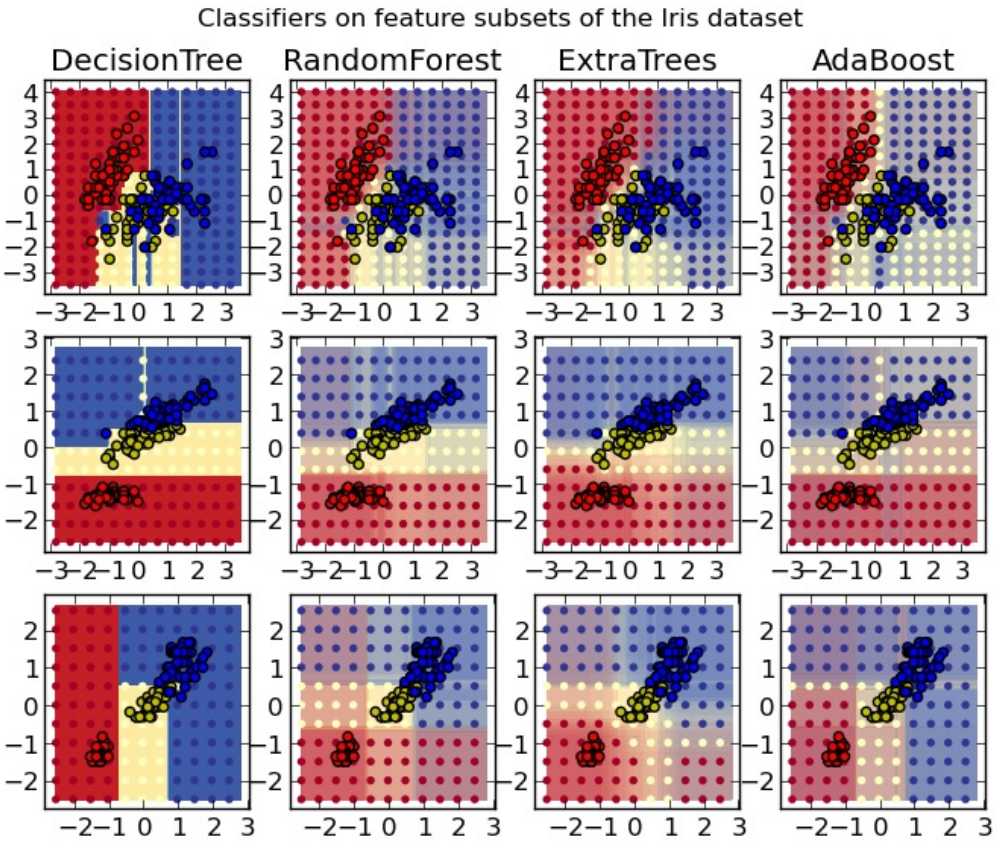
compensating for the increase in bias, hence yielding an overall better model.

In contrast to the original publication [B2001], the scikit-learn implementation combines classifiers by averaging their probabilistic prediction, instead of letting each classifier vote for a single class.

1.9.2.2. Extremely Randomized Trees

In extremely randomized trees (see **ExtraTreesClassifier** and **ExtraTreesRegressor** classes), randomness goes one step further in the way splits are computed. As in random forests, a random subset of candidate features is used, but instead of looking for the most discriminative thresholds, thresholds are drawn at random for each candidate feature and the best of these randomly-generated thresholds is picked as the splitting rule. This usually allows to reduce the variance of the model a bit more, at the expense of a slightly greater increase in bias:

```
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.datasets import make_blobs
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.tree import DecisionTreeClassifier

>>> X, y = make_blobs(n_samples=10000, n_features=10, centers=100,
...     random_state=0)

>>> clf = DecisionTreeClassifier(max_depth=None, min_samples_split=1,
...     random_state=0)
>>> scores = cross_val_score(clf, X, y)
>>> scores.mean()
0.97...

>>> clf = RandomForestClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=1, random_state=0)
>>> scores = cross_val_score(clf, X, y)
>>> scores.mean()
0.999...

>>> clf = ExtraTreesClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=1, random_state=0)
>>> scores = cross_val_score(clf, X, y)
>>> scores.mean() > 0.999
True
```



1.9.2.3. Parameters

The main parameters to adjust when using these methods is `n_estimators` and `max_features`. The former is the number of trees in the forest. The larger the better, but also the longer it will take to compute. In addition, note that results will stop getting significantly better beyond a critical number of trees. The latter is the size of the random subsets of features to consider when splitting a node. The lower the greater the reduction of variance, but also the greater the increase in bias. Empirical good default values are `max_features=n_features` for regression problems, and `max_features=sqrt(n_features)` for classification tasks (where `n_features` is the number of features in the data). The best results are also usually reached when setting `max_depth=None` in combination with `min_samples_split=1` (i.e., when fully developing the trees). Bear in mind though that these values are usually not optimal. The best parameter values should always be cross- validated. In addition, note that bootstrap samples are used by default in random forests (`bootstrap=True`) while the default strategy is to use the original dataset for building extra-trees (`bootstrap=False`).

1.9.2.4. Parallelization

Finally, this module also features the parallel construction of the trees and the parallel computation of the predictions through the `n_jobs` parameter. If `n_jobs=k` then computations are partitioned into `k` jobs, and run on `k` cores of the machine. If `n_jobs=-1` then all cores available on the machine are used. Note that because of inter-process communication overhead, the speedup might not be linear (i.e., using `k` jobs will unfortunately not be `k` times as fast). Significant speedup can still be achieved though when building a large number of trees, or when building a single tree requires a fair amount of time (e.g., on large datasets).

Examples:

- *Plot the decision surfaces of ensembles of trees on the iris dataset*
- *Pixel importances with a parallel forest of trees*
- *Face completion with a multi-output estimators*

References

[B2001] L. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.

[B1998] L. Breiman, “Arcing Classifiers”, Annals of Statistics 1998.

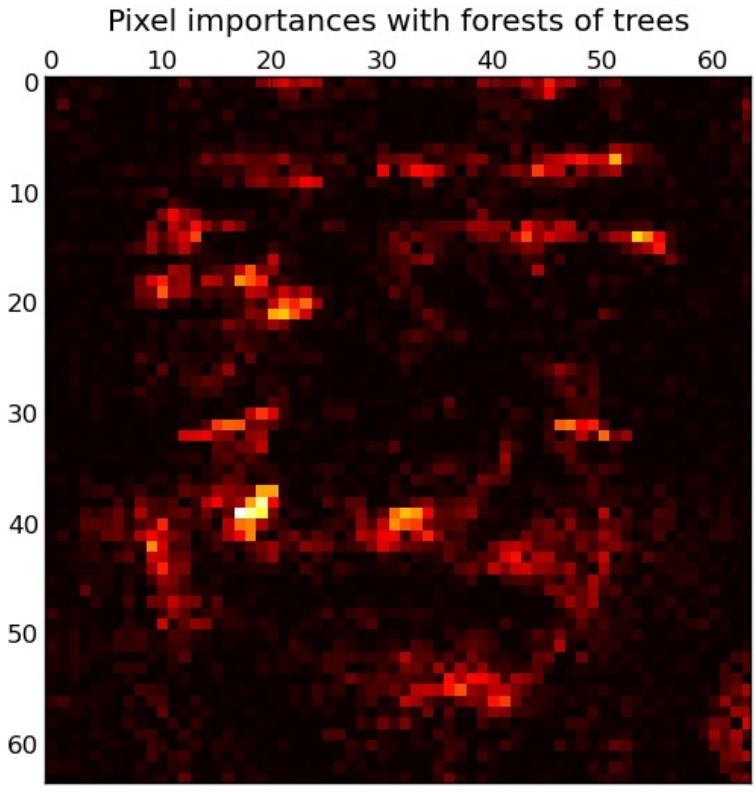
[GEW2006] P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, Machine Learning, 63(1), 3-42, 2006.

1.9.2.5. Feature importance evaluation

The relative rank (i.e. depth) of a feature used as a decision node in a tree can be used to assess the relative importance of that feature with respect to the predictability of the target variable. Features used at the top of the tree are used contribute to the final prediction decision of a larger fraction of the input samples. The **expected fraction of the samples** they contribute to can thus be used as an estimate of the **relative importance of the features**.

By **averaging** those expected activity rates over several randomized trees one can **reduce the variance** of such an estimate and use it for feature selection.

The following example shows a color-coded representation of the relative importances of each individual pixel for a face recognition task using a `ExtraTreesClassifier` model.



In practice those estimates are stored as an attribute named `feature_importances_` on the fitted model. This is an array with shape `(n_features,)` whose values are positive and sum to 1.0. The higher the value,

the more important is the contribution of the matching feature to the prediction function.

Examples:

- *Pixel importances with a parallel forest of trees*
- *Feature importances with forests of trees*

1.9.2.6. Totally Random Trees Embedding

`RandomTreesEmbedding` implements an unsupervised transformation of the data. Using a forest of completely random trees, `RandomTreesEmbedding` encodes the data by the indices of the leaves a data point ends up in. This index is then encoded in a one-of-K manner, leading to a high dimensional, sparse binary coding. This coding can be computed very efficiently and can then be used as a basis for other learning tasks. The size and sparsity of the code can be influenced by choosing the number of trees and the maximum depth per tree. For each tree in the ensemble, the coding contains one entry of one. The size of the coding is at most `n_estimators * 2 ** max_depth`, the maximum number of leaves in the forest.

As neighboring data points are more likely to lie within the same leaf of a tree, the transformation performs an implicit, non-parametric density estimation.

Examples:

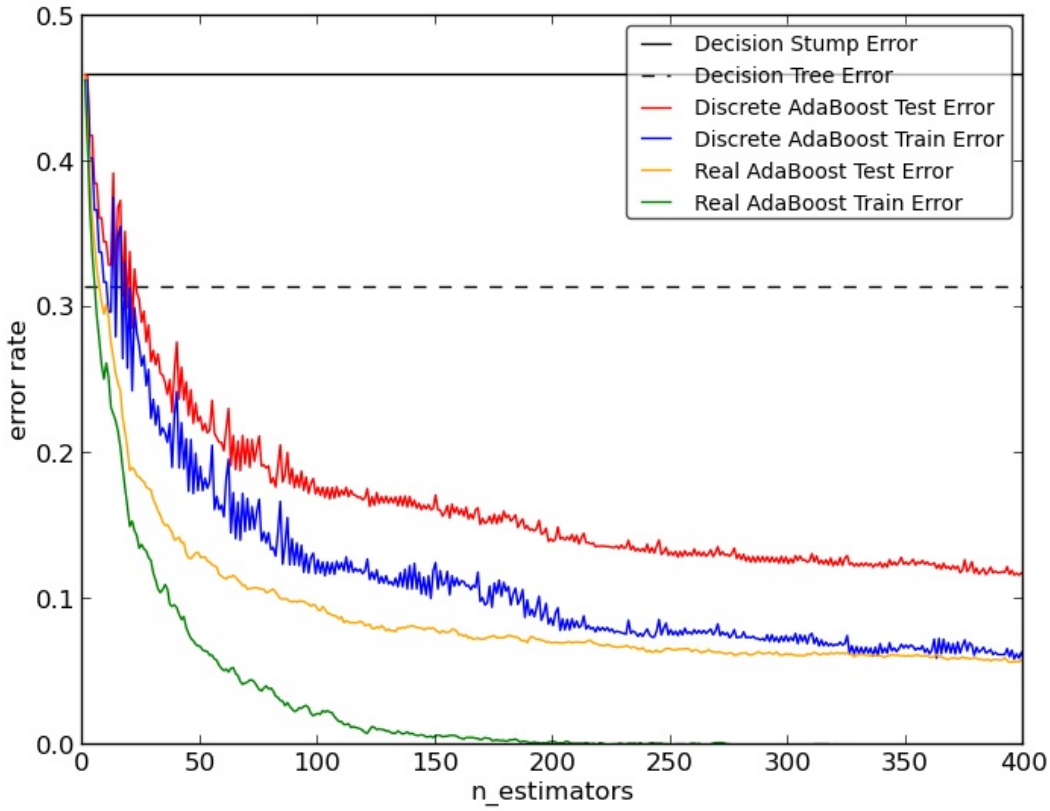
- *Hashing feature transformation using Totally Random Trees*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...* compares non-linear dimensionality reduction techniques on handwritten digits.

See also: *Manifold learning* techniques can also be useful to derive non-linear representations of feature space, also these approaches focus also on dimensionality reduction.

1.9.3. AdaBoost

The module `sklearn.ensemble` includes the popular boosting algorithm AdaBoost, introduced in 1995 by Freund and Schapire [FS1995].

The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each so-called boosting iteration consist of applying weights w_1, w_2, ..., w_N to each of the training samples. Initially, those weights are all set to $w_i = 1/N$, so that the first step simply trains a weak learner on the original data. For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence [HTF].



AdaBoost can be used both for classification and regression problems:

- For multi-class classification, **`AdaBoostClassifier`** implements AdaBoost-SAMME and AdaBoost-SAMME.R [ZZRH2009].
- For regression, **`AdaBoostRegressor`** implements AdaBoost.R2 [D1997].

1.9.3.1. Usage

The following example shows how to fit an AdaBoost classifier with 100 weak learners:

```
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.datasets import load_iris
>>> from sklearn.ensemble import AdaBoostClassifier

>>> iris = load_iris()
>>> clf = AdaBoostClassifier(n_estimators=100)
>>> scores = cross_val_score(clf, iris.data, iris.target)
>>> scores.mean()
0.9...
```

The number of weak learners is controlled by the parameter `n_estimators`. The `learning_rate` parameter controls the contribution of the weak learners in the final combination. By default, weak learners are decision stumps. Different weak learners can be specified through the `base_estimator` parameter. The main parameters to tune to obtain good results are `n_estimators` and the complexity of the base estimators (e.g., its depth `max_depth` or minimum required number of samples at a leaf `min_samples_leaf` in case of decision trees).

Examples:

- *Discrete versus Real AdaBoost* compares the classification error of a decision stump, decision tree, and a boosted decision stump using AdaBoost-SAMME and AdaBoost-SAMME.R.

- *Multi-class AdaBoosted Decision Trees* shows the performance of AdaBoost-SAMME and AdaBoost-SAMME.R on a multi-class problem.
- *Two-class AdaBoost* shows the decision boundary and decision function values for a non-linearly separable two-class problem using AdaBoost-SAMME.
- *Decision Tree Regression with AdaBoost* demonstrates regression with the AdaBoost.R2 algorithm.

References

[FS1995] Y. Freund, and R. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting", 1997.

[ZZRH2009] J. Zhu, H. Zou, S. Rosset, T. Hastie. "Multi-class AdaBoost", 2009.

[D1997] H. Drucker. "Improving Regressors using Boosting Techniques", 1997.

[HTF] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

1.9.4. Gradient Tree Boosting

Gradient Tree Boosting or Gradient Boosted Regression Trees (GBRT) is a generalization of boosting to arbitrary differentiable loss functions. GBRT is an accurate and effective off-the-shelf procedure that can be used for both regression and classification problems. Gradient Tree Boosting models are used in a variety of areas including Web search ranking and ecology.

The advantages of GBRT are:

- Natural handling of data of mixed type (= heterogeneous features)
- Predictive power
- Robustness to outliers in output space (via robust loss functions)

The disadvantages of GBRT are:

- Scalability, due to the sequential nature of boosting it can hardly be parallelized.

The module **`sklearn.ensemble`** provides methods for both classification and regression via gradient boosted regression trees.

1.9.4.1. Classification

`GradientBoostingClassifier` supports both binary and multi-class classification via the deviance loss function (`loss='deviance'`). The following example shows how to fit a gradient boosting classifier with 100 decision stumps as weak learners:

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> X_train, X_test = X[:2000], X[2000:]
>>> y_train, y_test = y[:2000], y[2000:]

>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.913...
```

The number of weak learners (i.e. regression trees) is controlled by the parameter `n_estimators`; *The size*

of each tree can be controlled either by setting the tree depth via `max_depth` or by setting the number of leaf nodes via `max_leaf_nodes`. The `learning_rate` is a hyper-parameter in the range (0.0, 1.0] that controls overfitting via *shrinkage* .

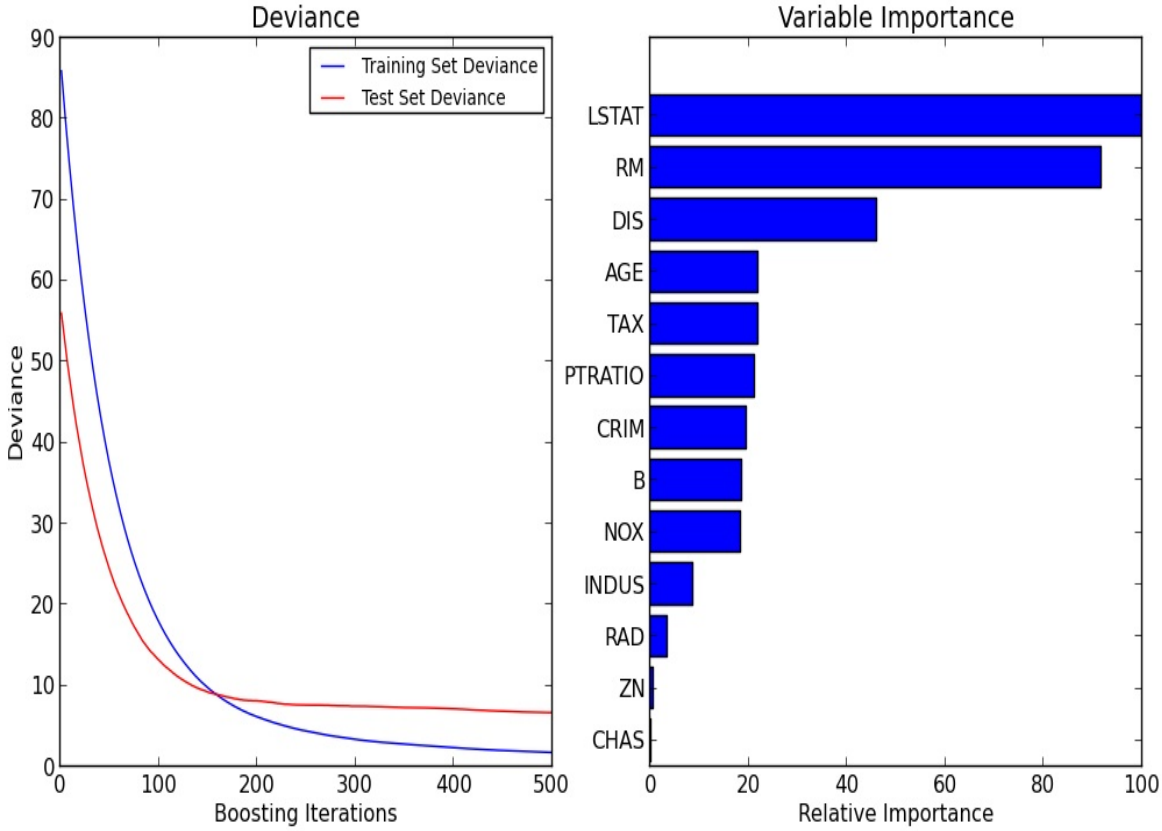
Note: Classification with more than 2 classes requires the induction of `n_classes` regression trees at each at each iteration, thus, the total number of induced trees equals `n_classes * n_estimators`. For datasets with a large number of classes we strongly recommend to use **`RandomForestClassifier`** as an alternative to **`GradientBoostingClassifier`** .

1.9.4.2. Regression

`GradientBoostingRegressor` supports a number of *different loss functions* for regression which can be specified via the argument `loss`; the default loss function for regression is least squares (`'ls'`).

```
>>> import numpy as np
>>> from sklearn.metrics import mean_squared_error
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor

>>> X, y = make_friedman1(n_samples=1200, random_state=0, noise=1.0)
>>> X_train, X_test = X[:200], X[200:]
>>> y_train, y_test = y[:200], y[200:]
>>> est = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,
...     max_depth=1, random_state=0, loss='ls').fit(X_train, y_train)
>>> mean_squared_error(y_test, est.predict(X_test))
5.00...
```

The figure below shows the results of applying **`GradientBoostingRegressor`** with least squares loss and 500 base learners to the Boston house price dataset (**`sklearn.datasets.load_boston`**). The plot on the left shows the train and test error at each iteration. The train error at each iteration is stored in the **`train_score_`** attribute of the gradient boosting model. The test error at each iterations can be obtained via the **`staged_predict`** method which returns a generator that yields the predictions at each stage. Plots like these can be used to determine the optimal number of trees (i.e. `n_estimators`) by early stopping. The plot on the right shows the feature importances which can be obtained via the `feature_importances_` property.



Examples:

- *Gradient Boosting regression*
- *Gradient Boosting Out-of-Bag estimates*

1.9.4.3. Fitting additional weak-learners

Both `GradientBoostingRegressor` and `GradientBoostingClassifier` support `warm_start=True` which allows you to add more estimators to an already fitted model.

```
>>> _ = est.set_params(n_estimators=200, warm_start=True)  # set warm_start and new nr of trees
>>> _ = est.fit(X_train, y_train) # fit additional 100 trees to est
>>> mean_squared_error(y_test, est.predict(X_test))
3.84...
```

1.9.4.4. Controlling the tree size

The size of the regression tree base learners defines the level of variable interactions that can be captured by the gradient boosting model. In general, a tree of depth `h` can capture interactions of order `h` . There are two ways in which the size of the individual regression trees can be controlled.

If you specify `max_depth=h` then complete binary trees of depth `h` will be grown. Such trees will have (at most) `2**h` leaf nodes and `2**h - 1` split nodes.

Alternatively, you can control the tree size by specifying the number of leaf nodes via the parameter `max_leaf_nodes`. In this case, trees will be grown using best-first search where nodes with the highest improvement in impurity will be expanded first. A tree with `max_leaf_nodes=k` has `k - 1` split nodes and thus can model interactions of up to order `max_leaf_nodes - 1` .

We found that `max_leaf_nodes=k` gives comparable results to `max_depth=k-1` but is significantly faster to train at the expense of a slightly higher training error. The parameter `max_leaf_nodes` corresponds to the variable `J` in the chapter on gradient boosting in [F2001] and is related to the parameter `interaction.depth` in R's gbm package where `max_leaf_nodes == interaction.depth + 1`.

1.9.4.5. Mathematical formulation

GBRT considers additive models of the following form:

$$F(x) = \sum_{m=1}^{M} \gamma_m h_m(x)$$

where $h_m(x)$ are the basis functions which are usually called *weak learners* in the context of boosting. Gradient Tree Boosting uses *decision trees* of fixed size as weak learners. Decision trees have a number of abilities that make them valuable for boosting, namely the ability to handle data of mixed type and the ability to model complex functions.

Similar to other boosting algorithms GBRT builds the additive model in a forward stagewise fashion:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

At each stage the decision tree $h_m(x)$ is chosen to minimize the loss function L given the current model F_{m-1} and its fit $F_{m-1}(x_i)$

$$F_m(x) = F_{m-1}(x) + \arg\min_h \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) - h(x))$$

The initial model F_0 is problem specific, for least-squares regression one usually chooses the mean of the target values.

> **Note:** The initial model can also be specified via the `init` argument. The passed object has to implement `fit` and `predict`.

Gradient Boosting attempts to solve this minimization problem numerically via steepest descent: The steepest descent direction is the negative gradient of the loss function evaluated at the current model F_{m-1} which can be calculated for any differentiable loss function:

$$F_m(x) = F_{m-1}(x) + \gamma_m \sum_{i=1}^{n} \nabla_F L(y_i, F_{m-1}(x_i))$$

Where the step length γ_m is chosen using line search:

$$\gamma_m = \arg\min_\gamma \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) - \gamma \frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)})$$

The algorithms for regression and classification only differ in the concrete loss function used.

1.9.4.5.1. Loss Functions

The following loss functions are supported and can be specified using the parameter `loss`:

- Regression
  - Least squares (`'ls'`): The natural choice for regression due to its superior computational properties. The initial model is given by the mean of the target values.
  - Least absolute deviation (`'lad'`): A robust loss function for regression. The initial model is given by the median of the target values.
  - Huber (`'huber'`): Another robust loss function that combines least squares and least absolute deviation; use `alpha` to control the sensitivity with regards to outliers (see [F2001] for more details).
  - Quantile (`'quantile'`): A loss function for quantile regression. Use `0 < alpha < 1` to specify the quantile. This loss function can be used to create prediction intervals (see *Prediction Intervals for Gradient Boosting Regression*).
- Classification
  - Binomial deviance (`'deviance'`): The negative binomial log-likelihood loss function for binary classification (provides probability estimates). The initial model is given by the log odds-ratio.
  - Multinomial deviance (`'deviance'`): The negative multinomial log-likelihood loss function for multi-class classification with `n_classes` mutually exclusive classes. It provides probability estimates. The initial model is given by the prior probability of each class. At each iteration `n_classes` regression trees have to be constructed which makes GBRT rather inefficient for data sets with a large number of classes.

1.9.4.6. Regularization

1.9.4.6.1. Shrinkage

[F2001] proposed a simple regularization strategy that scales the contribution of each weak learner by a factor ν:

$$F_m(x) = F_{m-1}(x) + \nu\gamma_m h_m(x)$$

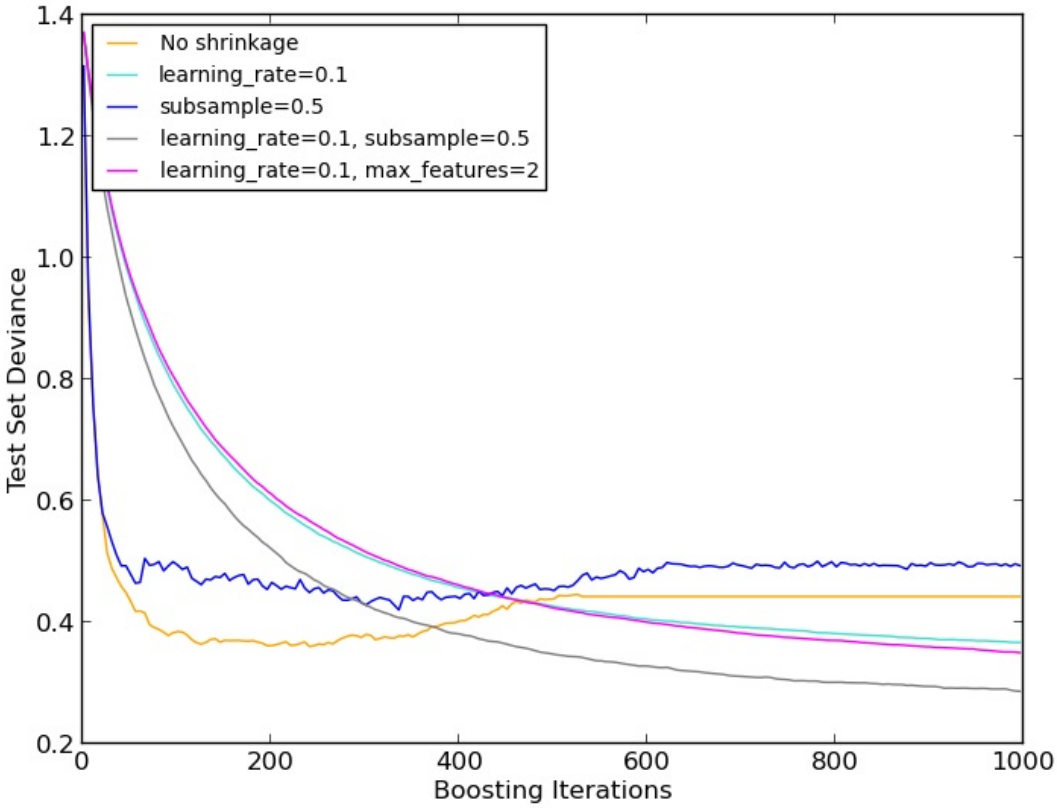
The parameter ν is also called the **learning rate** because it scales the step length the the gradient descent procedure; it can be set via the `learning_rate` parameter.

The parameter `learning_rate` strongly interacts with the parameter `n_estimators`, the number of weak learners to fit. Smaller values of `learning_rate` require larger numbers of weak learners to maintain a constant training error. Empirical evidence suggests that small values of `learning_rate` favor better test error. [HTF2009] recommend to set the learning rate to a small constant (e.g. `learning_rate <= 0.1`) and choose `n_estimators` by early stopping. For a more detailed discussion of the interaction between `learning_rate` and `n_estimators` see [R2007].

1.9.4.6.2. Subsampling

[F1999] proposed stochastic gradient boosting, which combines gradient boosting with bootstrap averaging (bagging). At each iteration the base classifier is trained on a fraction `subsample` of the available training data. The subsample is drawn without replacement. A typical value of `subsample` is 0.5.

The figure below illustrates the effect of shrinkage and subsampling on the goodness-of-fit of the model. We can clearly see that shrinkage outperforms no-shrinkage. Subsampling with shrinkage can further increase the accuracy of the model. Subsampling without shrinkage, on the other hand, does poorly.



Another strategy to reduce the variance is by subsampling the features analogous to the random splits in **`RandomForestClassifier`** . The number of subsampled features can be controlled via the `max_features` parameter.

Note: Using a small `max_features` value can significantly decrease the runtime.

Stochastic gradient boosting allows to compute out-of-bag estimates of the test deviance by computing the improvement in deviance on the examples that are not included in the bootstrap sample (i.e. the out-of-bag examples). The improvements are stored in the attribute **`oob_improvement_`**. `oob_improvement_[i]` holds the improvement in terms of the loss on the OOB samples if you add the i-th stage to the current predictions. Out-of-bag estimates can be used for model selection, for example to determine the optimal number of iterations. OOB estimates are usually very pessimistic thus we recommend to use cross-validation instead and only use OOB if cross-validation is too time consuming.

Examples:

- *Gradient Boosting regularization*
- *Gradient Boosting Out-of-Bag estimates*

1.9.4.7. Interpretation

Individual decision trees can be interpreted easily by simply visualizing the tree structure. Gradient boosting models, however, comprise hundreds of regression trees thus they cannot be easily interpreted by visual

inspection of the individual trees. Fortunately, a number of techniques have been proposed to summarize and interpret gradient boosting models.

1.9.4.7.1. Feature importance

Often features do not contribute equally to predict the target response; in many situations the majority of the features are in fact irrelevant. When interpreting a model, the first question usually is: what are those important features and how do they contributing in predicting the target response?

Individual decision trees intrinsically perform feature selection by selecting appropriate split points. This information can be used to measure the importance of each feature; the basic idea is: the more often a feature is used in the split points of a tree the more important that feature is. This notion of importance can be extended to decision tree ensembles by simply averaging the feature importance of each tree (see *Feature importance evaluation* for more details).

The feature importance scores of a fit gradient boosting model can be accessed via the `feature_importances_` property:

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X, y)
>>> clf.feature_importances_
array([ 0.11,  0.1 ,  0.11,  ...
```

Examples:

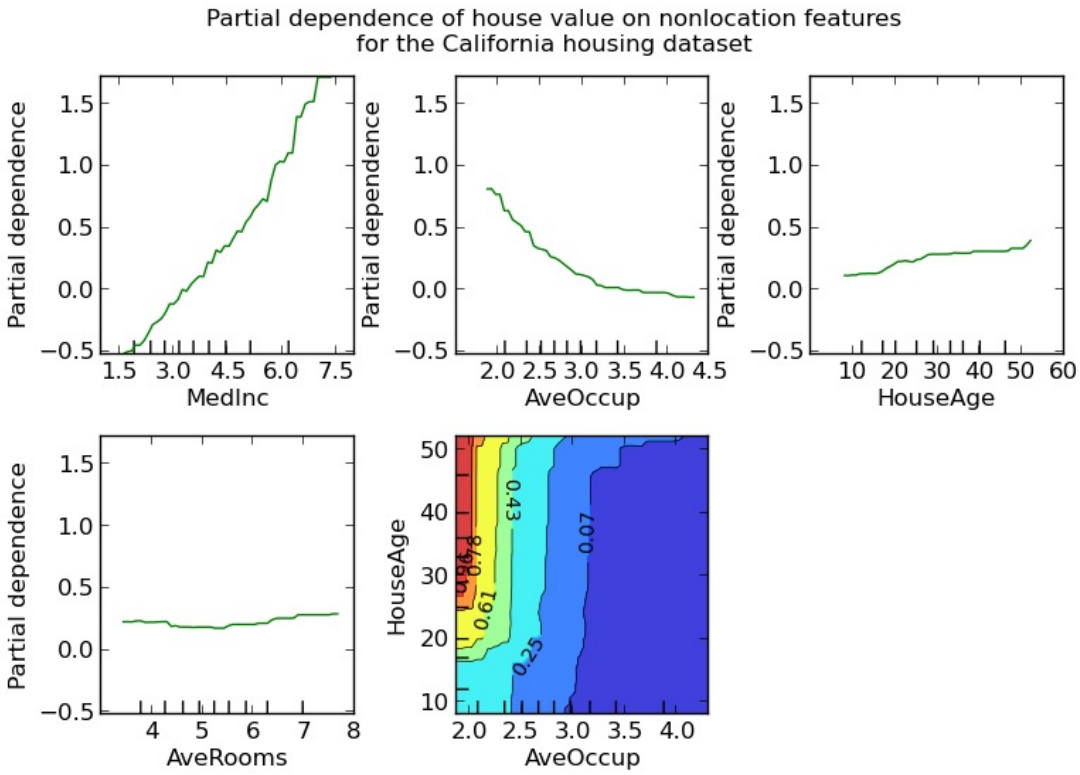
- *Gradient Boosting regression*

1.9.4.7.2. Partial dependence

Partial dependence plots (PDP) show the dependence between the target response and a set of ‘target’ features, marginalizing over the values of all other features (the ‘complement’ features). Intuitively, we can interpret the partial dependence as the expected target response [1] as a function of the ‘target’ features [2].

Due to the limits of human perception the size of the target feature set must be small (usually, one or two) thus the target features are usually chosen among the most important features.

The Figure below shows four one-way and one two-way partial dependence plots for the California housing dataset:



One-way PDPs tell us about the interaction between the target response and the target feature (e.g. linear, non-linear). The upper left plot in the above Figure shows the effect of the median income in a district on the median house price; we can clearly see a linear relationship among them.

PDPs with two target features show the interactions among the two features. For example, the two-variable PDP in the above Figure shows the dependence of median house price on joint values of house age and avg. occupants per household. We can clearly see an interaction between the two features: For an avg. occupancy greater than two, the house price is nearly independent of the house age, whereas for values less than two there is a strong dependence on age.

The module **`partial_dependence`** provides a convenience function **`plot_partial_dependence`** to create one-way and two-way partial dependence plots. In the below example we show how to create a grid of partial dependence plots: two one-way PDPs for the features `0` and `1` and a two-way PDP between the two features:

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> from sklearn.ensemble.partial_dependence import plot_partial_dependence

>>> X, y = make_hastie_10_2(random_state=0)
>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X, y)
>>> features = [0, 1, (0, 1)]
>>> fig, axs = plot_partial_dependence(clf, X, features)
```

For multi-class models, you need to set the class label for which the PDPs should be created via the `label` argument:

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> mc_clf = GradientBoostingClassifier(n_estimators=10,
...     max_depth=1).fit(iris.data, iris.target)
>>> features = [3, 2, (3, 2)]
>>> fig, axs = plot_partial_dependence(mc_clf, X, features, label=0)
```

If you need the raw values of the partial dependence function rather than the plots you can use the `partial_dependence` function:

```
>>> from sklearn.ensemble.partial_dependence import partial_dependence

>>> pdp, axes = partial_dependence(clf, [0], X=X)
>>> pdp
array([[ 2.46643157,  2.46643157, ...
>>> axes
[array([-1.62497054, -1.59201391, ...
```

The function requires either the argument `grid` which specifies the values of the target features on which the partial dependence function should be evaluated or the argument `X` which is a convenience mode for automatically creating `grid` from the training data. If `X` is given, the `axes` value returned by the function gives the axis for each target feature.

For each value of the 'target' features in the `grid` the partial dependence function need to marginalize the predictions of a tree over all possible values of the 'complement' features. In decision trees this function can be evaluated efficiently without reference to the training data. For each grid point a weighted tree traversal is performed: if a split node involves a 'target' feature, the corresponding left or right branch is followed, otherwise both branches are followed, each branch is weighted by the fraction of training samples that entered that branch. Finally, the partial dependence is given by a weighted average of all visited leaves. For tree ensembles the results of each individual tree are again averaged.

Footnotes

[1] For classification with `loss='deviance'` the target response is logit(p).

[2] More precisely its the expectation of the target response after accounting for the initial model; partial dependence plots do not include the `init` model.

Examples:

- *Partial Dependence Plots*

References

[F2001] *(1, 2, 3)* J. Friedman, "Greedy Function Approximation: A Gradient Boosting Machine", The Annals of Statistics, Vol. 29, No. 5, 2001.

[F1999] J. Friedman, "Stochastic Gradient Boosting", 1999

[HTF2009] T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

[R2007] G. Ridgeway, "Generalized Boosted Models: A guide to the gbm package", 2007

1.10. Multiclass and multilabel algorithms

Warning: All classifiers in scikit-learn do multiclass classification out-of-the-box. You don't need to use the `sklearn.multiclass` module unless you want to experiment with different multiclass strategies.

The `sklearn.multiclass` module implements *meta-estimators* to solve `multiclass` and `multilabel` classification problems by decomposing such problems into binary classification problems.

- **Multiclass classification** means a classification task with more than two classes; e.g., classify a set of images of fruits which may be oranges, apples, or pears. Multiclass classification makes the assumption that each sample is assigned to one and only one label: a fruit can be either an apple or a pear but not both at the same time.
- **Multilabel classification** assigns to each sample a set of target labels. This can be thought as predicting properties of a data-point that are not mutually exclusive, such as topics that are relevant for a document. A text might be about any of religion, politics, finance or education at the same time or none of these.
- **Multioutput-multiclass classification** and **multi-task classification** means that a single estimator has to handle several joint classification tasks. This is a generalization of the multi-label classification task, where the set of classification problem is restricted to binary classification, and of the multi-class classification task. *The output format is a 2d numpy array.*

  The set of labels can be different for each output variable. For instance a sample could be assigned "pear" for an output variable that takes possible values in a finite set of species such as "pear", "apple", "orange" and "green" for a second output variable that takes possible values in a finite set of colors such as "green", "red", "orange", "yellow"...

  This means that any classifiers handling multi-output multiclass or multi-task classification task supports the multi-label classification task as a special case. Multi-task classification is similar to the multi-output classification task with different model formulations. For more information, see the relevant estimator documentation.

All scikit-learn classifiers are capable of multiclass classification, but the meta-estimators offered by `sklearn.multiclass` permit changing the way they handle more than two classes because this may have an effect on classifier performance (either in terms of generalization error or required computational resources).

Below is a summary of the classifiers supported by scikit-learn grouped by strategy; you don't need the meta-estimators in this class if you're using one of these unless you want custom multiclass behavior:

- Inherently multiclass: *Naive Bayes*, `sklearn.lda.LDA`, *Decision Trees*, *Random Forests*, *Nearest Neighbors*.
- One-Vs-One: `sklearn.svm.SVC`.
- One-Vs-All: all linear models except `sklearn.svm.SVC`.

Some estimators also support multioutput-multiclass classification tasks *Decision Trees*, *Random Forests*, *Nearest Neighbors*.

Warning: At present, no metric in `sklearn.metrics` supports the multioutput-multiclass classification task.

1.10.1. Multilabel classification format

In multilabel learning, the joint set of binary classification tasks is expressed with label binary indicator array: each sample is one row of a 2d array of shape (n_samples, n_classes) with binary values: the one, i.e. the non zero elements, corresponds to the subset of labels. An array such as `np.array([[1, 0, 0], [0, 1, 1], [0, 0, 0]])` represents label 0 in the first sample, labels 1 and 2 in the second sample, and no labels in the third sample.

Producing multilabel data as a list of sets of labels may be more intuitive. The transformer `MultiLabelBinarizer` will convert between a collection of collections of labels and the indicator format.

```
>>> from sklearn.datasets import make_multilabel_classification
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> X, Y = make_multilabel_classification(n_samples=5, random_state=0,
...                                       return_indicator=False)
>>> Y
([0, 1, 2], [4, 1, 0, 2], [4, 0, 1], [1, 0], [3, 2])
>>> MultiLabelBinarizer().fit_transform(Y)
array([[1, 1, 1, 0, 0],
       [1, 1, 1, 0, 1],
       [1, 1, 0, 0, 1],
       [1, 1, 0, 0, 0],
       [0, 0, 1, 1, 0]])
```

1.10.2. One-Vs-The-Rest

This strategy, also known as **one-vs-all**, is implemented in `OneVsRestClassifier`. The strategy consists in fitting one classifier per class. For each classifier, the class is fitted against all the other classes. In addition to its computational efficiency (only *n_classes* classifiers are needed), one advantage of this approach is its interpretability. Since each class is represented by one and one classifier only, it is possible to gain knowledge about the class by inspecting its corresponding classifier. This is the most commonly used strategy and is a fair default choice.

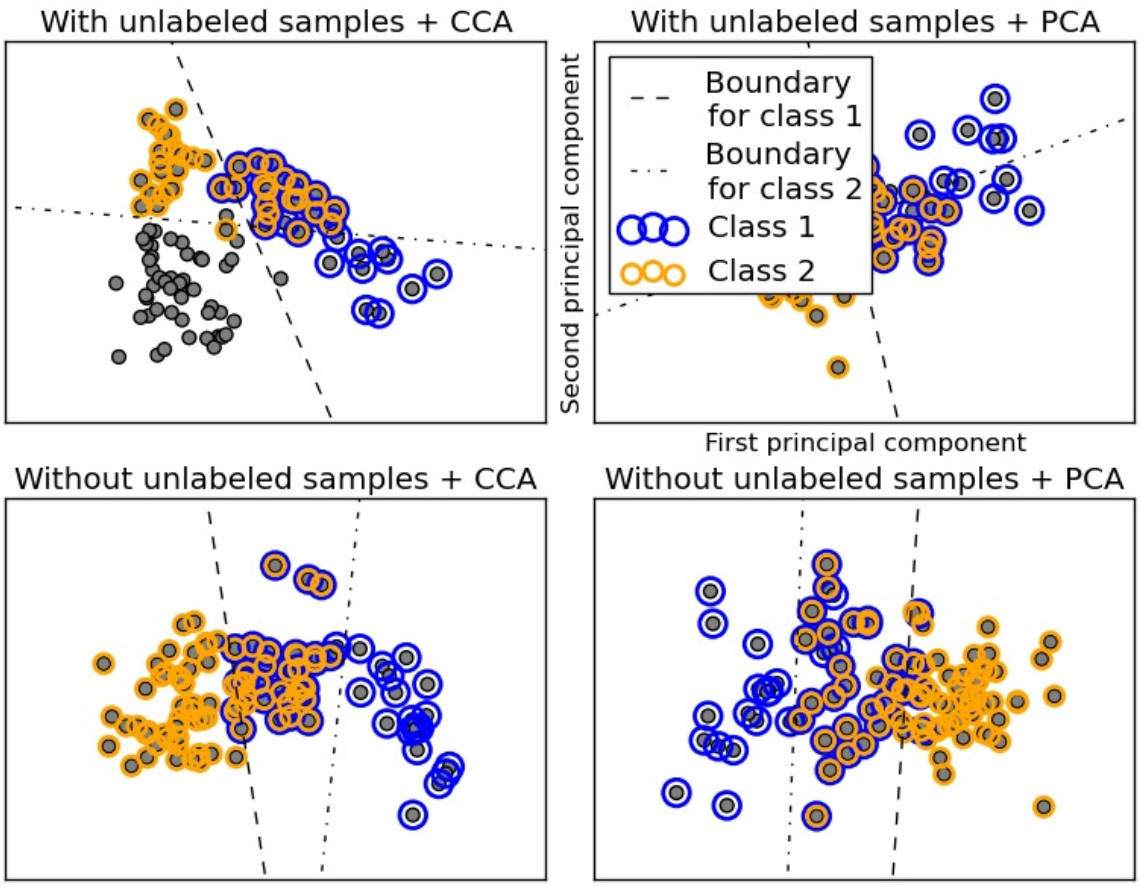
1.10.2.1. Multiclass learning

Below is an example of multiclass learning using OvR:

```
>>> from sklearn import datasets
>>> from sklearn.multiclass import OneVsRestClassifier
>>> from sklearn.svm import LinearSVC
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> OneVsRestClassifier(LinearSVC(random_state=0)).fit(X, y).predict(X)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

1.10.2.2. Multilabel learning

`OneVsRestClassifier` also supports multilabel classification. To use this feature, feed the classifier an indicator matrix, in which cell [i, j] indicates the presence of label j in sample i.



Examples:

- *Multilabel classification*

1.10.3. One-Vs-One

`OneVsOneClassifier` constructs one classifier per pair of classes. At prediction time, the class which received the most votes is selected. Since it requires to fit `n_classes * (n_classes - 1) / 2` classifiers, this method is usually slower than one-vs-the-rest, due to its O(n_classes^2) complexity. However, this method may be advantageous for algorithms such as kernel algorithms which don't scale well with `n_samples`. This is because each individual learning problem only involves a small subset of the data whereas, with one-vs-the-rest, the complete dataset is used `n_classes` times.

1.10.3.1. Multiclass learning

Below is an example of multiclass learning using OvO:

```
>>> from sklearn import datasets
>>> from sklearn.multiclass import OneVsOneClassifier
>>> from sklearn.svm import LinearSVC
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> OneVsOneClassifier(LinearSVC(random_state=0)).fit(X, y).predict(X)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

1.10.4. Error-Correcting Output-Codes

Output-code based strategies are fairly different from one-vs-the-rest and one-vs-one. With these strategies, each class is represented in a euclidean space, where each dimension can only be 0 or 1. Another way to put it is that each class is represented by a binary code (an array of 0 and 1). The matrix which keeps track of the location/code of each class is called the code book. The code size is the dimensionality of the aforementioned space. Intuitively, each class should be represented by a code as unique as possible and a good code book should be designed to optimize classification accuracy. In this implementation, we simply use a randomly-generated code book as advocated in [2] although more elaborate methods may be added in the future.

At fitting time, one binary classifier per bit in the code book is fitted. At prediction time, the classifiers are used to project new points in the class space and the class closest to the points is chosen.

In `OutputCodeClassifier`, the `code_size` attribute allows the user to control the number of classifiers which will be used. It is a percentage of the total number of classes.

A number between 0 and 1 will require fewer classifiers than one-vs-the-rest. In theory, `log2(n_classes) / n_classes` is sufficient to represent each class unambiguously. However, in practice, it may not lead to good accuracy since `log2(n_classes)` is much smaller than n_classes.

A number greater than than 1 will require more classifiers than one-vs-the-rest. In this case, some classifiers will in theory correct for the mistakes made by other classifiers, hence the name “error-correcting”. In practice, however, this may not happen as classifier mistakes will typically be correlated. The error-correcting output codes have a similar effect to bagging.

1.10.4.1. Multiclass learning

Below is an example of multiclass learning using Output-Codes:

```
>>> from sklearn import datasets
>>> from sklearn.multiclass import OutputCodeClassifier
>>> from sklearn.svm import LinearSVC
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf = OutputCodeClassifier(LinearSVC(random_state=0),
...                            code_size=2, random_state=0)
>>> clf.fit(X, y).predict(X)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 1, 1, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

References:

[1] “Solving multiclass learning problems via error-correcting output codes”, Dietterich T., Bakiri G., Journal of Artificial Intelligence Research 2, 1995.

[2] “The error coding method and PICTs”, James G., Hastie T., Journal of Computational and Graphical statistics 7, 1998.
[3] “The Elements of Statistical Learning”, Hastie T., Tibshirani R., Friedman J., page 606 (second-edition) 2008.

Previous Next

1.11. Feature selection

The classes in the `sklearn.feature_selection` module can be used for feature selection/dimensionality reduction on sample sets, either to improve estimators' accuracy scores or to boost their performance on very high-dimensional datasets.

1.11.1. Removing features with low variance

`VarianceThreshold` is a simple baseline approach to feature selection. It removes all features whose variance doesn't meet some threshold. By default, it removes all zero-variance features, i.e. features that have the same value in all samples.

As an example, suppose that we have a dataset with boolean features, and we want to remove all features that are either one or zero (on or off) in more than 80% of the samples. Boolean features are Bernoulli random variables, and the variance of such variables is given by

$$\mathrm{Var}[X] = p(1-p)$$

so we can select using the threshold `.8 * (1 - .8)`:

```
>>> from sklearn.feature_selection import VarianceThreshold
>>> X = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]
>>> sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
>>> sel.fit_transform(X)
array([[0, 1],
       [1, 0],
       [0, 0],
       [1, 1],
       [1, 0],
       [1, 1]])
```

As expected, `VarianceThreshold` has removed the first column, which has a probability $p = 5/6 > .8$ of containing a one.

1.11.2. Univariate feature selection

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

- `SelectKBest` removes all but the k highest scoring features
- `SelectPercentile` removes all but a user-specified highest scoring percentage of features
- using common univariate statistical tests for each feature: false positive rate `SelectFpr`, false discovery rate `SelectFdr`, or family wise error `SelectFwe`.
- `GenericUnivariateSelect` allows to perform univariate feature

selection with a configurable strategy. This allows to select the best univariate selection strategy with hyper-parameter search estimator.

For instance, we can perform a χ^2 test to the samples to retrieve only the two best features as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import chi2
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> X_new = SelectKBest(chi2, k=2).fit_transform(X, y)
>>> X_new.shape
(150, 2)
```

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

- For regression: `f_regression`
- For classification: `chi2` or `f_classif`

Feature selection with sparse data

If you use sparse data (i.e. data represented as sparse matrices), only `chi2` will deal with the data without making it dense.

Warning: Beware not to use a regression scoring function with a classification problem, you will get useless results.

Examples:

Univariate Feature Selection

1.11.3. Recursive feature elimination

Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), recursive feature elimination (`RFE`) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and weights are assigned to each one of them. Then, features whose absolute weights are the smallest are pruned from the current set features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

`RFECV` performs RFE in a cross-validation loop to find the optimal number of features.

Examples:

- *Recursive feature elimination*: A recursive feature elimination example showing the relevance of pixels in a digit classification task.
- *Recursive feature elimination with cross-validation*: A recursive feature elimination example with automatic tuning of the number of features selected with cross-validation.

1.11.4. L1-based feature selection

1.11.4.1. Selecting non-zero coefficients

Linear models penalized with the L1 norm have sparse solutions: many of their estimated coefficients are zero. When the goal is to reduce the dimensionality of the data to use with another classifier, they expose a `transform` method to select the non-zero coefficient. In particular, sparse estimators useful for this purpose are the **`linear_model.Lasso`** for regression, and of **`linear_model.LogisticRegression`** and **`svm.LinearSVC`** for classification:

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> X_new = LinearSVC(C=0.01, penalty="l1", dual=False).fit_transform(X, y)
>>> X_new.shape
(150, 3)
```

With SVMs and logistic-regression, the parameter C controls the sparsity: the smaller C the fewer features selected. With Lasso, the higher the alpha parameter, the fewer features selected.

Examples:

- *Classification of text documents using sparse features*: Comparison of different algorithms for document classification including L1-based feature selection.

L1-recovery and compressive sensing

For a good choice of alpha, the *Lasso* can fully recover the exact set of non-zero variables using only few observations, provided certain specific conditions are met. In particular, the number of samples should be “sufficiently large”, or L1 models will perform at random, where “sufficiently large” depends on the number of non-zero coefficients, the logarithm of the number of features, the amount of noise, the smallest absolute value of non-zero coefficients, and the structure of the design matrix X. In addition, the design matrix must display certain specific properties, such as not being too correlated.

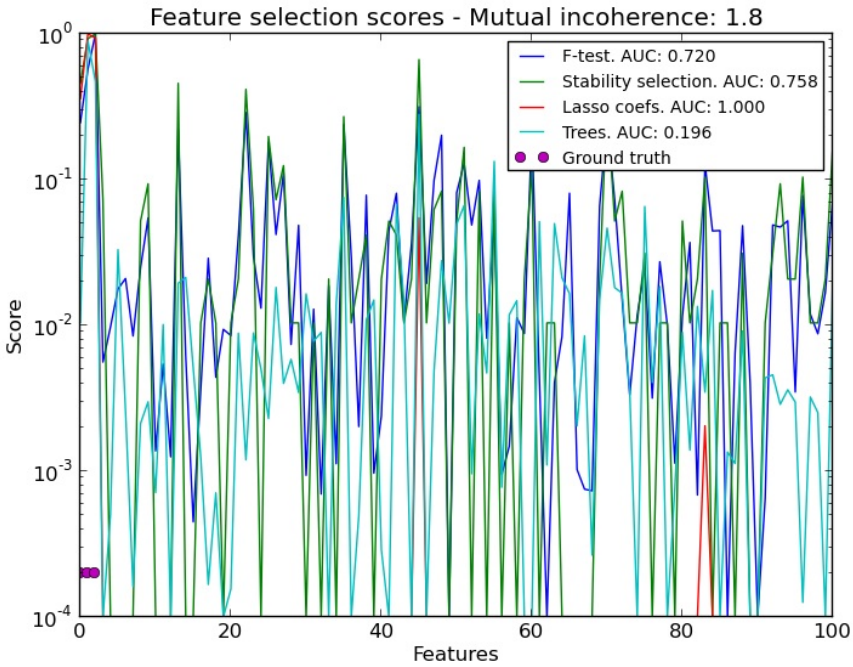
There is no general rule to select an alpha parameter for recovery of non-zero coefficients. It can by set by cross-validation (**`LassoCV`** or **`LassoLarsCV`**), though this may lead to under-penalized models: including a small number of non-relevant variables is not detrimental to prediction score. BIC (**`LassoLarsIC`**) tends, on the opposite, to set high values of alpha.

Reference Richard G. Baraniuk “Compressive Sensing”, IEEE Signal Processing Magazine [120] July 2007 http://dsp.rice.edu/files/cs/baraniukCSlecture07.pdf

1.11.4.2. Randomized sparse models

The limitation of L1-based sparse models is that faced with a group of very correlated features, they will select only one. To mitigate this problem, it is possible to use randomization techniques, reestimating the sparse model many times perturbing the design matrix or sub-sampling data and counting how many times a given regressor is selected.

`RandomizedLasso` implements this strategy for regression settings, using the Lasso, while `RandomizedLogisticRegression` uses the logistic regression and is suitable for classification tasks. To get a full path of stability scores you can use `lasso_stability_path`.



Note that for randomized sparse models to be more powerful than standard F statistics at detecting non-zero features, the ground truth model should be sparse, in other words, there should be only a small fraction of features non zero.

Examples:

- *Sparse recovery: feature selection for sparse linear models*: An example comparing different feature selection approaches and discussing in which situation each approach is to be favored.

References:

- N. Meinshausen, P. Buhlmann, “Stability selection”, Journal of the Royal Statistical Society, 72 (2010) http://arxiv.org/pdf/0809.2932
- F. Bach, “Model-Consistent Sparse Estimation through the Bootstrap” http://hal.inria.fr/hal-00354771/

1.11.5. Tree-based feature selection

Tree-based estimators (see the `sklearn.tree` module and forest of trees in the `sklearn.ensemble` module) can be used to compute feature importances, which in turn can be used to discard irrelevant features:

```
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> clf = ExtraTreesClassifier()
>>> X_new = clf.fit(X, y).transform(X)
```

```
>>> clf.feature_importances_
array([ 0.04...,  0.05...,  0.4...,  0.4...])
>>> X_new.shape
(150, 2)
```

Examples:

- *Feature importances with forests of trees*: example on synthetic data showing the recovery of the actually meaningful features.
- *Pixel importances with a parallel forest of trees*: example on face recognition data.

1.11.6. Feature selection as part of a pipeline

Feature selection is usually used as a pre-processing step before doing the actual learning. The recommended way to do this in scikit-learn is to use a **sklearn.pipeline.Pipeline**:

```
clf = Pipeline([
  ('feature_selection', LinearSVC(penalty="l1")),
  ('classification', RandomForestClassifier())
])
clf.fit(X, y)
```

In this snippet we make use of a **sklearn.svm.LinearSVC** to evaluate feature importances and select the most relevant features. Then, a **sklearn.ensemble.RandomForestClassifier** is trained on the transformed output, i.e. using only relevant features. You can perform similar operations with the other feature selection methods and also classifiers that provide a way to evaluate feature importances of course. See the **sklearn.pipeline.Pipeline** examples for more details.

1.12. Semi-Supervised

Semi-supervised learning is a situation in which in your training data some of the samples are not labeled. The semi-supervised estimators in `sklearn.semi_supervised` are able to make use of this additional unlabeled data to better capture the shape of the underlying data distribution and generalize better to new samples. These algorithms can perform well when we have a very small amount of labeled points and a large amount of unlabeled points.

Unlabeled entries in *y*

It is important to assign an identifier to unlabeled points along with the labeled data when training the model with the `fit` method. The identifier that this implementation uses is the integer value -1.

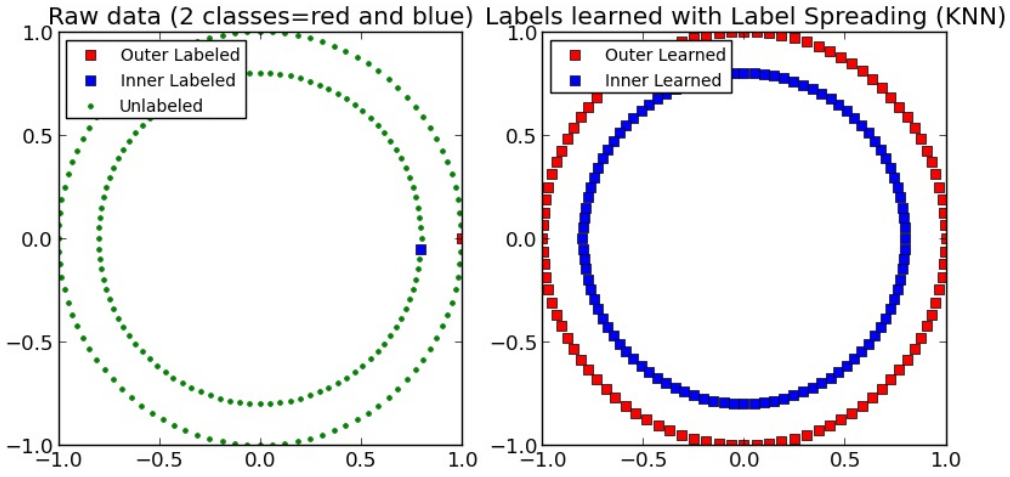
1.12.1. Label Propagation

Label propagation denotes a few variations of semi-supervised graph inference algorithms.

A few features available in this model:

- Can be used for classification and regression tasks
- Kernel methods to project data into alternate dimensional spaces

scikit-learn provides two label propagation models: `LabelPropagation` and `LabelSpreading`. Both work by constructing a similarity graph over all items in the input dataset.



An illustration of label-propagation: *the structure of unlabeled observations is consistent with the class structure, and thus the class label can be propagated to the unlabeled observations of the training set.*

`LabelPropagation` and `LabelSpreading` differ in modifications to the similarity matrix that graph and the clamping effect on the label distributions. Clamping allows the algorithm to change the weight of the true ground labeled data to some degree. The `LabelPropagation` algorithm performs hard clamping of input labels, which means $\alpha = 1$. This clamping factor can be relaxed, to say $\alpha = 0.8$, which means that we will always retain 80 percent of our original label distribution, but the algorithm gets to change it's confidence of

the distribution within 20 percent.

`LabelPropagation` uses the raw similarity matrix constructed from the data with no modifications. In contrast, `LabelSpreading` minimizes a loss function that has regularization properties, as such it is often more robust to noise. The algorithm iterates on a modified version of the original graph and normalizes the edge weights by computing the normalized graph Laplacian matrix. This procedure is also used in *Spectral clustering*.

Label propagation models have two built-in kernel methods. Choice of kernel effects both scalability and performance of the algorithms. The following are available:

- rbf ($\exp(-\gamma|x-y|^2), \gamma > 0$). γ is specified by keyword gamma.
- knn ($1[x' \in kNN(x)]$). k is specified by keyword n_neighbors.

The RBF kernel will produce a fully connected graph which is represented in memory by a dense matrix. This matrix may be very large and combined with the cost of performing a full matrix multiplication calculation for each iteration of the algorithm can lead to prohibitively long running times. On the other hand, the KNN kernel will produce a much more memory-friendly sparse matrix which can drastically reduce running times.

Examples

- *Decision boundary of label propagation versus SVM on the Iris dataset*
- *Label Propagation learning a complex structure*
- *Label Propagation digits active learning*

References

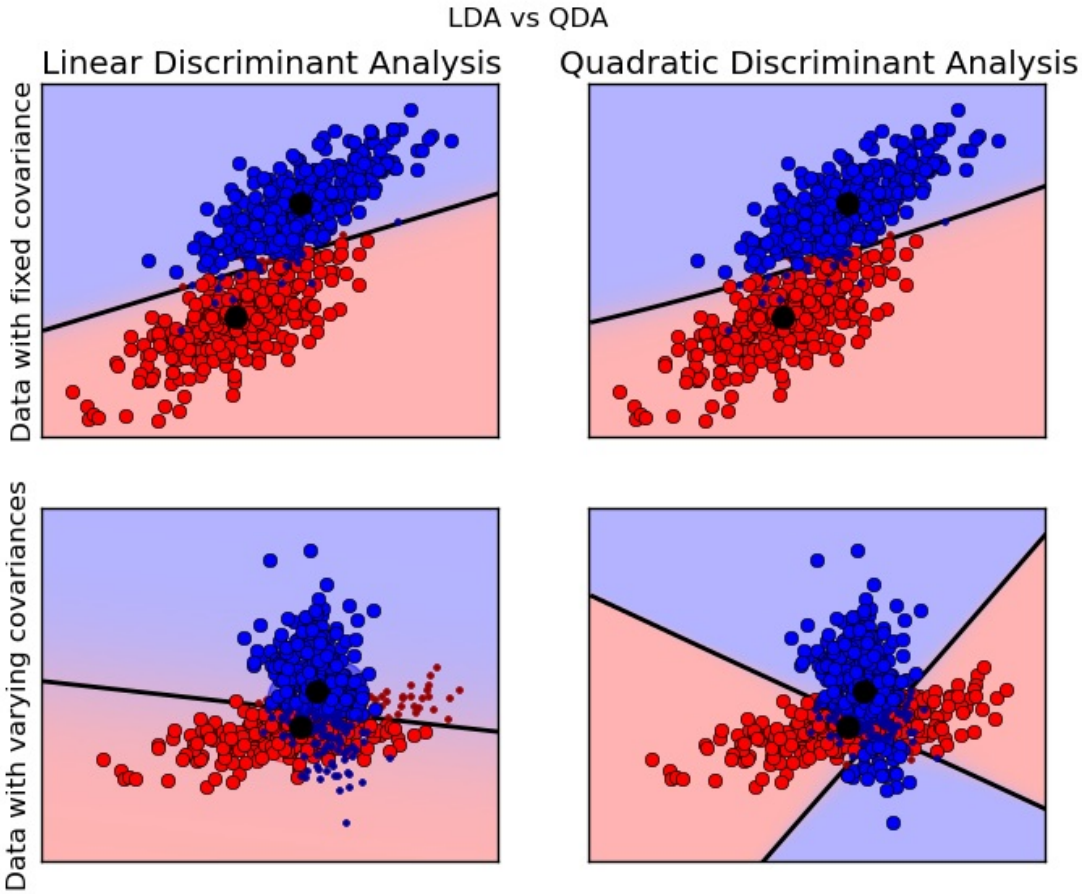
[1] Yoshua Bengio, Olivier Delalleau, Nicolas Le Roux. In Semi-Supervised Learning (2006), pp. 193-216

[2] Olivier Delalleau, Yoshua Bengio, Nicolas Le Roux. Efficient Non-Parametric Function Induction in Semi-Supervised Learning. AISTAT 2005 http://research.microsoft.com/en-us/people/nicolasl/efficient_ssl.pdf

1.13. Linear and quadratic discriminant analysis

Linear discriminant analysis (`lda.LDA`) and quadratic discriminant analysis (`qda.QDA`) are two classic classifiers, with, as their names suggest, a linear and a quadratic decision surface, respectively.

These classifiers are attractive because they have closed-form solutions that can be easily computed, are inherently multiclass, and have proven to work well in practice. Also there are no parameters to tune for these algorithms.



The plot shows decision boundaries for LDA and QDA. The bottom row demonstrates that LDA can only learn linear boundaries, while QDA can learn quadratic boundaries and is therefore more flexible.

Examples:

Linear and Quadratic Discriminant Analysis with confidence ellipsoid: Comparison of LDA and QDA on synthetic data.

References:

[3] "The Elements of Statistical Learning", Hastie T., Tibshirani R., Friedman J., 2008.

1.13.1. Dimensionality reduction using LDA

`lda.LDA` can be used to perform supervised dimensionality reduction by projecting the input data to a subspace consisting of the most discriminant directions. This is implemented in `lda.LDA.transform`. The desired dimensionality can be set using the `n_components` constructor parameter. This parameter has no influence on `lda.LDA.fit` or `lda.LDA.predict`.

1.13.2. Mathematical Idea

Both methods work by modeling the class conditional distribution of the data $P(X|y = k)$ for each class k. Predictions can be obtained by using Bayes' rule:

$$P(y|X) = P(X|y) \cdot P(y)/P(X) = P(X|y) \cdot P(Y)/(\sum_{y'} P(X|y') \cdot p(y'))$$

In linear and quadratic discriminant analysis, $P(X|y)$ is modelled as a Gaussian distribution. In the case of LDA, the Gaussians for each class are assumed to share the same covariance matrix. This leads to a linear decision surface, as can be seen by comparing the the log-probability rations $log[P(y = k|X)/P(y = l|X)]$.

In the case of QDA, there are no assumptions on the covariance matrices of the Gaussians, leading to a quadratic decision surface.

Previous Next

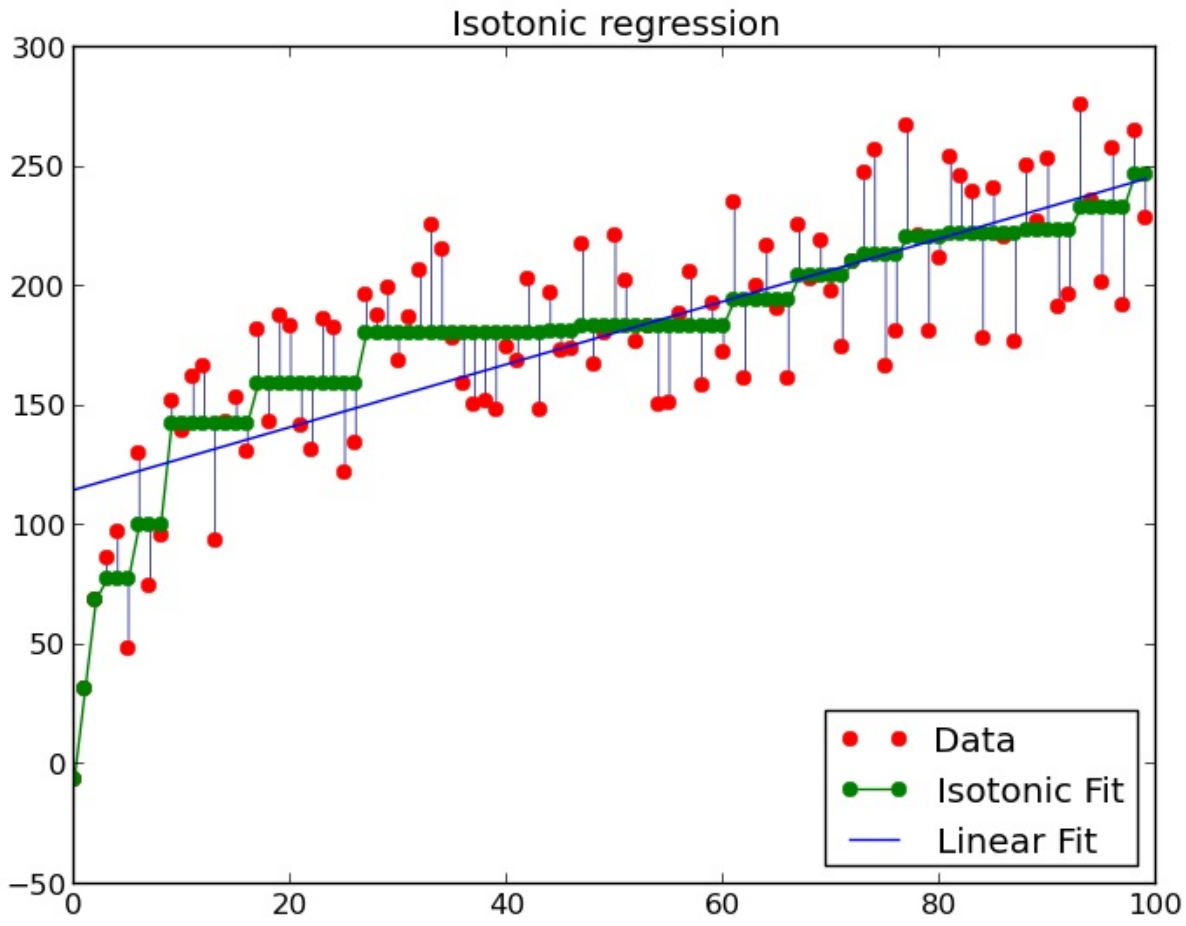
1.14. Isotonic regression

The class `IsotonicRegression` fits a non-decreasing function to data. It solves the following problem:

minimize $\sum_i w_i(y_i - \hat{y}_i)^2$

subject to $\hat{y}_{min} = \hat{y}_1 \leq \hat{y}_2 \ldots \leq \hat{y}_n = \hat{y}_{max}$

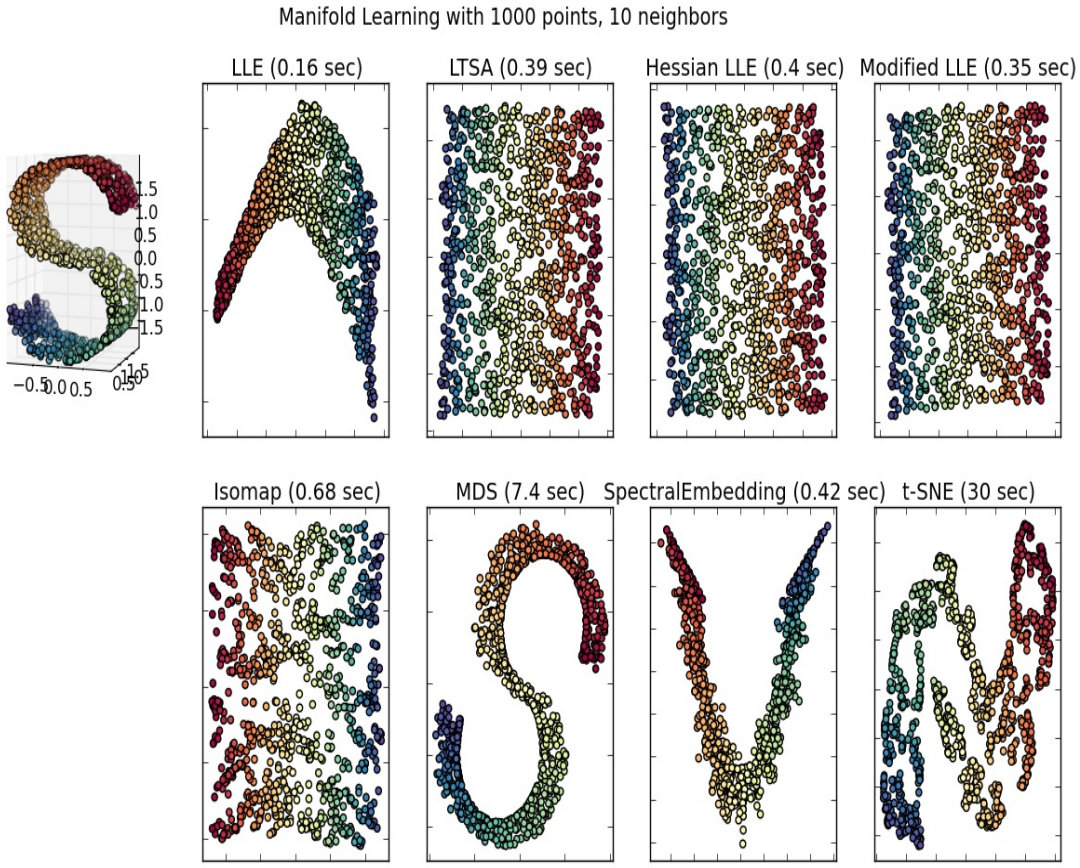
where each w_i is strictly positive and each y_i is an arbitrary real number. It yields the vector which is composed of non-decreasing elements the closest in terms of mean squared error. In practice this list of elements forms a function that is piecewise linear.



2.2. Manifold learning

Look for the bare necessities
The simple bare necessities
Forget about your worries and your strife
I mean the bare necessities
Old Mother Nature's recipes
That bring the bare necessities of life

– Baloo's song [The Jungle Book]

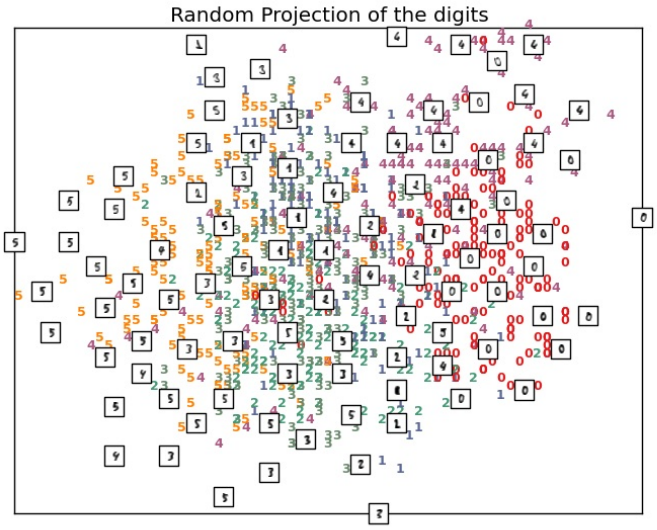


Manifold learning is an approach to non-linear dimensionality reduction. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high.

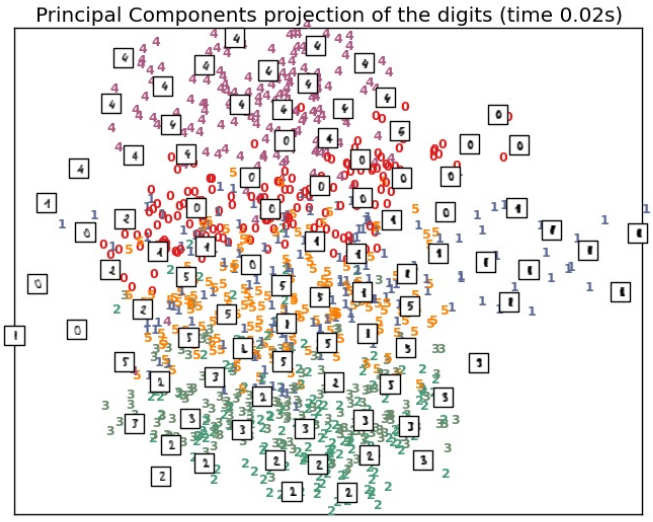
2.2.1. Introduction

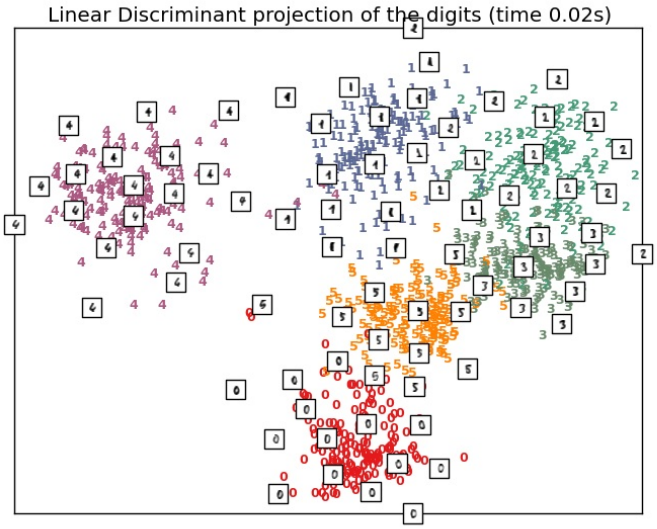
High-dimensional datasets can be very difficult to visualize. While data in two or three dimensions can be plotted to show the inherent structure of the data, equivalent high-dimensional plots are much less intuitive. To aid visualization of the structure of a dataset, the dimension must be reduced in some way.

The simplest way to accomplish this dimensionality reduction is by taking a random projection of the data. Though this allows some degree of visualization of the data structure, the randomness of the choice leaves much to be desired. In a random projection, it is likely that the more interesting structure within the data will be lost.



To address this concern, a number of supervised and unsupervised linear dimensionality reduction frameworks have been designed, such as Principal Component Analysis (PCA), Independent Component Analysis, Linear Discriminant Analysis, and others. These algorithms define specific rubrics to choose an “interesting” linear projection of the data. These methods can be powerful, but often miss important non-linear structure in the data.





Manifold Learning can be thought of as an attempt to generalize linear frameworks like PCA to be sensitive to non-linear structure in data. Though supervised variants exist, the typical manifold learning problem is unsupervised: it learns the high-dimensional structure of the data from the data itself, without the use of predetermined classifications.

Examples:

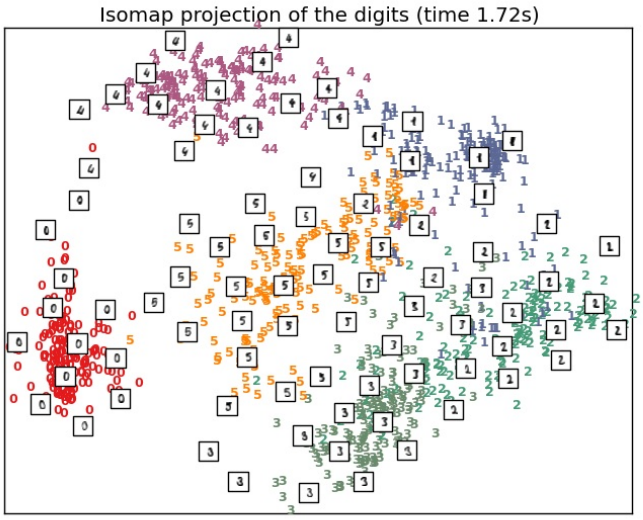
- See *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...* for an example of dimensionality reduction on handwritten digits.
- See *Comparison of Manifold Learning methods* for an example of dimensionality reduction on a toy "S-curve" dataset.

The manifold learning implementations available in sklearn are summarized below

2.2.2. Isomap

One of the earliest approaches to manifold learning is the Isomap algorithm, short for Isometric Mapping. Isomap can be viewed as an extension of Multi-dimensional Scaling (MDS) or Kernel PCA. Isomap seeks a lower-dimensional embedding which maintains geodesic distances between all points. Isomap can be

performed with the object `Isomap`.



2.2.2.1. Complexity

The Isomap algorithm comprises three stages:

1. **Nearest neighbor search.** Isomap uses `sklearn.neighbors.BallTree` for efficient neighbor search. The cost is approximately $O[D \log(k) N \log(N)]$, for k nearest neighbors of N points in D dimensions.
2. **Shortest-path graph search.** The most efficient known algorithms for this are *Dijkstra's Algorithm*, which is approximately $O[N^2(k + \log(N))]$, or the *Floyd-Warshall algorithm*, which is $O[N^3]$. The algorithm can be selected by the user with the `path_method` keyword of `Isomap`. If unspecified, the code attempts to choose the best algorithm for the input data.
3. **Partial eigenvalue decomposition.** The embedding is encoded in the eigenvectors corresponding to the d largest eigenvalues of the $N \times N$ isomap kernel. For a dense solver, the cost is approximately $O[dN^2]$. This cost can often be improved using the `ARPACK` solver. The eigensolver can be specified by the user with the `path_method` keyword of `Isomap`. If unspecified, the code attempts to choose the best algorithm for the input data.

The overall complexity of Isomap is $O[D \log(k) N \log(N)] + O[N^2(k + \log(N))] + O[dN^2]$.

- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

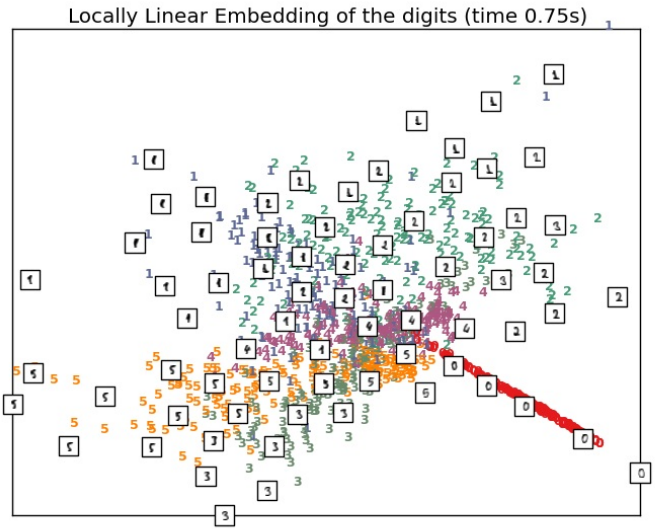
- "A global geometric framework for nonlinear dimensionality reduction" Tenenbaum, J.B.; De Silva, V.; & Langford, J.C. Science 290 (5500)

2.2.3. Locally Linear Embedding

Locally linear embedding (LLE) seeks a lower-dimensional projection of the data which preserves distances

within local neighborhoods. It can be thought of as a series of local Principal Component Analyses which are globally compared to find the best non-linear embedding.

Locally linear embedding can be performed with function `locally_linear_embedding` or its object-oriented counterpart `LocallyLinearEmbedding`.



2.2.3.1. Complexity

The standard LLE algorithm comprises three stages:

1. **Nearest Neighbors Search**. See discussion under Isomap above.
2. **Weight Matrix Construction**. $O[DNk^3]$. The construction of the LLE weight matrix involves the solution of a $k \times k$ linear equation for each of the N local neighborhoods
3. **Partial Eigenvalue Decomposition**. See discussion under Isomap above.

The overall complexity of standard LLE is $O[D \log(k) N \log(N)] + O[DNk^3] + O[dN^2]$.

- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

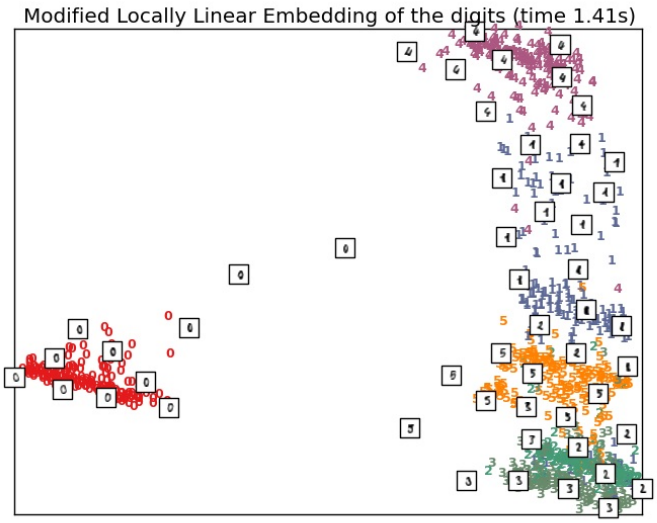
- "Nonlinear dimensionality reduction by locally linear embedding" Roweis, S. & Saul, L. Science 290:2323 (2000)

2.2.4. Modified Locally Linear Embedding

One well-known issue with LLE is the regularization problem. When the number of neighbors is greater than the number of input dimensions, the matrix defining each local neighborhood is rank-deficient. To address this, standard LLE applies an arbitrary regularization parameter r, which is chosen relative to the trace of the local weight matrix. Though it can be shown formally that as $r \to 0$, the solution converges to the desired embedding, there is no guarantee that the optimal solution will be found for $r > 0$. This problem manifests

itself in embeddings which distort the underlying geometry of the manifold.

One method to address the regularization problem is to use multiple weight vectors in each neighborhood. This is the essence of *modified locally linear embedding* (MLLE). MLLE can be performed with function `locally_linear_embedding` or its object-oriented counterpart `LocallyLinearEmbedding`, with the keyword `method = 'modified'`. It requires `n_neighbors > n_components`.



2.2.4.1. Complexity

The MLLE algorithm comprises three stages:

1. **Nearest Neighbors Search**. Same as standard LLE
2. **Weight Matrix Construction**. Approximately $O[DNk^3] + O[N(k-D)k^2]$. The first term is exactly equivalent to that of standard LLE. The second term has to do with constructing the weight matrix from multiple weights. In practice, the added cost of constructing the MLLE weight matrix is relatively small compared to the cost of steps 1 and 3.
3. **Partial Eigenvalue Decomposition**. Same as standard LLE

The overall complexity of MLLE is
$O[D\log(k)N\log(N)] + O[DNk^3] + O[N(k-D)k^2] + O[dN^2]$.

- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

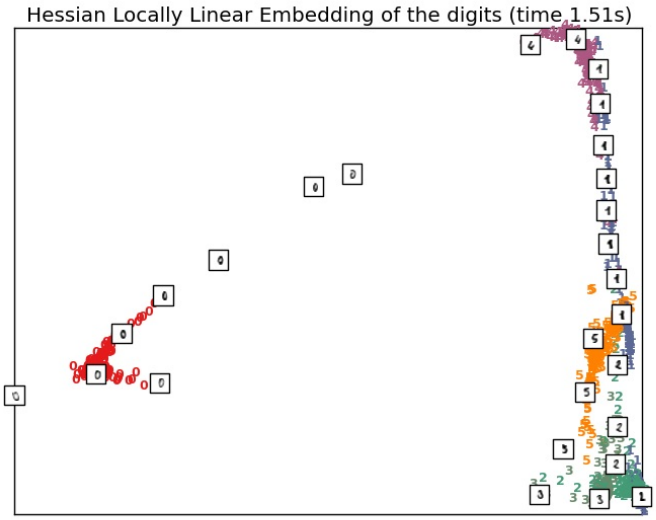
References:

- "MLLE: Modified Locally Linear Embedding Using Multiple Weights" Zhang, Z. & Wang, J.

2.2.5. Hessian Eigenmapping

Hessian Eigenmapping (also known as Hessian-based LLE: HLLE) is another method of solving the regularization problem of LLE. It revolves around a hessian-based quadratic form at each neighborhood

which is used to recover the locally linear structure. Though other implementations note its poor scaling with data size, `sklearn` implements some algorithmic improvements which make its cost comparable to that of other LLE variants for small output dimension. HLLE can be performed with function **`locally_linear_embedding`** or its object-oriented counterpart **`LocallyLinearEmbedding`**, with the keyword `method = 'hessian'`. It requires `n_neighbors > n_components * (n_components + 3) / 2`.



2.2.5.1. Complexity

The HLLE algorithm comprises three stages:

1. **Nearest Neighbors Search**. Same as standard LLE
2. **Weight Matrix Construction**. Approximately $O[DNk^3] + O[Nd^6]$. The first term reflects a similar cost to that of standard LLE. The second term comes from a QR decomposition of the local hessian estimator.
3. **Partial Eigenvalue Decomposition**. Same as standard LLE

The overall complexity of standard HLLE is $O[D \log(k) N \log(N)] + O[DNk^3] + O[Nd^6] + O[dN^2]$.

- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

- "Hessian Eigenmaps: Locally linear embedding techniques for high-dimensional data" Donoho, D. & Grimes, C. Proc Natl Acad Sci USA. 100:5591 (2003)

2.2.6. Spectral Embedding

Spectral Embedding (also known as Laplacian Eigenmaps) is one method to calculate non-linear embedding. It finds a low dimensional representation of the data using a spectral decomposition of the graph Laplacian. The graph generated can be considered as a discrete approximation of the low dimensional manifold in the

high dimensional space. Minimization of a cost function based on the graph ensures that points close to each other on the manifold are mapped close to each other in the low dimensional space, preserving local distances. Spectral embedding can be performed with the function `spectral_embedding` or its object-oriented counterpart `SpectralEmbedding`.

2.2.6.1. Complexity

The Spectral Embedding algorithm comprises three stages:

1. **Weighted Graph Construction**. Transform the raw input data into graph representation using affinity (adjacency) matrix representation.
2. **Graph Laplacian Construction**. unnormalized Graph Laplacian is constructed as $L = D - A$ for and normalized one as $L = D^{-\frac{1}{2}}(D - A)D^{-\frac{1}{2}}$.
3. **Partial Eigenvalue Decomposition**. Eigenvalue decomposition is done on graph Laplacian

The overall complexity of spectral embedding is $O[D \log(k) N \log(N)] + O[DNk^3] + O[dN^2]$.

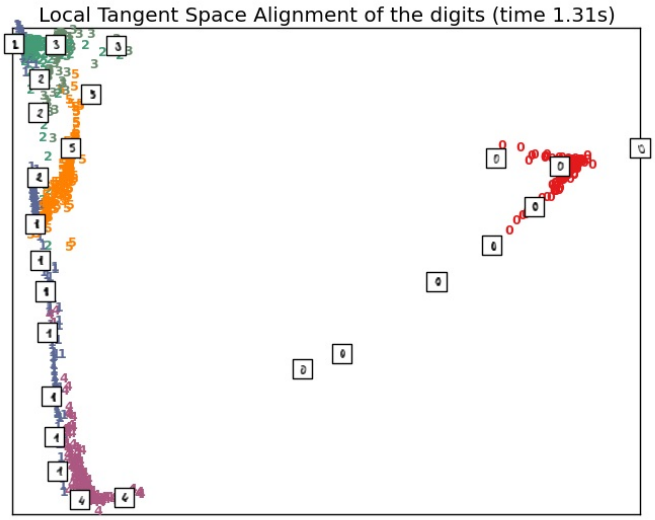
- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

- "Laplacian Eigenmaps for Dimensionality Reduction and Data Representation" M. Belkin, P. Niyogi, Neural Computation, June 2003; 15 (6):1373-1396

2.2.7. Local Tangent Space Alignment

Though not technically a variant of LLE, Local tangent space alignment (LTSA) is algorithmically similar enough to LLE that it can be put in this category. Rather than focusing on preserving neighborhood distances as in LLE, LTSA seeks to characterize the local geometry at each neighborhood via its tangent space, and performs a global optimization to align these local tangent spaces to learn the embedding. LTSA can be performed with function `locally_linear_embedding` or its object-oriented counterpart `LocallyLinearEmbedding`, with the keyword `method = 'ltsa'`.



2.2.7.1. Complexity

The LTSA algorithm comprises three stages:

1. **Nearest Neighbors Search**. Same as standard LLE
2. **Weight Matrix Construction**. Approximately $O[DNk^3] + O[k^2d]$. The first term reflects a similar cost to that of standard LLE.
3. **Partial Eigenvalue Decomposition**. Same as standard LLE

The overall complexity of standard LTSA is $O[D\log(k)N\log(N)] + O[DNk^3] + O[k^2d] + O[dN^2]$.

- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

- "Principal manifolds and nonlinear dimensionality reduction via tangent space alignment" Zhang, Z. & Zha, H. Journal of Shanghai Univ. 8:406 (2004)

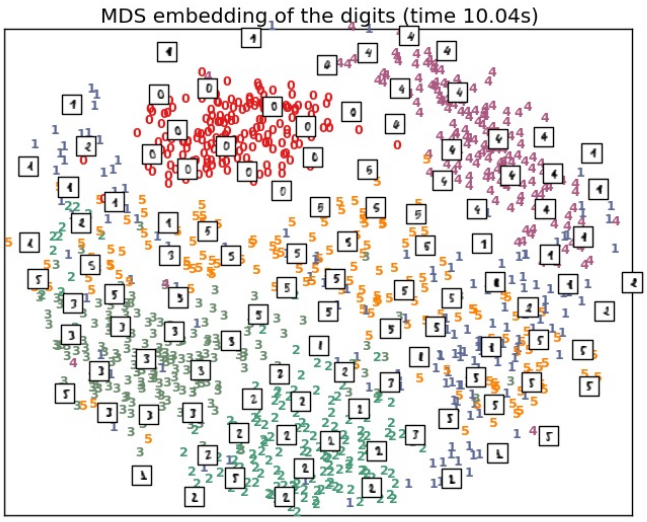
2.2.8. Multi-dimensional Scaling (MDS)

Multidimensional scaling (`MDS`) seeks a low-dimensional representation of the data in which the distances respect well the distances in the original high-dimensional space.

In general, is a technique used for analyzing similarity or dissimilarity data. `MDS` attempts to model similarity or dissimilarity data as distances in a geometric spaces. The data can be ratings of similarity between objects, interaction frequencies of molecules, or trade indices between countries.

There exists two types of MDS algorithm: metric and non metric. In the scikit-learn, the class `MDS` implements both. In Metric MDS, the input similarity matrix arises from a metric (and thus respects the triangular inequality), the distances between output two points are then set to be as close as possible to the similarity or

dissimilarity data. In the non-metric version, the algorithms will try to preserve the order of the distances, and hence seek for a monotonic relationship between the distances in the embedded space and the similarities/dissimilarities.



Let S be the similarity matrix, and X the coordinates of the n input points. Disparities $\hat{d}_{ij}$ are transformation of the similarities chosen in some optimal ways. The objective, called the stress, is then defined by $sum_{i<j} d_{ij}(X) - \hat{d}_{ij}(X)$

2.2.8.1. Metric MDS

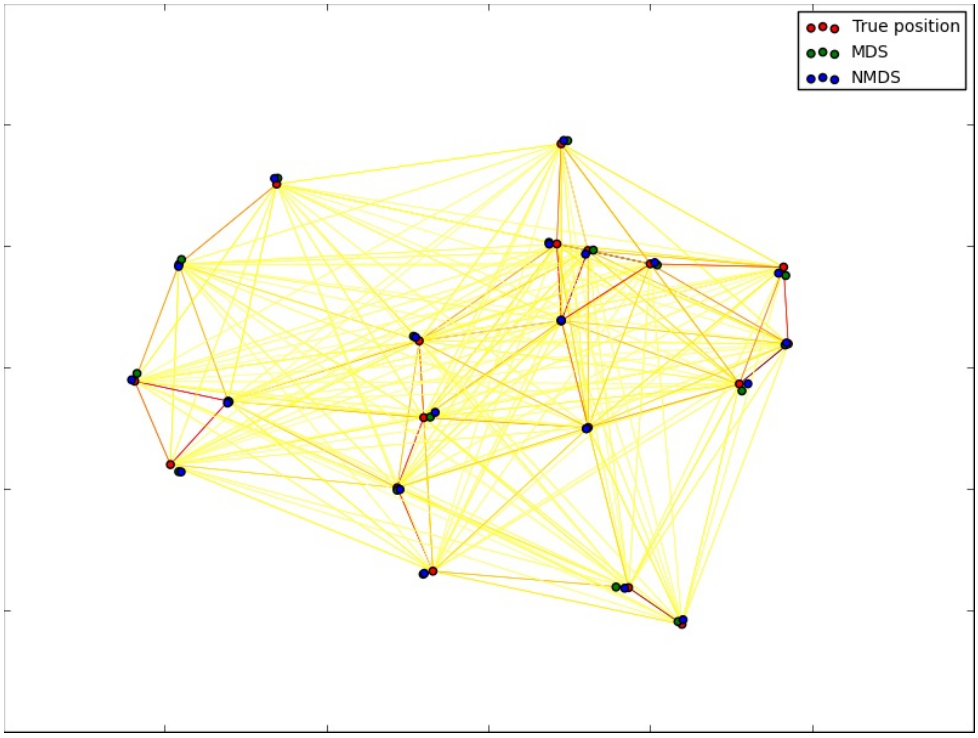
The simplest metric `MDS` model, called *absolute MDS*, disparities are defined by $\hat{d}_{ij} = S_{ij}$. With absolute MDS, the value S_{ij} should then correspond exactly to the distance between point i and j in the embedding point.

Most commonly, disparities are set to $\hat{d}_{ij} = bS_{ij}$.

2.2.8.2. Nonmetric MDS

Non metric `MDS` focuses on the ordination of the data. If $S_{ij} < S_{kl}$, then the embedding should enforce $d_{ij} < d_{jk}$. A simple algorithm to enforce that is to use a monotonic regression of d_{ij} on S_{ij}, yielding disparities $\hat{d}_{ij}$ in the same order as S_{ij}.

A trivial solution to this problem is to set all the points on the origin. In order to avoid that, the disparities $\hat{d}_{ij}$ are normalized.

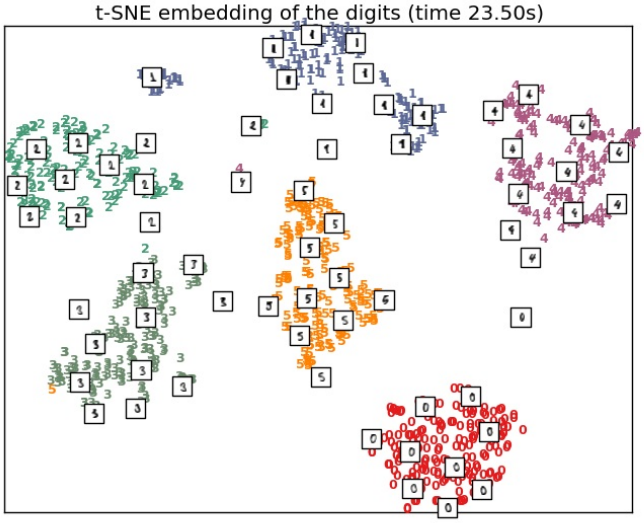


References:

- "Modern Multidimensional Scaling - Theory and Applications" Borg, I.; Groenen P. Springer Series in Statistics (1997)
- "Nonmetric multidimensional scaling: a numerical method" Kruskal, J. Psychometrika, 29 (1964)
- "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis" Kruskal, J. Psychometrika, 29, (1964)

2.2.9. t-distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE (`TSNE`) converts affinities of data points to probabilities. The affinities in the original space are represented by Gaussian joint probabilities and the affinities in the embedded space are represented by Student's t-distributions. The Kullback-Leibler (KL) divergence of the joint probabilities in the original space and the embedded space will be minimized by gradient descent. Note that the KL divergence is not convex, i.e. multiple restarts with different initializations will end up in local minima of the KL divergence. Hence, it is sometimes useful to try different seeds and select the embedding with the lowest KL divergence.



The main purpose of t-SNE is visualization of high-dimensional data. Hence, it works best when the data will be embedded on two or three dimensions.

Optimizing the KL divergence can be a little bit tricky sometimes. There are three parameters that control the optimization of t-SNE:

- early exaggeration factor
- learning rate
- maximum number of iterations

The maximum number of iterations is usually high enough and does not need any tuning. The optimization consists of two phases: the early exaggeration phase and the final optimization. During early exaggeration the joint probabilities in the original space will be artificially increased by multiplication with a given factor. Larger factors result in larger gaps between natural clusters in the data. If the factor is too high, the KL divergence could increase during this phase. Usually it does not have to be tuned. A critical parameter is the learning rate. If it is too low gradient descent will get stuck in a bad local minimum. If it is too high the KL divergence will increase during optimization. More tips can be found in Laurens van der Maaten's FAQ (see references).

Standard t-SNE that has been implemented here is usually much slower than other manifold learning algorithms. The optimization is quite difficult and the computation of the gradient is on $O[dN^2]$, where d is the number of output dimensions and N is the number of samples.

While Isomap, LLE and variants are best suited to unfold a single continuous low dimensional manifold, t-SNE will focus on the local structure of the data and will tend to extract clustered local groups of samples as highlighted on the S-curve example. This ability to group samples based on the local structure might be beneficial to visually disentangle a dataset that comprises several manifolds at once as is the case in the digits dataset.

Also note that the digits labels roughly match the natural grouping found by t-SNE while the linear 2D projection of the PCA model yields a representation where label regions largely overlap. This is a strong clue that this data can be well separated by non linear methods that focus on the local structure (e.g. an SVM with a Gaussian RBF kernel). However, failing to visualize well separated homogeneously labeled groups with t-SNE in 2D does not necessarily implie that the data cannot be correctly classified by a supervised model. It might be the case that 2 dimensions are not enough low to accurately represents the internal structure of the

data.

References:

- "Visualizing High-Dimensional Data Using t-SNE" van der Maaten, L.J.P.; Hinton, G. Journal of Machine Learning Research (2008)
- "t-Distributed Stochastic Neighbor Embedding" van der Maaten, L.J.P.

2.2.10. Tips on practical use

- Make sure the same scale is used over all features. Because manifold learning methods are based on a nearest-neighbor search, the algorithm may perform poorly otherwise. See *StandardScaler* for convenient ways of scaling heterogeneous data.
- The reconstruction error computed by each routine can be used to choose the optimal output dimension. For a d-dimensional manifold embedded in a D-dimensional parameter space, the reconstruction error will decrease as `n_components` is increased until `n_components == d`.
- Note that noisy data can "short-circuit" the manifold, in essence acting as a bridge between parts of the manifold that would otherwise be well-separated. Manifold learning on noisy and/or incomplete data is an active area of research.
- Certain input configurations can lead to singular weight matrices, for example when more than two points in the dataset are identical, or when the data is split into disjointed groups. In this case, `solver='arpack'` will fail to find the null space. The easiest way to address this is to use `solver='dense'` which will work on a singular matrix, though it may be very slow depending on the number of input points. Alternatively, one can attempt to understand the source of the singularity: if it is due to disjoint sets, increasing `n_neighbors` may help. If it is due to identical points in the dataset, removing these points may help.

See also: *Totally Random Trees Embedding* can also be useful to derive non-linear representations of feature space, also it does not perform dimensionality reduction.

2.3. Clustering

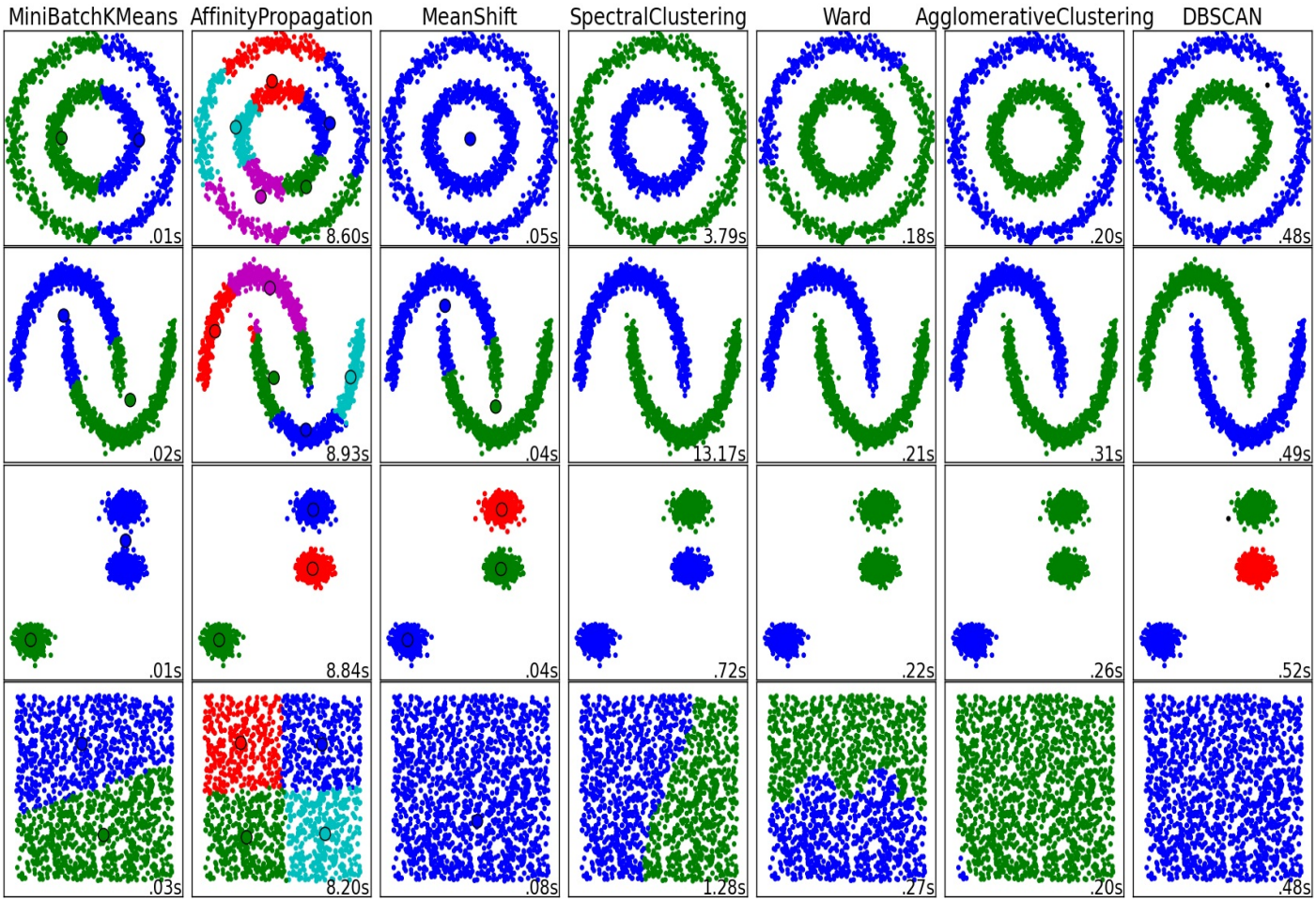
Clustering of unlabeled data can be performed with the module `sklearn.cluster`.

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.

Input data

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 `sklearn.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 `sklearn.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.

2.3.1. Overview of clustering methods



A comparison of the clustering algorithms in scikit-learn

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large `n_samples`, medium `n_clusters` with *MiniBatch code*	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium `n_samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters, linkage type, distance	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large `n_samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes	Distances between nearest points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers

Non-flat geometry clustering is useful when the clusters have a specific shape, i.e. a non-flat manifold, and the standard euclidean distance is not the right metric. This case arises in the two top rows of the figure above.

Gaussian mixture models, useful for clustering, are described in *another chapter of the documentation* dedicated to mixture models. KMeans can be seen as a special case of Gaussian mixture model with equal covariance per component.

2.3.2. 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* <*inertia*> or within-cluster sum-of-squares. This algorithm requires the number of clusters to be specified. It scales well to large number of samples and has been used across a large range of application areas in many different fields.

The k-means algorithm divides a set of N samples X into K disjoint clusters C, each described by the mean μ_j of the samples in the cluster. The means are commonly called the cluster “centroids”; note that they are not, in general, points from X, although they live in the same space. The K-means algorithm aims to choose centroids that minimise the *inertia*, or within-cluster sum of squared criterion:

$$\sum_{i=0}^{n} \min_{\mu_j \in C}(||x_j - \mu_i||^2)$$

Inertia, or the within-cluster sum of squares criterion, can be recognized as a measure of how internally coherent clusters are. It suffers from various drawbacks:

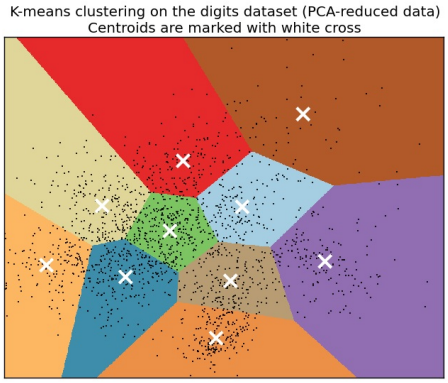
- Inertia makes the assumption that clusters are convex and isotropic, which is not always the case. It responds poorly to elongated clusters, or manifolds with irregular shapes.
- Inertia is not a normalized metric: we just know that lower values are better and zero is optimal. But in very high-dimensional spaces, Euclidean distances tend to become inflated (this is an instance of the so-called “curse of dimensionality”). Running a dimensionality reduction algorithm such as *PCA* <*PCA*> prior to k-means clustering can alleviate this problem and speed up the computations.

K-means is often referred to as Lloyd’s algorithm. In basic terms, the algorithm has three steps. The first step chooses the initial centroids, with the most basic method being to choose k samples from the dataset X. After initialization, K-means consists of looping between the two other steps. The first step assigns each sample to its nearest centroid. The second step creates new centroids by taking the mean value of all of the samples assigned to each previous centroid. The difference between the old and the new centroids are computed and the algorithm repeats these last two steps until this value is less than a threshold. In other words, it repeats until the centroids do not move significantly.

K-means is equivalent to the expectation-maximization algorithm with a small, all-equal, diagonal covariance matrix.

The algorithm can also be understood through the concept of Voronoi diagrams. First the Voronoi diagram of the points is calculated using the current centroids. Each segment in the Voronoi diagram becomes a separate cluster. Secondly, the centroids are updated to the mean of each segment. The algorithm then repeats this until a stopping criterion is fulfilled. Usually, the algorithm stops when the relative decrease in the objective function between iterations is less than the given tolerance value. This is not the case in this

implementation: iteration stops when centroids move less than the tolerance.



Given enough time, K-means will always converge, however this may be to a local minimum. This is highly dependent on the initialization of the centroids. As a result, the computation is often done several times, with different initializations of the centroids. One method to help address this issue is the k-means++ initialization scheme, which has been implemented in scikit-learn (use the `init='kmeans++'` parameter). This initializes the centroids to be (generally) distant from each other, leading to provably better results than random initialization, as shown in the reference.

A parameter can be given to allow K-means to be run in parallel, called `n_jobs`. Giving this parameter a positive value uses that many processors (default: 1). A value of -1 uses all available processors, with -2 using one less, and so on. Parallelization generally speeds up computation at the cost of memory (in this case, multiple copies of centroids need to be stored, one for each job).

Warning: The parallel version of K-Means is broken on OS X when numpy uses the Accelerate Framework. This is expected behavior: Accelerate can be called after a fork but you need to execv the subprocess with the Python binary (which multiprocessing does not do under posix).

K-means can be used for vector quantization. This is achieved using the transform method of a trained model of **`KMeans`**.

Examples:

- *A demo of K-Means clustering on the handwritten digits data*: Clustering handwritten digits

References:

- "k-means++: The advantages of careful seeding" Arthur, David, and Sergei Vassilvitskii, *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics (2007)

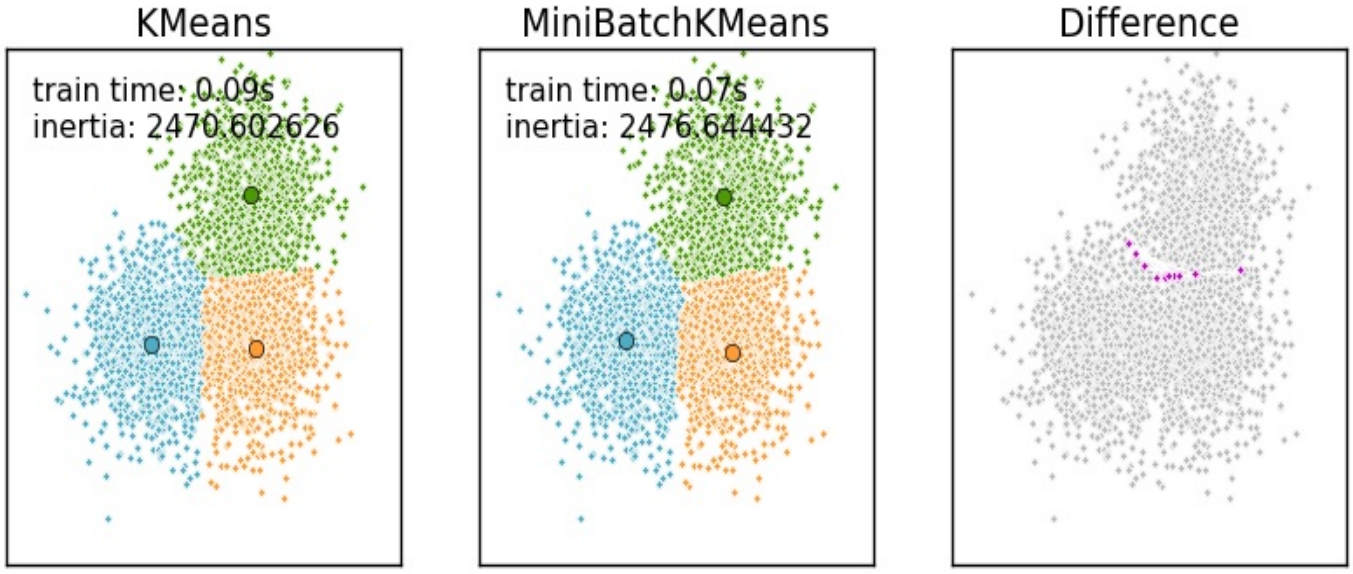
2.3.2.1. Mini Batch K-Means

The **`MiniBatchKMeans`** is a variant of the **`KMeans`** algorithm which uses mini-batches to reduce the computation time, while still attempting to optimise the same objective function. Mini-batches are subsets of the input data, randomly sampled in each training iteration. These mini-batches drastically reduce the amount of computation required to converge to a local solution. In contrast to other algorithms that reduce the convergence time of k-means, mini-batch k-means produces results that are generally only slightly worse than the standard algorithm.

The algorithm iterates between two major steps, similar to vanilla k-means. In the first step, b samples are drawn randomly from the dataset, to form a mini-batch. These are then assigned to the nearest centroid. In the second step, the centroids are updated. In contrast to k-means, this is done on a per-sample basis. For each sample in the mini-batch, the assigned centroid is updated by taking the streaming average of the

sample and all previous samples assigned to that centroid. This has the effect of decreasing the rate of change for a centroid over time. These steps are performed until convergence or a predetermined number of iterations is reached.

`MiniBatchKMeans` converges faster than `KMeans`, but the quality of the results is reduced. In practice this difference in quality can be quite small, as shown in the example and cited reference.



Examples:

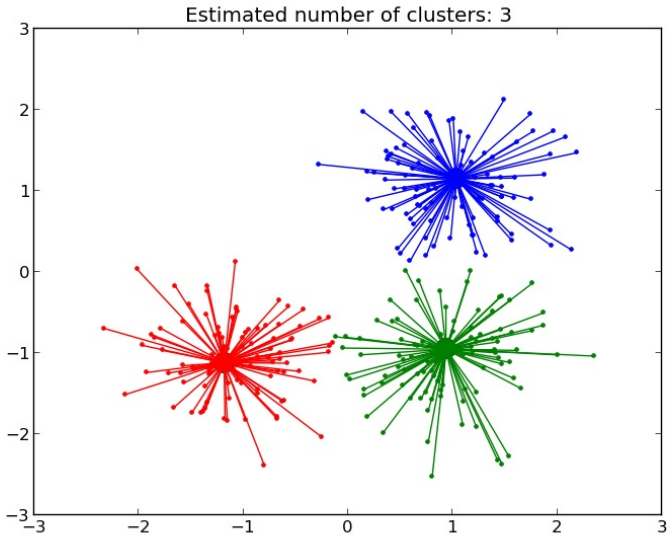
- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*: Comparison of KMeans and MiniBatchKMeans
- *Clustering text documents using k-means*: Document clustering using sparse MiniBatchKMeans
- *Online learning of a dictionary of parts of faces*

References:

- "Web Scale K-Means clustering" D. Sculley, *Proceedings of the 19th international conference on World wide web* (2010)

2.3.3. Affinity Propagation

`AffinityPropagation` creates clusters by sending messages between pairs of samples until convergence. A dataset is then described using a small number of exemplars, which are identified as those most representative of other samples. The messages sent between pairs represent the suitability for one sample to be the exemplar of the other, which is updated in response to the values from other pairs. This updating happens iteratively until convergence, at which point the final exemplars are chosen, and hence the final clustering is given.



Affinity Propagation can be interesting as it chooses the number of clusters based on the data provided. For this purpose, the two important parameters are the *preference*, which controls how many exemplars are used, and the *damping factor*.

The main drawback of Affinity Propagation is its complexity. The algorithm has a time complexity of the order $O(N^2T)$, where N is the number of samples and T is the number of iterations until convergence. Further, the memory complexity is of the order $O(N^2)$ if a dense similarity matrix is used, but reducible if a sparse similarity matrix is used. This makes Affinity Propagation most appropriate for small to medium sized datasets.

Examples:

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

Algorithm description: The messages sent between points belong to one of two categories. The first is the responsibility $r(i, k)$, which is the accumulated evidence that sample k should be the exemplar for sample i. The second is the availability $a(i, k)$ which is the accumulated evidence that sample i should choose sample k to be its exemplar, and considers the values for all other samples that k should be an exemplar. In this way, exemplars are chosen by samples if they are (1) similar enough to many samples and (2) chosen by many samples to be representative of themselves.

More formally, the responsibility of a sample k to be the exemplar of sample i is given by:

$$r(i, k) \leftarrow s(i, k) - max[a(i, \acute{k}) + s(i, \acute{k}) \forall \acute{k} \neq k]$$

Where $s(i, k)$ is the similarity between samples i and k. The availability of sample k to be the exemplar of sample i is given by:

$$a(i, k) \leftarrow min[0, r(k, k) + \sum_{\acute{i}\ s.t.\ \acute{i} \notin \{i,k\}} r(\acute{i}, k)]$$

To begin with, all values for r and a are set to zero, and the calculation of each iterates until convergence.

2.3.4. Mean Shift

`MeanShift` clustering aims to discover *blobs* in a smooth density of samples. It is a centroid based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.

Given a candidate centroid x_i for iteration t, the candidate is updated according to the following equation:

$$x_i^{t+1} = x_i^t + m(x_i^t)$$

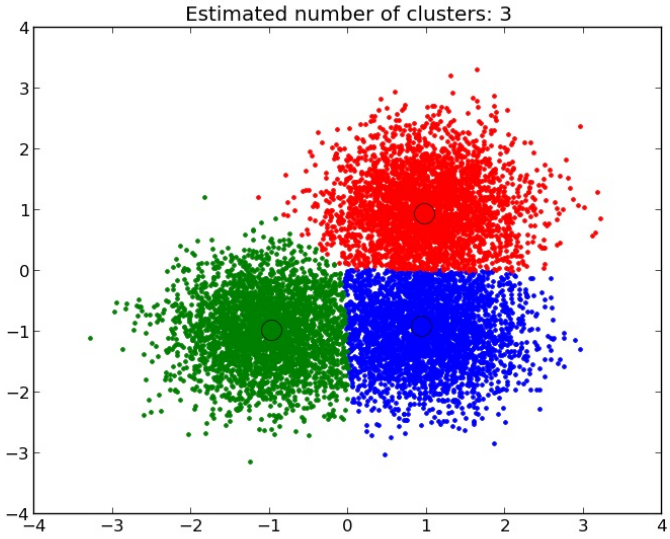
Where $N(x_i)$ is the neighborhood of samples within a given distance around x_i and m is the *mean shift* vector that is computed for each centroid that points towards a region of the maximum increase in the density of points. This is computed using the following equation, effectively updating a centroid to be the mean of the samples within its neighborhood:

$$m(x_i) = \frac{\sum_{x_j \in N(x_i)} K(x_j - x_i) x_j}{\sum_{x_j \in N(x_i)} K(x_j - x_i)}$$

The algorithm automatically sets the number of clusters, instead of relying on a parameter `bandwidth`, which dictates the size of the region to search through. This parameter can be set manually, but can be estimated using the provided `estimate_bandwidth` function, which is called if the bandwidth is not set.

The algorithm is not highly scalable, as it requires multiple nearest neighbor searches during the execution of the algorithm. The algorithm is guaranteed to converge, however the algorithm will stop iterating when the change in centroids is small.

Labelling a new sample is performed by finding the nearest centroid for a given sample.



Examples:

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

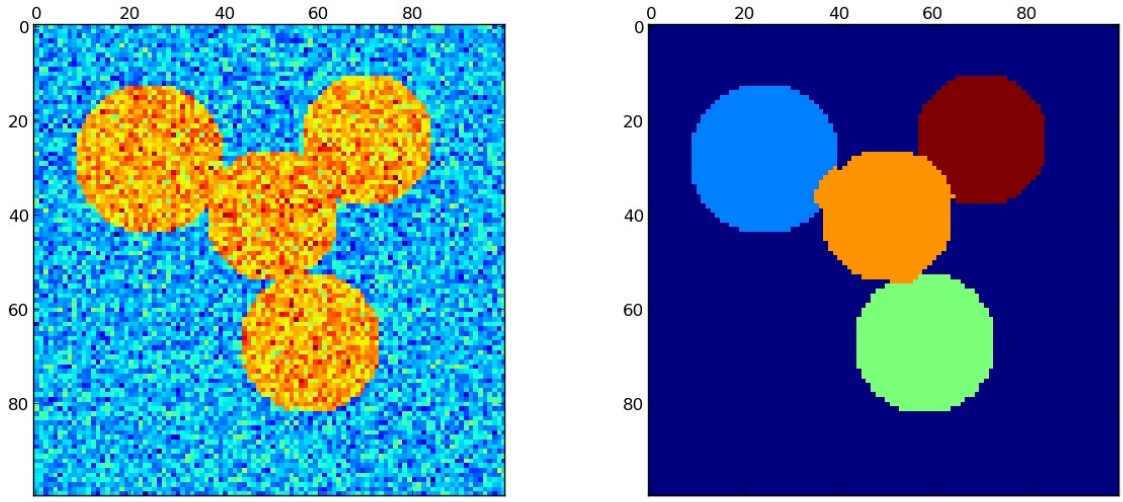
References:

- "Mean shift: A robust approach toward feature space analysis." D. Comaniciu, & P. Meer *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2002)

2.3.5. Spectral clustering

`SpectralClustering` does a 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 of the 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.



Warning: Transforming distance to well-behaved similarities

Note that if the values of your similarity matrix are not well distributed, e.g. with negative values or with a distance matrix rather than a similarity, the spectral problem will be singular and the problem not solvable. In which case it is advised to apply a transformation to the entries of the matrix. For instance, in the case of a signed distance matrix, is common to apply a heat kernel:

```
similarity = np.exp(-beta * distance / distance.std())
```

See the examples for such an application.

Examples:

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

2.3.5.1. Different label assignment strategies

Different label assignment strategies can be used, corresponding to the `assign_labels` parameter of `SpectralClustering`. The `"kmeans"` strategy can match finer details of the data, but it can be more unstable. In particular, unless you control the `random_state`, it may not be reproducible from run-to-run, as it depends on a random initialization. On the other hand, the `"discretize"` strategy is 100% reproducible, but it tends to create parcels of fairly even and geometrical shape.

``assign_labels="kmeans"`	`assign_labels="discretize"`
Spectral clustering: kmeans, 221.49s	Spectral clustering: discretize, 206.04s
	

References:

- "A Tutorial on Spectral Clustering" Ulrike von Luxburg, 2007
- "Normalized cuts and image segmentation" Jianbo Shi, Jitendra Malik, 2000
- "A Random Walks View of Spectral Segmentation" Marina Meila, Jianbo Shi, 2001
- "On Spectral Clustering: Analysis and an algorithm" Andrew Y. Ng, Michael I. Jordan, Yair Weiss, 2001

2.3.6. Hierarchical clustering

Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample. See the Wikipedia page for more details.

The `AgglomerativeClustering` object performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together. The linkage criteria determines the metric used for the merge strategy:

- **Ward** minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.

- **Maximum** or **complete linkage** minimizes the maximum distance between observations of pairs of clusters.
- **Average linkage** minimizes the average of the distances between all observations of pairs of clusters.

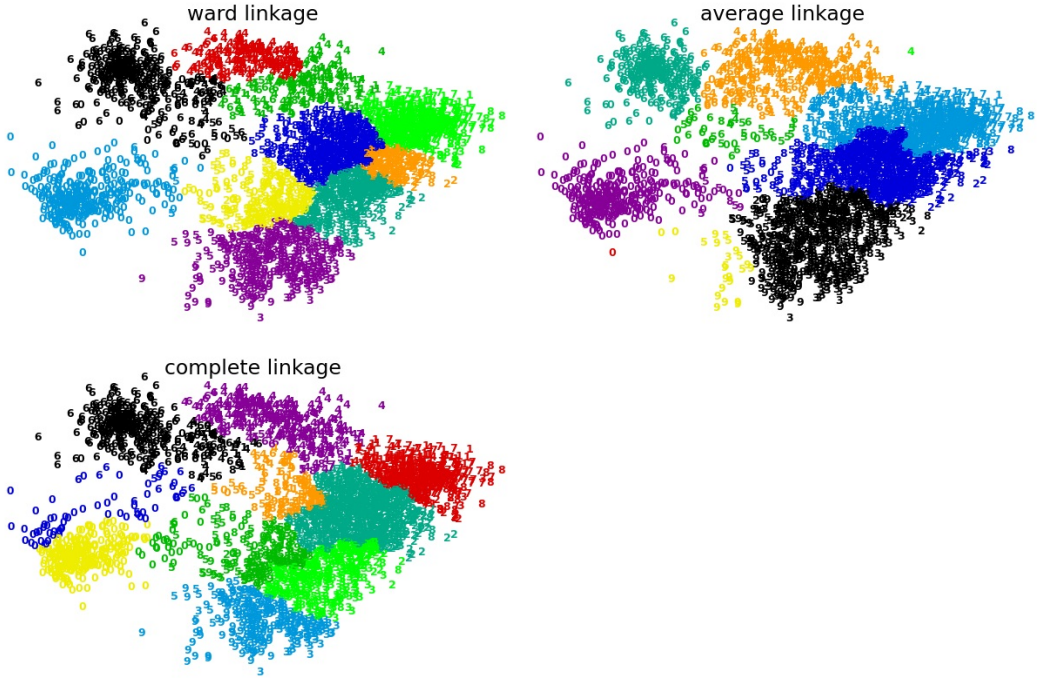
`AgglomerativeClustering` can also scale to large number of samples when it is used jointly with a connectivity matrix, but is computationally expensive when no connectivity constraints are added between samples: it considers at each step all the possible merges.

`FeatureAgglomeration`

The `FeatureAgglomeration` uses agglomerative clustering to group together features that look very similar, thus decreasing the number of features. It is a dimensionality reduction tool, see *Unsupervised data reduction*.

2.3.6.1. Different linkage type: Ward, complete and average linkage

`AgglomerativeClustering` supports Ward, average, and complete linkage strategies.



Agglomerative cluster has a “rich get richer” behavior that leads to uneven cluster sizes. In this regard, complete linkage is the worst strategy, and Ward gives the most regular sizes. However, the affinity (or distance used in clustering) cannot be varied with Ward, thus for non Euclidean metrics, average linkage is a good alternative.

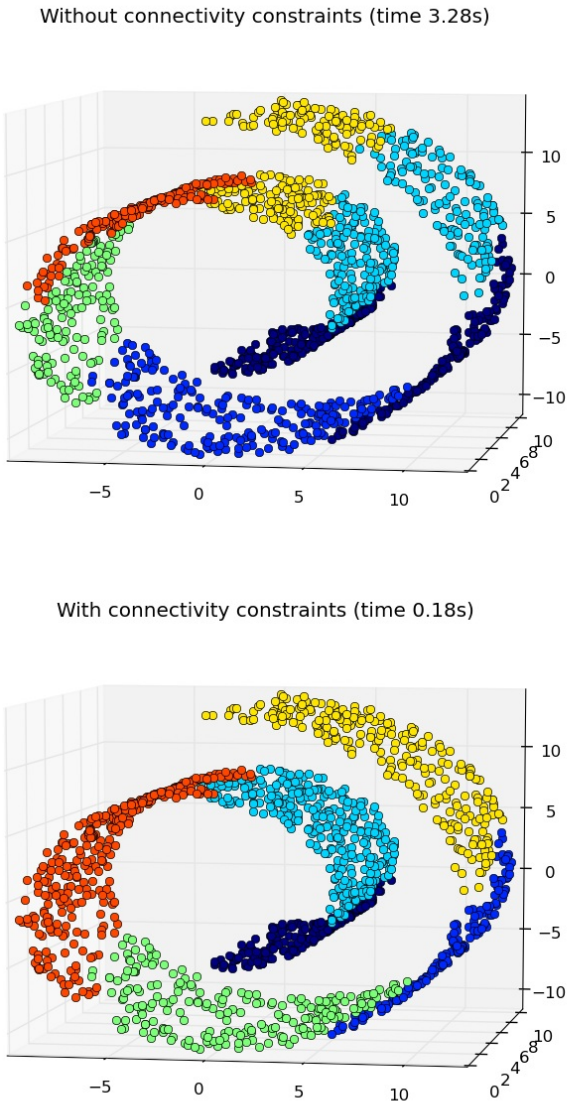
Examples:

- *Various Agglomerative Clustering on a 2D embedding of digits*: exploration of the different linkage strategies in a real dataset.

2.3.6.2. Adding connectivity constraints

An interesting aspect of `AgglomerativeClustering` is that connectivity constraints can be added to this algorithm (only adjacent clusters can be merged together), through a connectivity matrix that defines for

each sample the neighboring samples following a given structure of the data. For instance, in the swiss-roll example below, the connectivity constraints forbid the merging of points that are not adjacent on the swiss roll, and thus avoid forming clusters that extend across overlapping folds of the roll.



These constraint are useful to impose a certain local structure, but they also make the algorithm faster, especially when the number of the samples is high.

The connectivity constraints are imposed via an connectivity matrix: a scipy sparse matrix that has elements only at the intersection of a row and a column with indices of the dataset that should be connected. This matrix can be constructed from a-priori information: for instance, you may wish to cluster web pages by only merging pages with a link pointing from one to another. It can also be learned from the data, for instance using `sklearn.neighbors.kneighbors_graph` to restrict merging to nearest neighbors as in *this example*, or using `sklearn.feature_extraction.image.grid_to_graph` to enable only merging of neighboring pixels on an image, as in the *Lena* example.

Examples:

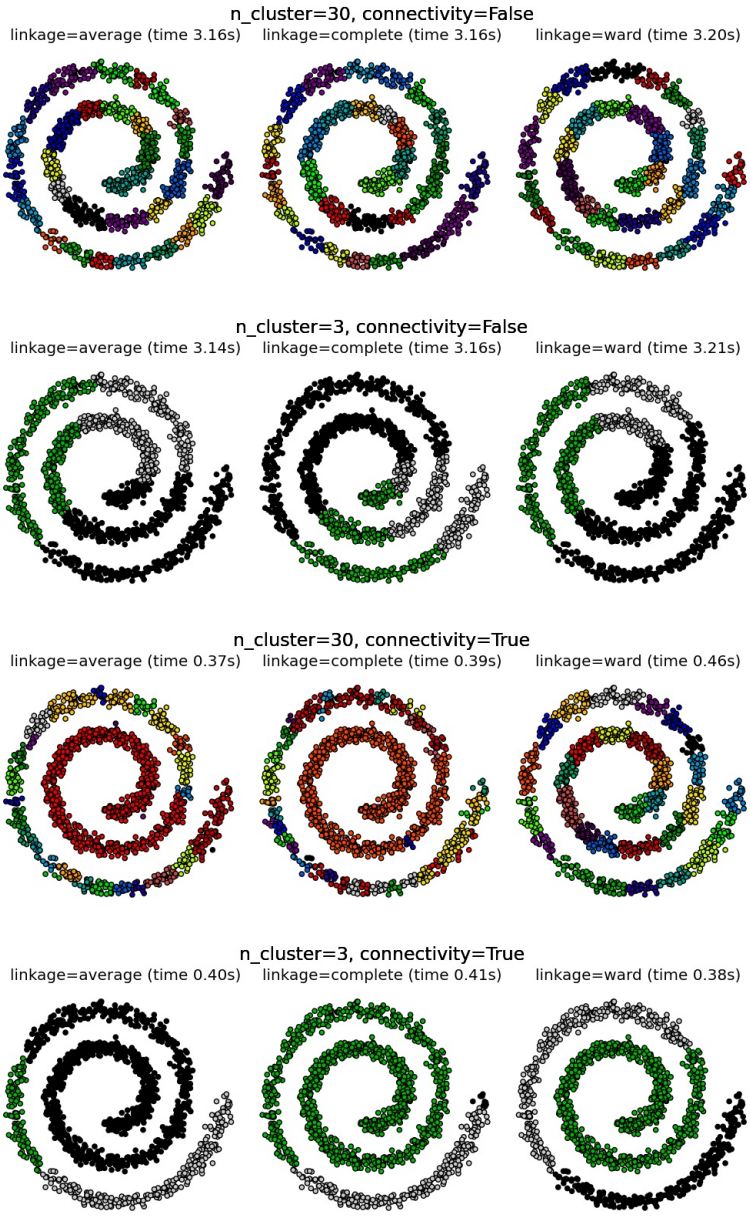
- *A demo of structured Ward hierarchical clustering on Lena image*: Ward clustering to split the image of lena in regions.
- *Hierarchical clustering: structured vs unstructured ward*: Example of Ward algorithm on a swiss-roll,

comparison of structured approaches versus unstructured approaches.

- *Feature agglomeration vs. univariate selection*: Example of dimensionality reduction with feature agglomeration based on Ward hierarchical clustering.
- *Agglomerative clustering with and without structure*

Warning: Connectivity constraints with average and complete linkage

Connectivity constraints and complete or average linkage can enhance the 'rich getting richer' aspect of agglomerative clustering, particularly so if they are built with `sklearn.neighbors.kneighbors_graph`. In the limit of a small number of clusters, they tend to give a few macroscopically occupied clusters and almost empty ones. (see the discussion in *Agglomerative clustering with and without structure*).



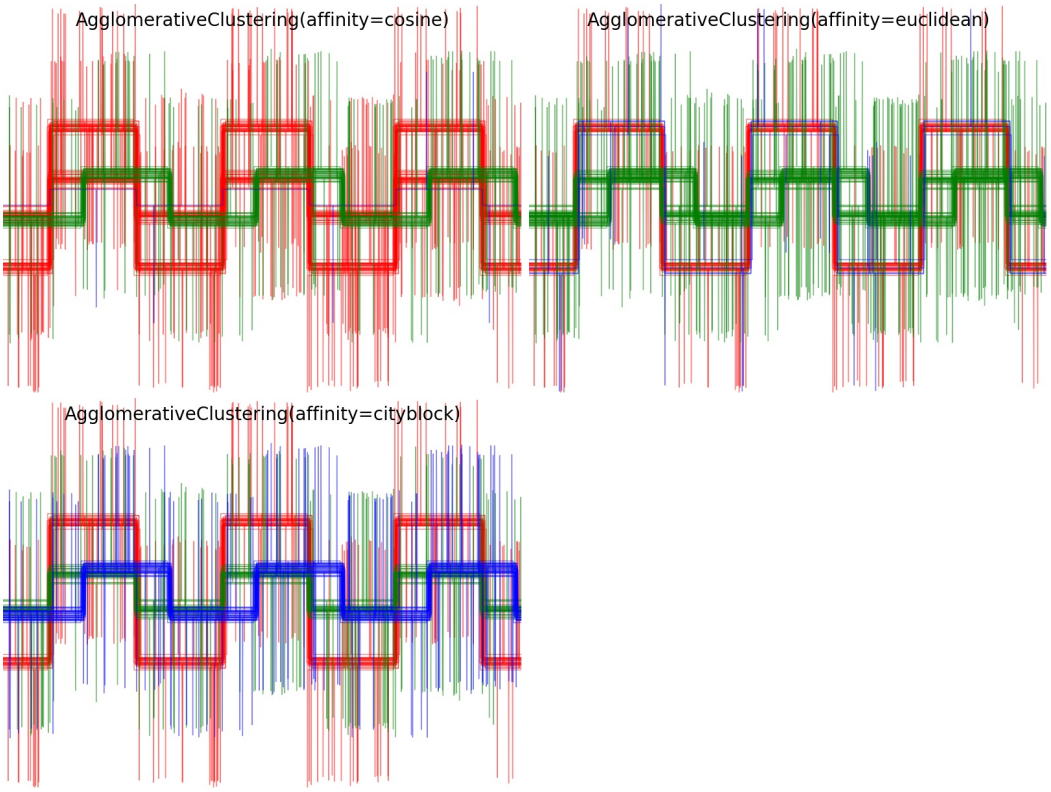
2.3.6.3. Varying the metric

Average and complete linkage can be used with a variety of distances (or affinities), in particular Euclidean distance (*l2*), Manhattan distance (or Cityblock, or *l1*), cosine distance, or any precomputed affinity matrix.

- *l1* distance is often good for sparse features, or sparse noise: ie many of the features are zero, as in

text mining using occurences of rare words.
- *cosine* distance is interesting because it is invariant to global scalings of the signal.

The guidelines for choosing a metric is to use one that maximizes the distance between samples in different classes, and minimizes that within each class.



Examples:

- *Agglomerative clustering with different metrics*

2.3.7. DBSCAN

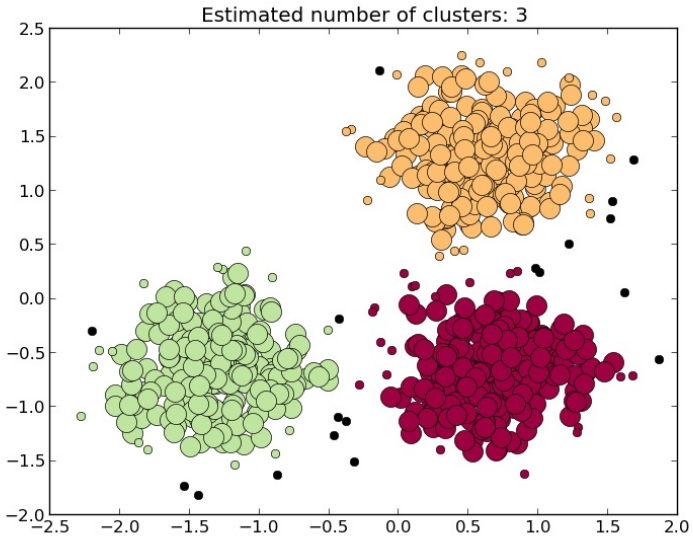
The `DBSCAN` algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to k-means which assumes that clusters are convex shaped. The central component to the DBSCAN is the concept of *core samples*, which are samples that are in areas of high density. A cluster is therefore a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples). There are two parameters to the algorithm, `min_samples` and `eps`, which define formally what we mean when we say *dense*. Higher `min_samples` or lower `eps` indicate higher density necessary to form a cluster.

More formally, we define a core sample as being a sample in the dataset such that there exist `min_samples` other samples within a distance of `eps`, which are defined as *neighbors* of the core sample. This tells us that the core sample is in a dense area of the vector space. A cluster is a set of core samples, that can be built by recursively by taking a core sample, finding all of its neighbors that are core samples, finding all of *their* neighbors that are core samples, and so on. A cluster also has a set of non-core samples, which are samples that are neighbors of a core sample in the cluster but are not themselves core samples. Intuitively, these samples are on the fringes of a cluster.

Any core sample is part of a cluster, by definition. Further, any cluster has at least `min_samples` points in it,

following the definition of a core sample. For any sample that is not a core sample, and does have a distance higher than `eps` to any core sample, it is considered an outlier by the algorithm.

In the figure below, the color indicates cluster membership, with large circles indicating core samples found by the algorithm. Smaller circles are non-core samples that are still part of a cluster. Moreover, the outliers are indicated by black points below.



Examples:

- *Demo of DBSCAN clustering algorithm*

Implementation

The algorithm is non-deterministic, but the core samples will always belong to the same clusters (although the labels may be different). The non-determinism comes from deciding to which cluster a non-core sample belongs. A non-core sample can have a distance lower than `eps` to two core samples in different clusters. By the triangular inequality, those two core samples must be more distant than `eps` from each other, or they would be in the same cluster. The non-core sample is assigned to whichever cluster is generated first, where the order is determined randomly. Other than the ordering of the dataset, the algorithm is deterministic, making the results relatively stable between runs on the same data.

The current implementation uses ball trees and kd-trees to determine the neighborhood of points, which avoids calculating the full distance matrix (as was done in scikit-learn versions before 0.14). The possibility to use custom metrics is retained; for details, see **`NearestNeighbors`**.

References:

- “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise” Ester, M., H. P. Kriegel, J. Sander, and X. Xu, In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226–231. 1996

2.3.8. Clustering performance evaluation

Evaluating the performance of a clustering algorithm is not as trivial as counting the number of errors or the precision and recall of a supervised classification algorithm. In particular any evaluation metric should not take the absolute values of the cluster labels into account but rather if this clustering define separations of the data similar to some ground truth set of classes or satisfying some assumption such that members belong to the same class are more similar that members of different classes according to some similarity metric.

2.3.8.1. Adjusted Rand index

2.3.8.1.1. Presentation and usage

Given the knowledge of the ground truth class assignments `labels_true` and our clustering algorithm assignments of the same samples `labels_pred`, the **adjusted Rand index** is a function that measures the **similarity** of the two assignments, ignoring permutations and **with chance normalization**:

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.adjusted_rand_score(labels_true, labels_pred)
0.24...
```

One can permute 0 and 1 in the predicted labels, rename 2 to 3, and get the same score:

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
0.24...
```

Furthermore, `adjusted_rand_score` is **symmetric**: swapping the argument does not change the score. It can thus be used as a **consensus measure**:

```
>>> metrics.adjusted_rand_score(labels_pred, labels_true)
0.24...
```

Perfect labeling is scored 1.0:

```
>>> labels_pred = labels_true[:]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
1.0
```

Bad (e.g. independent labelings) have negative or close to 0.0 scores:

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
-0.12...
```

2.3.8.1.2. Advantages

- **Random (uniform) label assignments have a ARI score close to 0.0** for any value of `n_clusters` and `n_samples` (which is not the case for raw Rand index or the V-measure for instance).
- **Bounded range [-1, 1]**: negative values are bad (independent labelings), similar clusterings have a positive ARI, 1.0 is the perfect match score.
- **No assumption is made on the cluster structure**: can be used to compare clustering algorithms such as k-means which assumes isotropic blob shapes with results of spectral clustering algorithms which can find cluster with “folded” shapes.

2.3.8.1.3. Drawbacks

- Contrary to inertia, **ARI requires knowledge of the ground truth classes** while is almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

  However ARI can also be useful in a purely unsupervised setting as a building block for a Consensus Index that can be used for clustering model selection (TODO).

Examples:

- *Adjustment for chance in clustering performance evaluation*: Analysis of the impact of the dataset size on the value of clustering measures for random assignments.

2.3.8.1.4. Mathematical formulation

If C is a ground truth class assignment and K the clustering, let us define a and b as:

- a, the number of pairs of elements that are in the same set in C and in the same set in K
- b, the number of pairs of elements that are in different sets in C and in different sets in K

The raw (unadjusted) Rand index is then given by:

$$\mathrm{RI} = \frac{a + b}{C_2^{n_{samples}}}$$

Where $C_2^{n_{samples}}$ is the total number of possible pairs in the dataset (without ordering).

However the RI score does not guarantee that random label assignments will get a value close to zero (esp. if the number of clusters is in the same order of magnitude as the number of samples).

To counter this effect we can discount the expected RI $E[\mathrm{RI}]$ of random labelings by defining the adjusted Rand index as follows:

$$\mathrm{ARI} = \frac{\mathrm{RI} - E[\mathrm{RI}]}{\max(\mathrm{RI}) - E[\mathrm{RI}]}$$

References

- Comparing Partitions L. Hubert and P. Arabie, Journal of Classification 1985
- Wikipedia entry for the adjusted Rand index

2.3.8.2. Mutual Information based scores

2.3.8.2.1. Presentation and usage

Given the knowledge of the ground truth class assignments `labels_true` and our clustering algorithm assignments of the same samples `labels_pred`, the **Mutual Information** is a function that measures the **agreement** of the two assignments, ignoring permutations. Two different normalized versions of this

measure are available, **Normalized Mutual Information(NMI)** and **Adjusted Mutual Information(AMI)**. NMI is often used in the literature while AMI was proposed more recently and is **normalized against chance**:

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
0.22504...
```

One can permute 0 and 1 in the predicted labels, rename 2 to 3 and get the same score:

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
0.22504...
```

All, **`mutual_info_score`**, **`adjusted_mutual_info_score`** and **`normalized_mutual_info_score`** are symmetric: swapping the argument does not change the score. Thus they can be used as a **consensus measure**:

```
>>> metrics.adjusted_mutual_info_score(labels_pred, labels_true)
0.22504...
```

Perfect labeling is scored 1.0:

```
>>> labels_pred = labels_true[:]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
1.0

>>> metrics.normalized_mutual_info_score(labels_true, labels_pred)
1.0
```

This is not true for `mutual_info_score`, which is therefore harder to judge:

```
>>> metrics.mutual_info_score(labels_true, labels_pred)
0.69...
```

Bad (e.g. independent labelings) have non-positive scores:

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
-0.10526...
```

2.3.8.2.2. Advantages

- **Random (uniform) label assignments have a AMI score close to 0.0** for any value of `n_clusters` and `n_samples` (which is not the case for raw Mutual Information or the V-measure for instance).
- **Bounded range [0, 1]**: Values close to zero indicate two label assignments that are largely independent, while values close to one indicate significant agreement. Further, values of exactly 0 indicate **purely** independent label assignments and a AMI of exactly 1 indicates that the two label assignments are equal (with or without permutation).
- **No assumption is made on the cluster structure**: can be used to compare clustering algorithms such as k-means which assumes isotropic blob shapes with results of spectral clustering algorithms which can find cluster with “folded” shapes.

2.3.8.2.3. Drawbacks

- Contrary to inertia, **MI-based measures require the knowledge of the ground truth classes** while

almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

However MI-based measures can also be useful in purely unsupervised setting as a building block for a Consensus Index that can be used for clustering model selection.

- NMI and MI are not adjusted against chance.

Examples:

- *Adjustment for chance in clustering performance evaluation*: Analysis of the impact of the dataset size on the value of clustering measures for random assignments. This example also includes the Adjusted Rand Index.

2.3.8.2.4. Mathematical formulation

Assume two label assignments (of the same N objects), U and V. Their entropy is the amount of uncertainty for a partition set, defined by:

$$H(U) = \sum_{i=1}^{|U|} P(i) \log(P(i))$$

where $P(i) = |U_i|/N$ is the probability that an object picked at random from U falls into class U_i. Likewise for V:

$$H(V) = \sum_{j=1}^{|V|} P'(j) \log(P'(j))$$

With $P'(j) = |V_j|/N$. The mutual information (MI) between U and V is calculated by:

$$\mathrm{MI}(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} P(i, j) \log \left(\frac{P(i, j)}{P(i)P'(j)} \right)$$

where $P(i, j) = |U_i \cap V_j|/N$ is the probability that an object picked at random falls into both classes U_i and V_j.

The normalized mutual information is defined as

$$\mathrm{NMI}(U, V) = \frac{\mathrm{MI}(U, V)}{\sqrt{H(U)H(V)}}$$

This value of the mutual information and also the normalized variant is not adjusted for chance and will tend to increase as the number of different labels (clusters) increases, regardless of the actual amount of “mutual information” between the label assignments.

The expected value for the mutual information can be calculated using the following equation, from Vinh, Epps, and Bailey, (2009). In this equation, $a_i = |U_i|$ (the number of elements in U_i) and $b_j = |V_j|$ (the number of elements in V_j).

$$E[\mathrm{MI}(U,V)] = \sum_{i=1}^{|U|}\sum_{j=1}^{|V|}\sum_{n_{ij}=(a_i+b_j-N)^+}^{\min(a_i,b_j)} \frac{n_{ij}}{N}\log\left(\frac{N.n_{ij}}{a_i b_j}\right)\frac{a_i!b_j!(N-a_i)!(N-b_j)!}{N!n_{ij}!(a_i-n_{ij})!(b_j-n_{ij})!(N-a_i-b_j+n_{ij})!}$$

Using the expected value, the adjusted mutual information can then be calculated using a similar form to that of the adjusted Rand index:

$$\mathrm{AMI} = \frac{\mathrm{MI} - E[\mathrm{MI}]}{\max(H(U), H(V)) - E[\mathrm{MI}]}$$

References

- Strehl, Alexander, and Joydeep Ghosh (2002). “Cluster ensembles – a knowledge reuse framework for combining multiple partitions”. Journal of Machine Learning Research 3: 583–617. doi:10.1162/153244303321897735
- Vinh, Epps, and Bailey, (2009). “Information theoretic measures for clusterings comparison”. Proceedings of the 26th Annual International Conference on Machine Learning - ICML ‘09. doi:10.1145/1553374.1553511. ISBN 9781605585161.
- Vinh, Epps, and Bailey, (2010). Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance}, JMLR http://jmlr.csail.mit.edu/papers/volume11/vinh10a/vinh10a.pdf
- Wikipedia entry for the (normalized) Mutual Information
- Wikipedia entry for the Adjusted Mutual Information

2.3.8.3. Homogeneity, completeness and V-measure

2.3.8.3.1. Presentation and usage

Given the knowledge of the ground truth class assignments of the samples, it is possible to define some intuitive metric using conditional entropy analysis.

In particular Rosenberg and Hirschberg (2007) define the following two desirable objectives for any cluster assignment:

- **homogeneity**: each cluster contains only members of a single class.
- **completeness**: all members of a given class are assigned to the same cluster.

We can turn those concept as scores `homogeneity_score` and `completeness_score`. Both are bounded below by 0.0 and above by 1.0 (higher is better):

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.homogeneity_score(labels_true, labels_pred)
0.66...

>>> metrics.completeness_score(labels_true, labels_pred)
0.42...
```

Their harmonic mean called **V-measure** is computed by `v_measure_score`:

```
>>> metrics.v_measure_score(labels_true, labels_pred)
0.51...
```

The V-measure is actually equivalent to the mutual information (NMI) discussed above normalized by the sum of the label entropies [B2011].

Homogeneity, completeness and V-measure can be computed at once using **homogeneity_completeness_v_measure** as follows:

```
>>> metrics.homogeneity_completeness_v_measure(labels_true, labels_pred)
...
(0.66..., 0.42..., 0.51...)
```

The following clustering assignment is slightly better, since it is homogeneous but not complete:

```
>>> labels_pred = [0, 0, 0, 1, 2, 2]
>>> metrics.homogeneity_completeness_v_measure(labels_true, labels_pred)
...
(1.0, 0.68..., 0.81...)
```

Note: **v_measure_score** is **symmetric**: it can be used to evaluate the **agreement** of two independent assignments on the same dataset.

This is not the case for **completeness_score** and **homogeneity_score**: both are bound by the relationship:

```
homogeneity_score(a, b) == completeness_score(b, a)
```

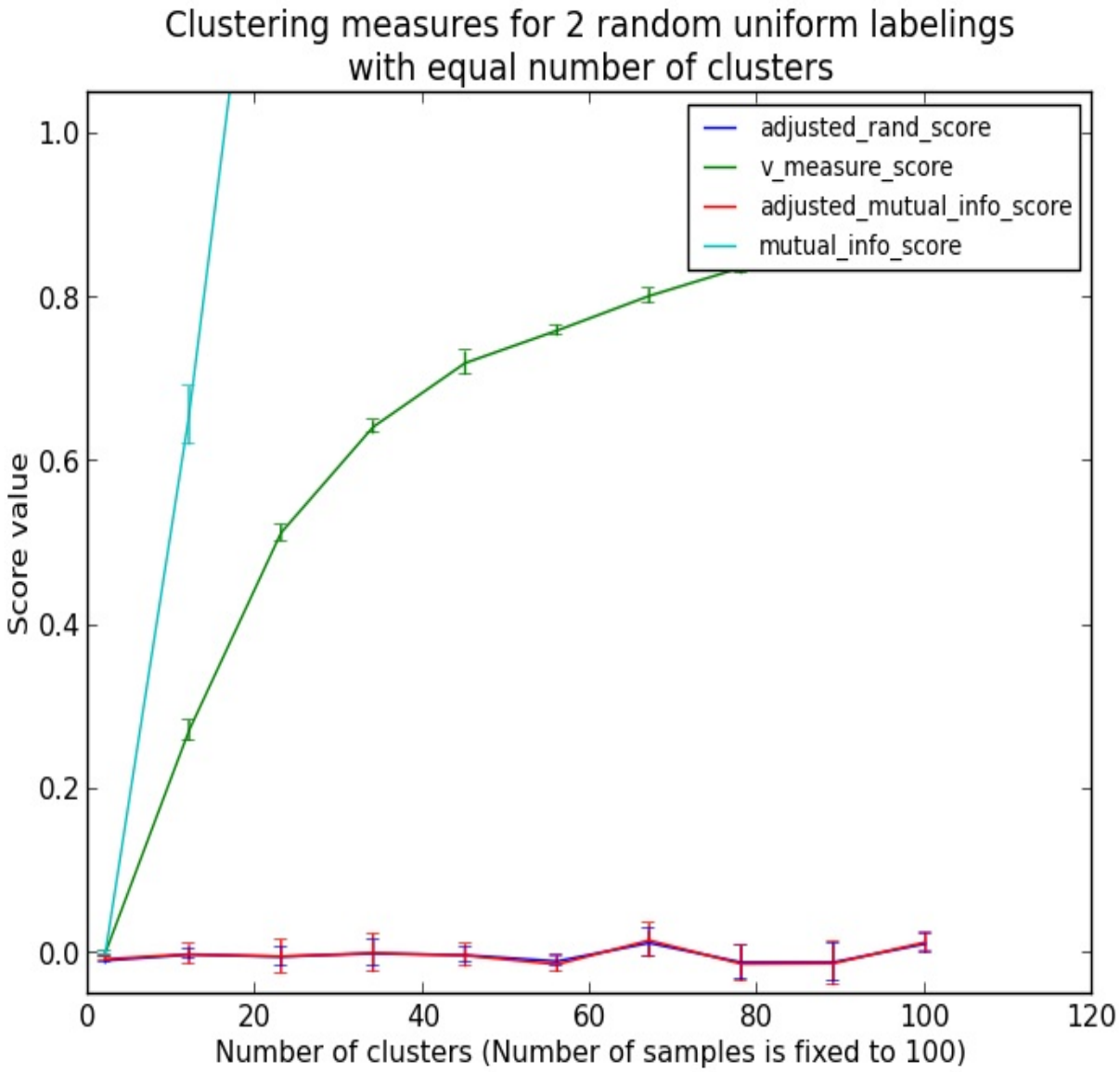
2.3.8.3.2. Advantages

- **Bounded scores**: 0.0 is as bad as it can be, 1.0 is a perfect score
- Intuitive interpretation: clustering with bad V-measure can be **qualitatively analyzed in terms of homogeneity and completeness** to better feel what 'kind' of mistakes is done by the assignment.
- **No assumption is made on the cluster structure**: can be used to compare clustering algorithms such as k-means which assumes isotropic blob shapes with results of spectral clustering algorithms which can find cluster with "folded" shapes.

2.3.8.3.3. Drawbacks

- The previously introduced metrics are **not normalized with regards to random labeling**: this means that depending on the number of samples, clusters and ground truth classes, a completely random labeling will not always yield the same values for homogeneity, completeness and hence v-measure. In particular **random labeling won't yield zero scores especially when the number of clusters is large**.

  This problem can safely be ignored when the number of samples is more than a thousand and the number of clusters is less than 10. **For smaller sample sizes or larger number of clusters it is safer to use an adjusted index such as the Adjusted Rand Index (ARI)**.



- These metrics **require the knowledge of the ground truth classes** while almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

Examples:

- *Adjustment for chance in clustering performance evaluation*: Analysis of the impact of the dataset size on the value of clustering measures for random assignments.

2.3.8.3.4. Mathematical formulation

Homogeneity and completeness scores are formally given by:

$$h = 1 - \frac{H(C|K)}{H(C)}$$

$$c = 1 - \frac{H(K|C)}{H(K)}$$

where $H(C|K)$ is the **conditional entropy of the classes given the cluster assignments** and is given by:

$$H(C|K) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|}\frac{n_{c,k}}{n}\cdot\log\left(\frac{n_{c,k}}{n_k}\right)$$

and $H(C)$ is the **entropy of the classes** and is given by:

$$H(C) = -\sum_{c=1}^{|C|}\frac{n_c}{n}\cdot\log\left(\frac{n_c}{n}\right)$$

with n the total number of samples, n_c and n_k the number of samples respectively belonging to class c and cluster k, and finally $n_{c,k}$ the number of samples from class c assigned to cluster k.

The **conditional entropy of clusters given class** $H(K|C)$ and the **entropy of clusters** $H(K)$ are defined in a symmetric manner.

Rosenberg and Hirschberg further define **V-measure** as the **harmonic mean of homogeneity and completeness**:

$$v = 2\cdot\frac{h\cdot c}{h+c}$$

References

[RH2007] V-Measure: A conditional entropy-based external cluster evaluation measure Andrew Rosenberg and Julia Hirschberg, 2007

[B2011] Identication and Characterization of Events in Social Media, Hila Becker, PhD Thesis.

2.3.8.4. Silhouette Coefficient

2.3.8.4.1. Presentation and usage

If the ground truth labels are not known, evaluation must be performed using the model itself. The Silhouette Coefficient (`sklearn.metrics.silhouette_score`) is an example of such an evaluation, where a higher Silhouette Coefficient score relates to a model with better defined clusters. The Silhouette Coefficient is defined for each sample and is composed of two scores:

- **a**: The mean distance between a sample and all other points in the same class.
- **b**: The mean distance between a sample and all other points in the *next nearest cluster*.

The Silhoeutte Coefficient *s* for a single sample is then given as:

$$s = \frac{b-a}{max(a,b)}$$

The Silhouette Coefficient for a set of samples is given as the mean of the Silhouette Coefficient for each sample.

```
>>> from sklearn import metrics
>>> from sklearn.metrics import pairwise_distances
>>> from sklearn import datasets
>>> dataset = datasets.load_iris()
>>> X = dataset.data
>>> y = dataset.target
```

In normal usage, the Silhouette Coefficient is applied to the results of a cluster analysis.

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> kmeans_model = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans_model.labels_
>>> metrics.silhouette_score(X, labels, metric='euclidean')
...
0.55...
```

References

- Peter J. Rousseeuw (1987). “Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis”. Computational and Applied Mathematics 20: 53–65. doi:10.1016/0377-0427(87)90125-7.

2.3.8.4.2. Advantages

- The score is bounded between -1 for incorrect clustering and +1 for highly dense clustering. Scores around zero indicate overlapping clusters.
- The score is higher when clusters are dense and well separated, which relates to a standard concept of a cluster.

2.3.8.4.3. Drawbacks

- The Silhouette Coefficient is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained through DBSCAN.

Previous Next

2.4. Biclustering

Biclustering can be performed with the module `sklearn.cluster.bicluster`. Biclustering algorithms simultaneously cluster rows and columns of a data matrix. These clusters of rows and columns are known as biclusters. Each determines a submatrix of the original data matrix with some desired properties.

For instance, given a matrix of shape `(10, 10)`, one possible bicluster with three rows and two columns induces a submatrix of shape `(3, 2)`:

```
>>> import numpy as np
>>> data = np.arange(100).reshape(10, 10)
>>> rows = np.array([0, 2, 3])[:, np.newaxis]
>>> columns = np.array([1, 2])
>>> data[rows, columns]
array([[ 1,  2],
       [21, 22],
       [31, 32]])
```

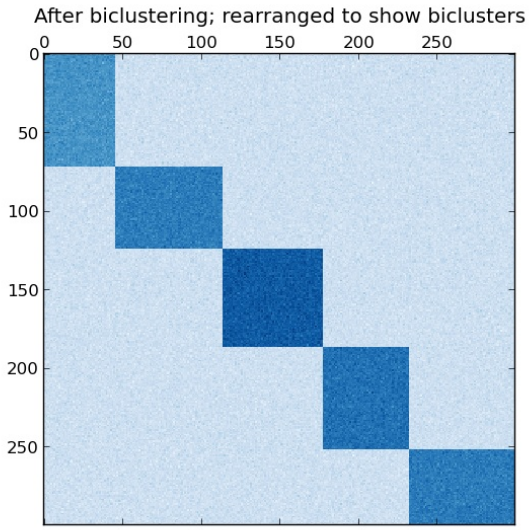
For visualization purposes, given a bicluster, the rows and columns of the data matrix may be rearranged to make the bicluster contiguous.

Algorithms differ in how they define biclusters. Some of the common types include:

- constant values, constant rows, or constant columns
- unusually high or low values
- submatrices with low variance
- correlated rows or columns

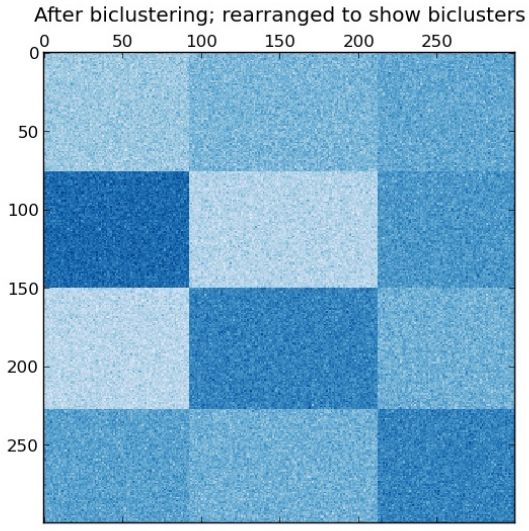
Algorithms also differ in how rows and columns may be assigned to biclusters, which leads to different bicluster structures. Block diagonal or checkerboard structures occur when rows and columns are divided into partitions.

If each row and each column belongs to exactly one bicluster, then rearranging the rows and columns of the data matrix reveals the biclusters on the diagonal. Here is an example of this structure where biclusters have higher average values than the other rows and columns:



An example of biclusters formed by partitioning rows and columns.

In the checkerboard case, each row belongs to all column clusters, and each column belongs to all row clusters. Here is an example of this structure where the variance of the values within each bicluster is small:



An example of checkerboard biclusters.

After fitting a model, row and column cluster membership can be found in the `rows_` and `columns_` attributes. `rows_[i]` is a binary vector with nonzero entries corresponding to rows that belong to bicluster `i`. Similarly, `columns_[i]` indicates which columns belong to bicluster `i`.

Some models also have `row_labels_` and `column_labels_` attributes. These models partition the rows and columns, such as in the block diagonal and checkerboard bicluster structures.

Note: Biclustering has many other names in different fields including co-clustering, two-mode clustering, two-way clustering, block clustering, coupled two-way clustering, etc. The names of some algorithms, such as the Spectral Co-Clustering algorithm, reflect these alternate names.

2.4.1. Spectral Co-Clustering

The `SpectralCoclustering` algorithm finds biclusters with values higher than those in the corresponding other rows and columns. Each row and each column belongs to exactly one bicluster, so rearranging the rows and columns to make partitions contiguous reveals these high values along the diagonal:

Note: The algorithm treats the input data matrix as a bipartite graph: the rows and columns of the matrix correspond to the two sets of vertices, and each entry corresponds to an edge between a row and a column. The algorithm approximates the normalized cut of this graph to find heavy subgraphs.

2.4.1.1. Mathematical formulation

An approximate solution to the optimal normalized cut may be found via the generalized eigenvalue decomposition of the Laplacian of the graph. Usually this would mean working directly with the Laplacian matrix. If the original data matrix A has shape $m \times n$, the Laplacian matrix for the corresponding bipartite graph has shape $(m + n) \times (m + n)$. However, in this case it is possible to work directly with A, which is smaller and more efficient.

The input matrix A is preprocessed as follows:

$$A_n = R^{-1/2} A C^{-1/2}$$

Where R is the diagonal matrix with entry i equal to $\sum_j A_{ij}$ and C is the diagonal matrix with entry j equal to $\sum_i A_{ij}$.

The singular value decomposition, $A_n = U \Sigma V^\top$, provides the partitions of the rows and columns of A. A subset of the left singular vectors gives the row partitions, and a subset of the right singular vectors gives the column partitions.

The $\ell = \lceil \log_2 k \rceil$ singular vectors, starting from the second, provide the desired partitioning information. They are used to form the matrix Z:

$$Z = \begin{bmatrix} R^{-1/2} U \\ \\ C^{-1/2} V \end{bmatrix}$$

where the the columns of U are $u_2, \ldots, u_{\ell+1}$, and similarly for V.

Then the rows of Z are clustered using *k-means*. The first `n_rows` labels provide the row partitioning, and the remaining `n_columns` labels provide the column partitioning.

Examples:

- *A demo of the Spectral Co-Clustering algorithm*: A simple example showing how to generate a data matrix with biclusters and apply this method to it.
- *Biclustering documents with the Spectral Co-clustering algorithm*: An example of finding biclusters in the twenty newsgroup dataset.

References:

- Dhillon, Inderjit S, 2001. Co-clustering documents and words using bipartite spectral graph partitioning.

2.4.2. Spectral Biclustering

The `SpectralBiclustering` algorithm assumes that the input data matrix has a hidden checkerboard structure. The rows and columns of a matrix with this structure may be partitioned so that the entries of any bicluster in the Cartesian product of row clusters and column clusters is are approximately constant. For instance, if there are two row partitions and three column partitions, each row will belong to three biclusters, and each column will belong to two biclusters.

The algorithm partitions the rows and columns of a matrix so that a corresponding blockwise-constant checkerboard matrix provides a good approximation to the original matrix.

2.4.2.1. Mathematical formulation

The input matrix A is first normalized to make the checkerboard pattern more obvious. There are three possible methods:

1. *Independent row and column normalization*, as in Spectral Co-Clustering. This method makes the rows sum to a constant and the columns sum to a different constant.
2. **Bistochastization**: repeated row and column normalization until convergence. This method makes both rows and columns sum to the same constant.
3. **Log normalization**: the log of the data matrix is computed: $L = \log A$. Then the column mean $\overline{L_{i\cdot}}$, row mean $\overline{L_{\cdot j}}$, and overall mean $\overline{L_{\cdot\cdot}}$ of L are computed. The final matrix is computed according to the formula

$$K_{ij} = L_{ij} - \overline{L_{i\cdot}} - \overline{L_{\cdot j}} + \overline{L_{\cdot\cdot}}$$

After normalizing, the first few singular vectors are computed, just as in the Spectral Co-Clustering algorithm.

If log normalization was used, all the singular vectors are meaningful. However, if independent normalization or bistochastization were used, the first singular vectors, u_1 and v_1. are discarded. From now on, the "first" singular vectors refers to $u_2 \dots u_{p+1}$ and $v_2 \dots v_{p+1}$ except in the case of log normalization.

Given these singular vectors, they are ranked according to which can be best approximated by a piecewise-constant vector. The approximations for each vector are found using one-dimensional k-means and scored using the Euclidean distance. Some subset of the best left and right singular vector are selected. Next, the data is projected to this best subset of singular vectors and clustered.

For instance, if p singular vectors were calculated, the q best are found as described, where $q < p$. Let U be the matrix with columns the q best left singular vectors, and similarly V for the right. To partition the rows, the rows of A are projected to a q dimensional space: $A * V$. Treating the m rows of this $m \times q$ matrix as samples and clustering using k-means yields the row labels. Similarly, projecting the columns to $A^\top * U$ and clustering this $n \times q$ matrix yields the column labels.

Examples:

- *A demo of the Spectral Biclustering algorithm*: a simple example showing how to generate a checkerboard matrix and bicluster it.

References:

- Kluger, Yuval, et. al., 2003. Spectral biclustering of microarray data: coclustering genes and conditions.

2.4.3. Biclustering evaluation

There are two ways of evaluating a biclustering result: internal and external. Internal measures, such as cluster stability, rely only on the data and the result themselves. Currently there are no internal bicluster measures in scikit-learn. External measures refer to an external source of information, such as the true solution. When working with real data the true solution is usually unknown, but biclustering artificial data may be useful for evaluating algorithms precisely because the true solution is known.

To compare a set of found biclusters to the set of true biclusters, two similarity measures are needed: a similarity measure for individual biclusters, and a way to combine these individual similarities into an overall score.

To compare individual biclusters, several measures have been used. For now, only the Jaccard index is implemented:

$$J(A, B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

where A and B are biclusters, $|A \cap B|$ is the number of elements in their intersection. The Jaccard index achieves its minimum of 0 when the biclusters to not overlap at all and its maximum of 1 when they are identical.

Several methods have been developed to compare two sets of biclusters. For now, only `consensus_score` (Hochreiter et. al., 2010) is available:

1. Compute bicluster similarities for pairs of biclusters, one in each set, using the Jaccard index or a similar measure.
2. Assign biclusters from one set to another in a one-to-one fashion to maximize the sum of their similarities. This step is performed using the Hungarian algorithm.
3. The final sum of similarities is divided by the size of the larger set.

The minimum consensus score, 0, occurs when all pairs of biclusters are totally dissimilar. The maximum score, 1, occurs when both sets are identical.

References:

- Hochreiter, Bodenhofer, et. al., 2010. FABIA: factor analysis for bicluster acquisition.

2.5. Decomposing signals in components (matrix factorization problems)

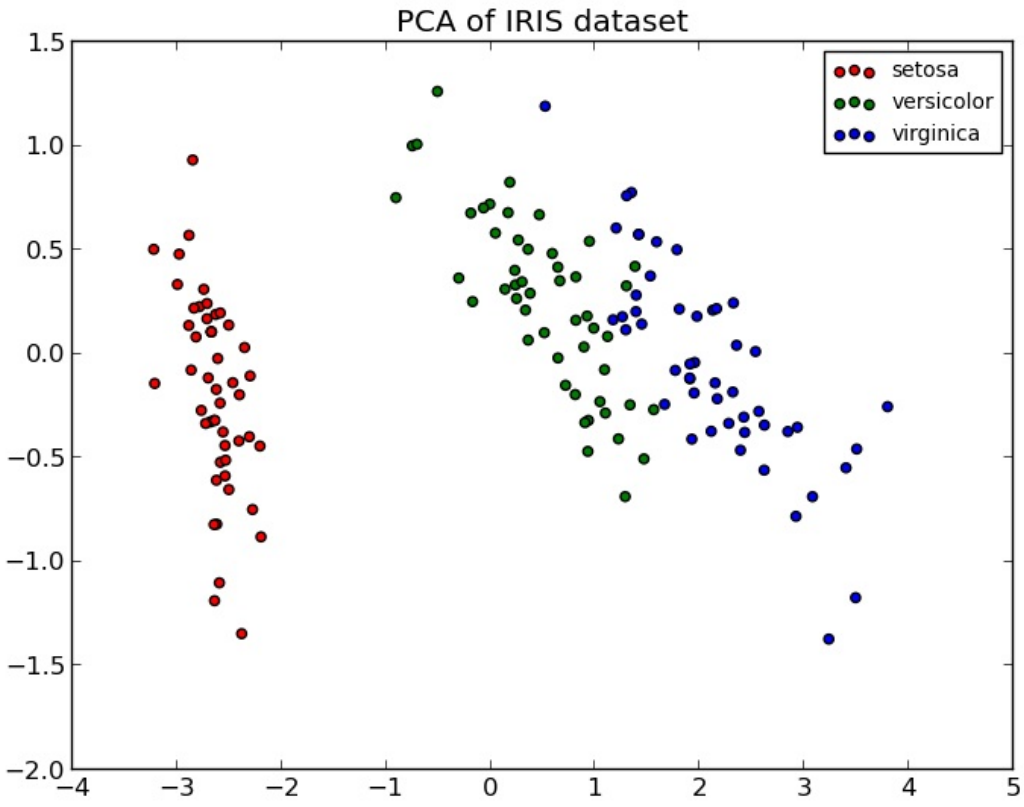
2.5.1. Principal component analysis (PCA)

2.5.1.1. 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 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.

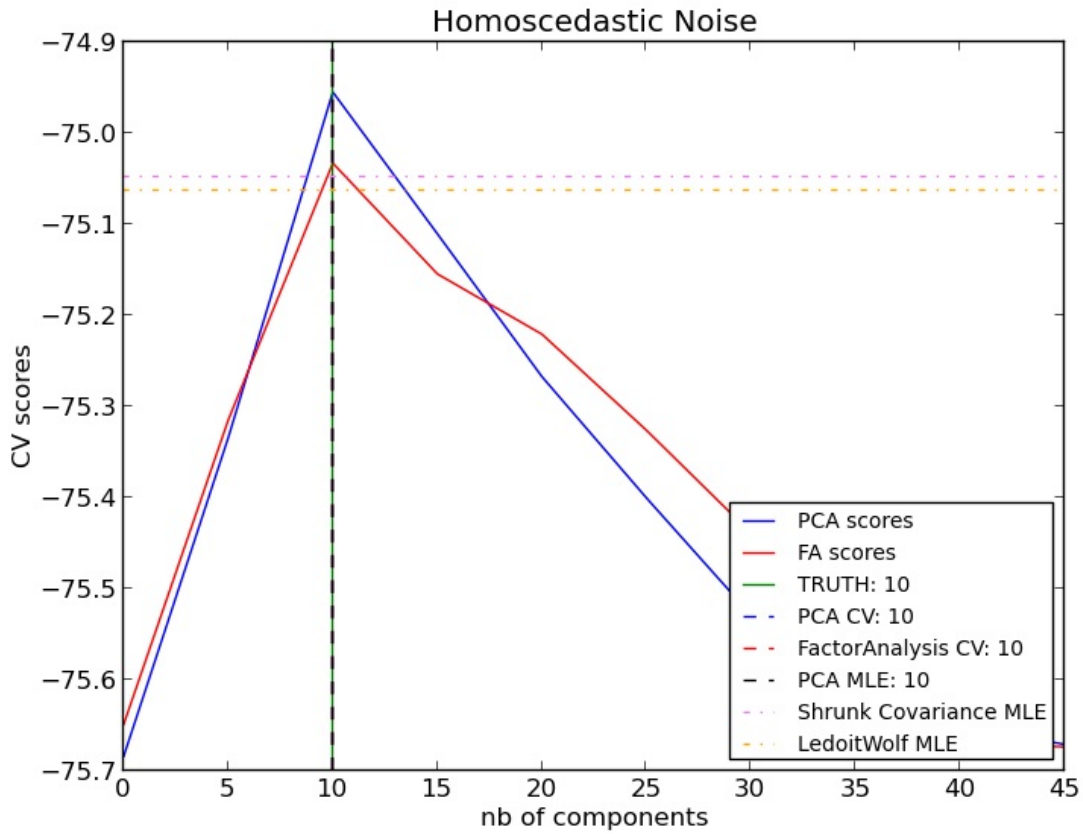
The optional parameter `whiten=True` parameter make it possible to project the data onto the singular space while scaling each component to unit variance. This is often useful if the models down-stream make strong assumptions on the isotropy of the signal: this is for example the case for Support Vector 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.

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



The `PCA` object also 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:



Examples:

- *Comparison of LDA and PCA 2D projection of Iris dataset*
- *Model selection with Probabilistic (PCA) and Factor Analysis (FA)*

2.5.1.2. Approximate PCA

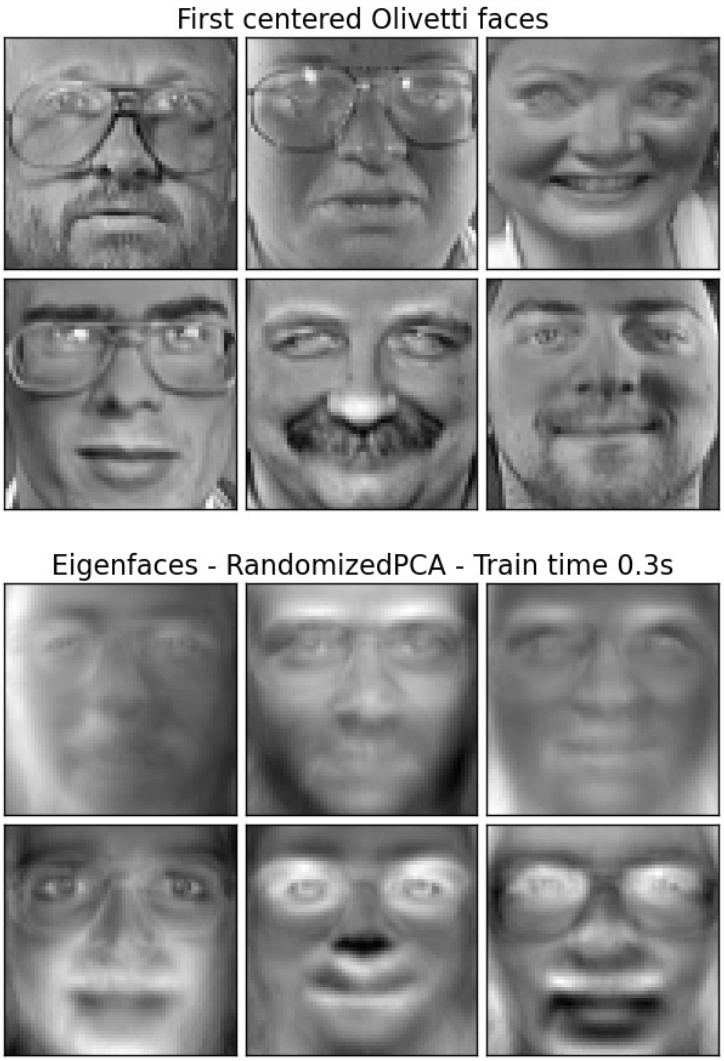
It is often interesting to project data to a lower-dimensional space that preserves most of the variance, by dropping the singular vector of components associated with lower singular values.

For instance, if we work with 64x64 pixel gray-level pictures for face recognition, the dimensionality of the data is 4096 and it is slow to train an RBF support vector machine on such wide data. Furthermore we know that the intrinsic dimensionality of the data is much lower than 4096 since all pictures of human faces look somewhat 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.

For instance, the following shows 16 sample portraits (centered around 0.0) from the Olivetti dataset. On the right hand side are the first 16 singular vectors reshaped as portraits. Since we only require the top 16 singular vectors of a dataset with size $n_{samples} = 400$ and $n_{features} = 64 \times 64 = 4096$, the

computation time it less than 1s:



RandomizedPCA can hence be used as a drop in replacement for **PCA** with the exception that we need to give it the size of the lower-dimensional space `n_components` as a 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:

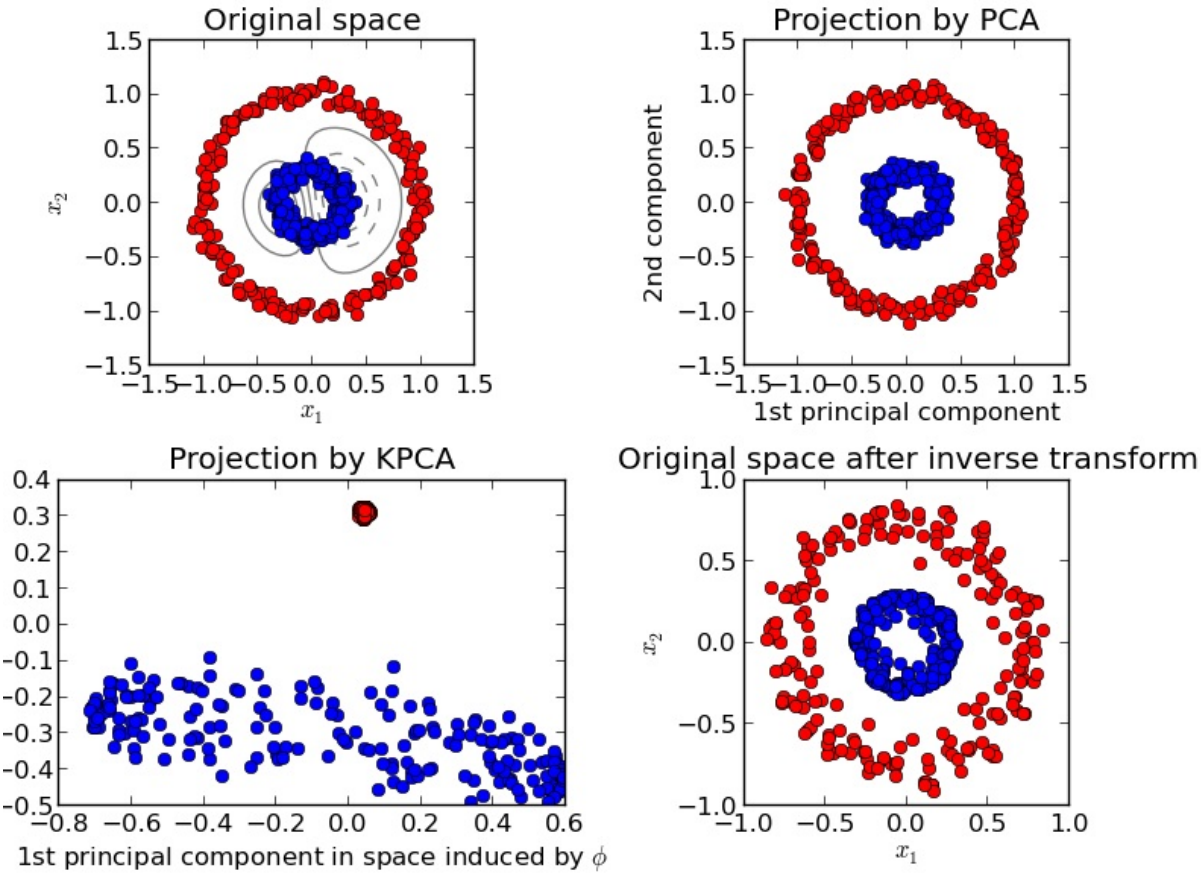
- *Faces recognition example using eigenfaces and SVMs*
- *Faces dataset decompositions*

References:

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

2.5.1.3. Kernel PCA

KernelPCA is an extension of PCA which achieves non-linear dimensionality reduction through the use of kernels. It has many applications including denoising, compression and structured prediction (kernel dependency estimation). **KernelPCA** supports both `transform` and `inverse_transform`.



Examples:

- *Kernel PCA*

2.5.1.4. Sparse principal components analysis (SparsePCA and MiniBatchSparsePCA)

SparsePCA is a variant of PCA, with the goal of extracting the set of sparse components that best reconstruct the data.

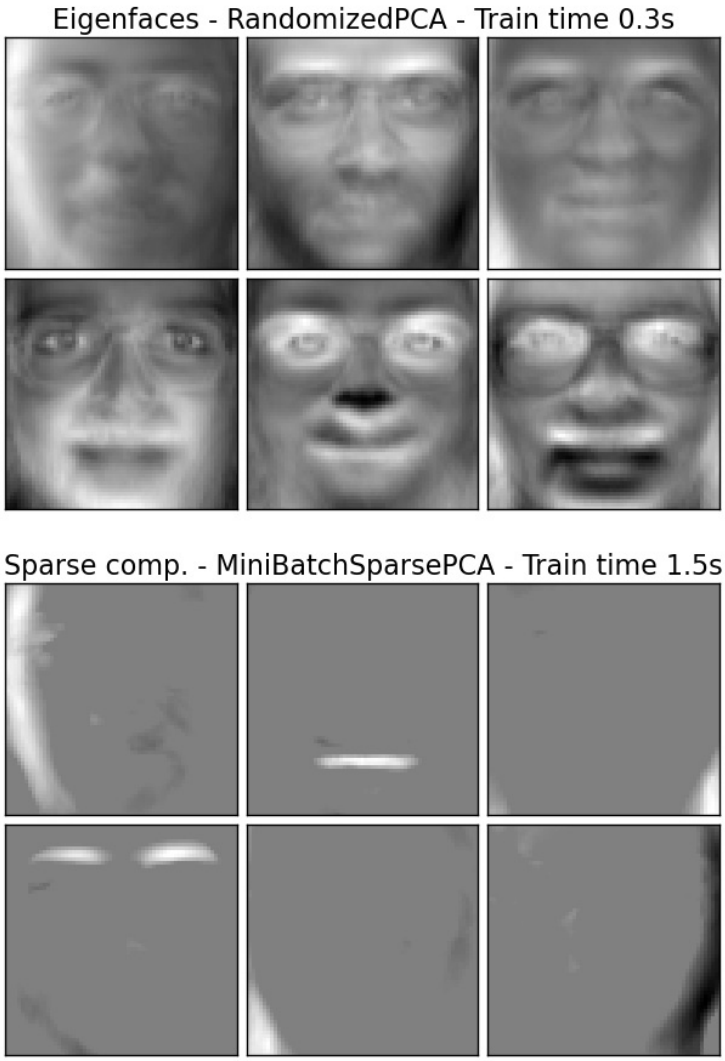
Mini-batch sparse PCA (**MiniBatchSparsePCA**) is a variant of **SparsePCA** that is faster but less accurate. The increased speed is reached by iterating over small chunks of the set of features, for a given number of iterations.

Principal component analysis (**PCA**) has the disadvantage that the components extracted by this method

have exclusively dense expressions, i.e. they have non-zero coefficients when expressed as linear combinations of the original variables. This can make interpretation difficult. In many cases, the real underlying components can be more naturally imagined as sparse vectors; for example in face recognition, components might naturally map to parts of faces.

Sparse principal components yields a more parsimonious, interpretable representation, clearly emphasizing which of the original features contribute to the differences between samples.

The following example illustrates 16 components extracted using sparse PCA from the Olivetti faces dataset. It can be seen how the regularization term induces many zeros. Furthermore, the natural structure of the data causes the non-zero coefficients to be vertically adjacent. The model does not enforce this mathematically: each component is a vector $h \in \mathbf{R}^{4096}$, and there is no notion of vertical adjacency except during the human-friendly visualization as 64x64 pixel images. The fact that the components shown below appear local is the effect of the inherent structure of the data, which makes such local patterns minimize reconstruction error. There exist sparsity-inducing norms that take into account adjacency and different kinds of structure; see [Jen09] for a review of such methods. For more details on how to use Sparse PCA, see the Examples section, below.



Note that there are many different formulations for the Sparse PCA problem. The one implemented here is based on [Mrl09] . The optimization problem solved is a PCA problem (dictionary learning) with an ℓ_1 penalty on the components:

$$(U^*, V^*) = \arg\min_{U,V} \frac{1}{2}||X - UV||_2^2 + \alpha||V||_1$$
$$\text{subject to } ||U_k||_2 = 1 \text{ for all } 0 \leq k < n_{components}$$

The sparsity-inducing ℓ_1 norm also prevents learning components from noise when few training samples are available. The degree of penalization (and thus sparsity) can be adjusted through the hyperparameter `alpha`. Small values lead to a gently regularized factorization, while larger values shrink many coefficients to zero.

Note: While in the spirit of an online algorithm, the class **`MiniBatchSparsePCA`** does not implement `partial_fit` because the algorithm is online along the features direction, not the samples direction.

Examples:

- *Faces dataset decompositions*

References:

[Mrl09] "Online Dictionary Learning for Sparse Coding" J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009
[Jen09] "Structured Sparse Principal Component Analysis" R. Jenatton, G. Obozinski, F. Bach, 2009

2.5.2. Truncated singular value decomposition and latent semantic analysis

`TruncatedSVD` implements a variant of singular value decomposition (SVD) that only computes the k largest singular values, where k is a user-specified parameter.

When truncated SVD is applied to term-document matrices (as returned by `CountVectorizer` or `TfidfVectorizer`), this transformation is known as latent semantic analysis (LSA), because it transforms such matrices to a "semantic" space of low dimensionality. In particular, LSA is known to combat the effects of synonymy and polysemy (both of which roughly mean there are multiple meanings per word), which cause term-document matrices to be overly sparse and exhibit poor similarity under measures such as cosine similarity.

Note: LSA is also known as latent semantic indexing, LSI, though strictly that refers to its use in persistent indexes for information retrieval purposes.

Mathematically, truncated SVD applied to training samples X produces a low-rank approximation X:

$$X \approx X_k = U_k \Sigma_k V_k^\top$$

After this operation, $U_k \Sigma_k^\top$ is the transformed training set with k features (called `n_components` in the API).

To also transform a test set X, we multiply it with V_k:

$$X' = X V_k^\top$$

Note: Most treatments of LSA in the natural language processing (NLP) and information retrieval (IR) literature swap the axis of the matrix X so that it has shape `n_features` × `n_samples`. We present LSA

in a different way that matches the scikit-learn API better, but the singular values found are the same.

`TruncatedSVD` is very similar to `PCA`, but differs in that it works on sample matrices X directly instead of their covariance matrices. When the columnwise (per-feature) means of X are subtracted from the feature values, truncated SVD on the resulting matrix is equivalent to PCA. In practical terms, this means that the `TruncatedSVD` transformer accepts `scipy.sparse` matrices without the need to densify them, as densifying may fill up memory even for medium-sized document collections.

While the `TruncatedSVD` transformer works with any (sparse) feature matrix, using it on tf–idf matrices is recommended over raw frequency counts in an LSA/document processing setting. In particular, sublinear scaling and inverse document frequency should be turned on (`sublinear_tf=True, use_idf=True`) to bring the feature values closer to a Gaussian distribution, compensating for LSA's erroneous assumptions about textual data.

Examples:

- *Clustering text documents using k-means*

References:

- Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze (2008), *Introduction to Information Retrieval*, Cambridge University Press, chapter 18: Matrix decompositions & latent semantic indexing

2.5.3. Dictionary Learning

2.5.3.1. Sparse coding with a precomputed dictionary

The `SparseCoder` object is an estimator that can be used to transform signals into sparse linear combination of atoms from a fixed, precomputed dictionary such as a discrete wavelet basis. This object therefore does not implement a `fit` method. The transformation amounts to a sparse coding problem: finding a representation of the data as a linear combination of as few dictionary atoms as possible. All variations of dictionary learning implement the following transform methods, controllable via the `transform_method` initialization parameter:

- Orthogonal matching pursuit (*Orthogonal Matching Pursuit (OMP)*)
- Least-angle regression (*Least Angle Regression*)
- Lasso computed by least-angle regression
- Lasso using coordinate descent (*Lasso*)
- Thresholding

Thresholding is very fast but it does not yield accurate reconstructions. They have been shown useful in literature for classification tasks. For image reconstruction tasks, orthogonal matching pursuit yields the most accurate, unbiased reconstruction.

The dictionary learning objects offer, via the `split_code` parameter, the possibility to separate the positive and negative values in the results of sparse coding. This is useful when dictionary learning is used for

extracting features that will be used for supervised learning, because it allows the learning algorithm to assign different weights to negative loadings of a particular atom, from to the corresponding positive loading.

The split code for a single sample has length `2 * n_components` and is constructed using the following rule: First, the regular code of length `n_components` is computed. Then, the first `n_components` entries of the `split_code` are filled with the positive part of the regular code vector. The second half of the split code is filled with the negative part of the code vector, only with a positive sign. Therefore, the split_code is non-negative.

Examples:

- *Sparse coding with a precomputed dictionary*

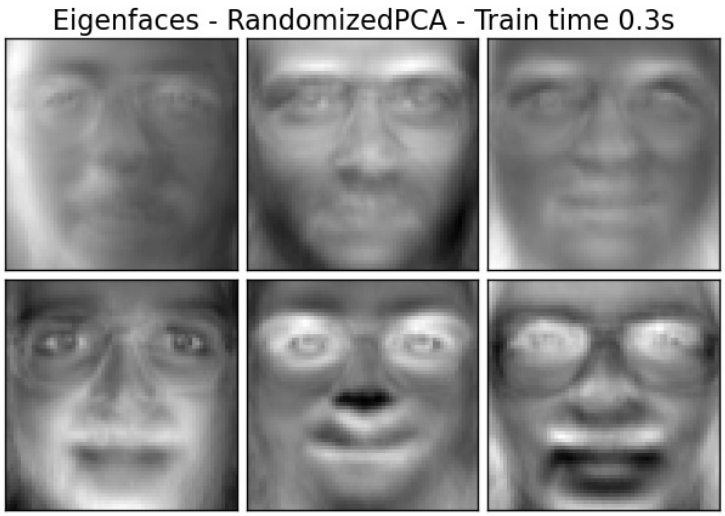
2.5.3.2. Generic dictionary learning

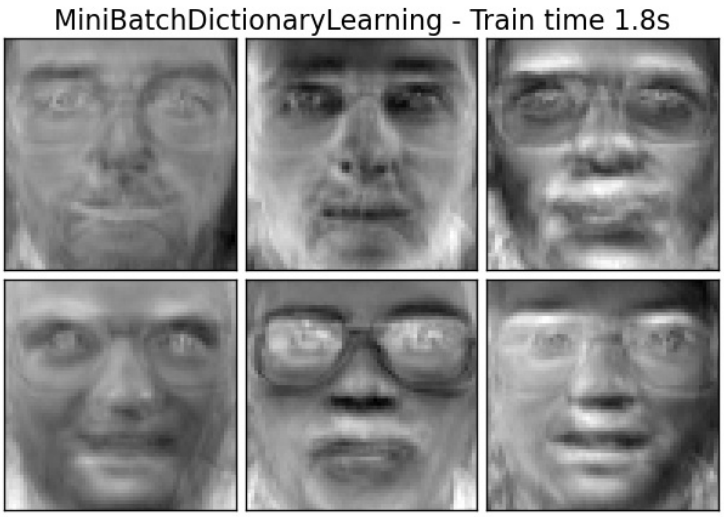
Dictionary learning (**`DictionaryLearning`**) is a matrix factorization problem that amounts to finding a (usually overcomplete) dictionary that will perform good at sparsely encoding the fitted data.

Representing data as sparse combinations of atoms from an overcomplete dictionary is suggested to be the way the mammal primary visual cortex works. Consequently, dictionary learning applied on image patches has been shown to give good results in image processing tasks such as image completion, inpainting and denoising, as well as for supervised recognition tasks.

Dictionary learning is an optimization problem solved by alternatively updating the sparse code, as a solution to multiple Lasso problems, considering the dictionary fixed, and then updating the dictionary to best fit the sparse code.

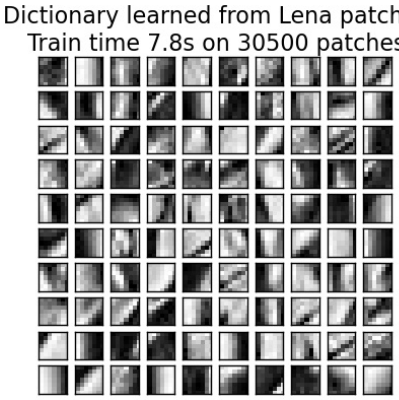
$$(U^*, V^*) = \arg\min_{U,V} \frac{1}{2}||X - UV||_2^2 + \alpha||U||_1$$
$$\text{subject to } ||V_k||_2 = 1 \text{ for all } 0 \leq k < n_{atoms}$$





After using such a procedure to fit the dictionary, the transform is simply a sparse coding step that shares the same implementation with all dictionary learning objects (see *Sparse coding with a precomputed dictionary*).

The following image shows how a dictionary learned from 4x4 pixel image patches extracted from part of the image of Lena looks like.



Examples:

- *Image denoising using dictionary learning*

References:

- "Online dictionary learning for sparse coding" J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009

2.5.3.3. Mini-batch dictionary learning

`MiniBatchDictionaryLearning` implements a faster, but less accurate version of the dictionary learning algorithm that is better suited for large datasets.

By default, `MiniBatchDictionaryLearning` divides the data into mini-batches and optimizes in an online manner by cycling over the mini-batches for the specified number of iterations. However, at the moment it does not implement a stopping condition.

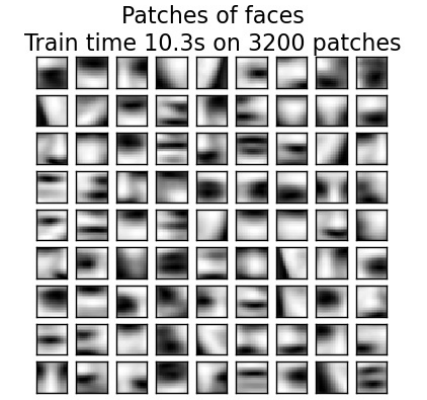
The estimator also implements `partial_fit`, which updates the dictionary by iterating only once over a mini-batch. This can be used for online learning when the data is not readily available from the start, or for

when the data does not fit into the memory.

Clustering for dictionary learning

Note that when using dictionary learning to extract a representation (e.g. for sparse coding) clustering can be a good proxy to learn the dictionary. For instance the `MiniBatchKMeans` estimator is computationally efficient and implements on-line learning with a `partial_fit` method.

Example: *Online learning of a dictionary of parts of faces*



2.5.4. Factor Analysis

In unsupervised learning we only have a dataset $X = \{x_1, x_2, \ldots, x_n\}$. How can this dataset be described mathematically? A very simple *continuous latent variabel* model for X is

$$x_i = W h_i + \mu + \epsilon$$

The vector h_i is called "latent" because it is unobserved. ϵ is considered a noise term distributed according to a Gaussian with mean 0 and covariance Ψ (i.e. $\epsilon \sim \mathcal{N}(0, \Psi)$), μ is some arbitrary offset vector. Such a model is called "generative" as it describes how x_i is generated from h_i. If we use all the x_i's as columns to form a matrix $\mathbf{X}$ and all the h_i's as columns of a matrix $\mathbf{H}$ then we can write (with suitably defined $\mathbf{M}$ and $\mathbf{E}$):

$$\mathbf{X} = W\mathbf{H} + \mathbf{M} + \mathbf{E}$$

In other words, we *decomposed* matrix $\mathbf{X}$.

If h_i is given, the above equation automatically implies the following probabilistic interpretation:

$$p(x_i|h_i) = \mathcal{N}(Wh_i + \mu, \Psi)$$

For a complete probabilistic model we also need a prior distribution for the latent variable h. The most straightforward assumption (based on the nice properties of the Gaussian distribution) is $h \sim \mathcal{N}(0, \mathbf{I})$. This yields a Gaussian as the marginal distribution of x:

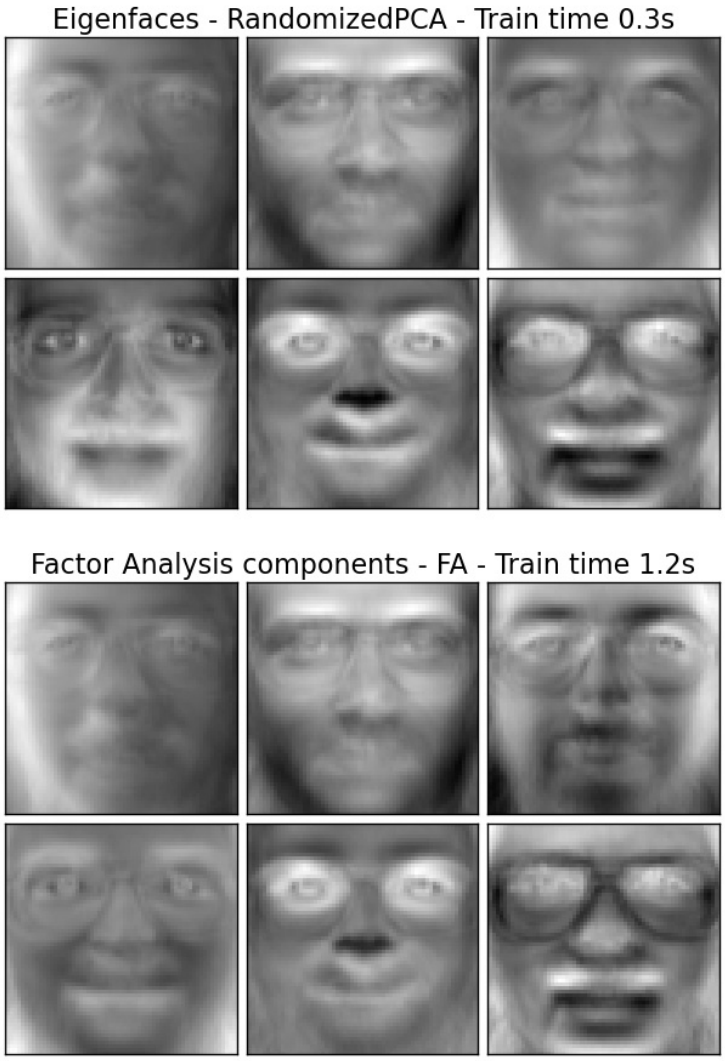
$$p(x) = \mathcal{N}(\mu, WW^T + \Psi)$$

Now, without any further assumptions the idea of having a latent variable h would be superfluous – x can be completely modelled with a mean and a covariance. We need to impose some more specific structure on one of these two parameters. A simple additional assumption regards the structure of the error covariance Ψ:

- $\Psi = \sigma^2 \mathbf{I}$: This assumption leads to the probabilistic model of `PCA`.
- $\Psi = diag(\psi_1, \psi_2, \ldots, \psi_n)$: This model is called Factor Analysis, a classical statistical model. The matrix W is sometimes called the "factor loading matrix".

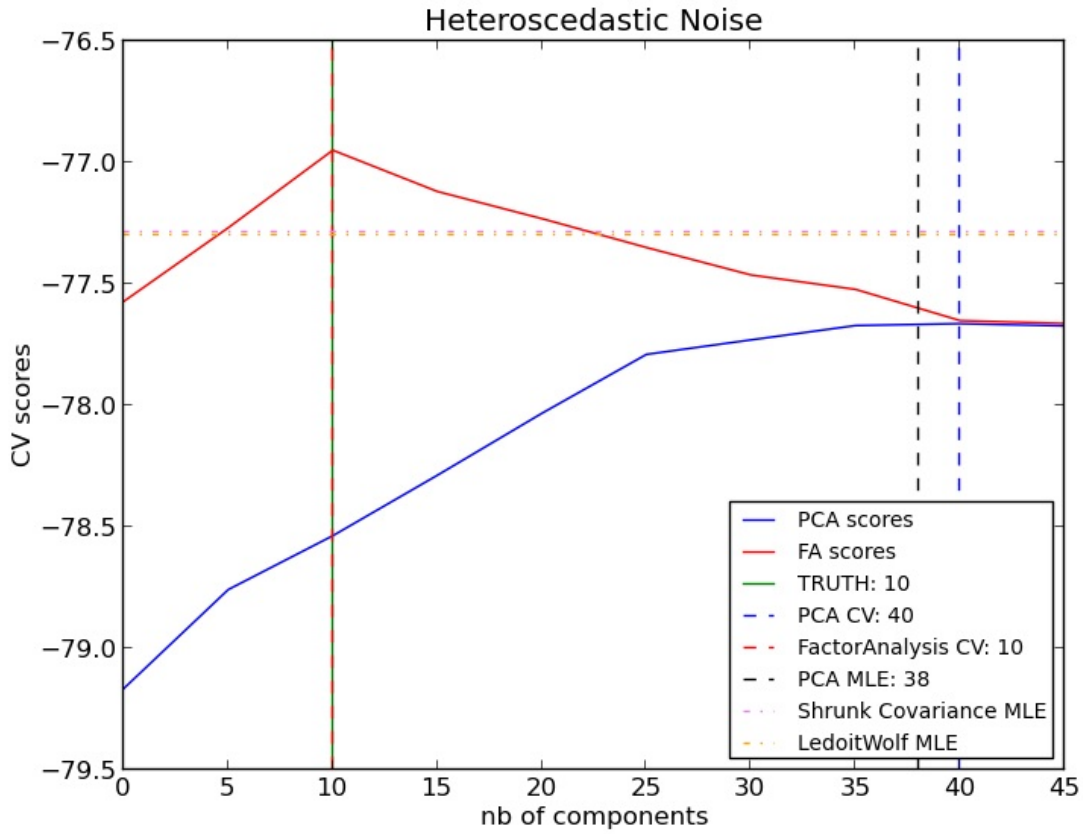
Both model essentially estimate a Gaussian with a low-rank covariance matrix. Because both models are probabilistic they can be integrated in more complex models, e.g. Mixture of Factor Analysers. One gets very different models (e.g. `FastICA`) if non-Gaussian priors on the latent variables are assumed.

Factor analysis *can* produce similar components (the columns of its loading matrix) to `PCA`. However, one can not make any general statements about these components (e.g. whether they are orthogonal):



The main advantage for Factor Analysis (over `PCA` is that it can model the variance in every direction of the input space independently (heteroscedastic noise):

This allows better model selection than probabilistic PCA in the presence of heteroscedastic noise:



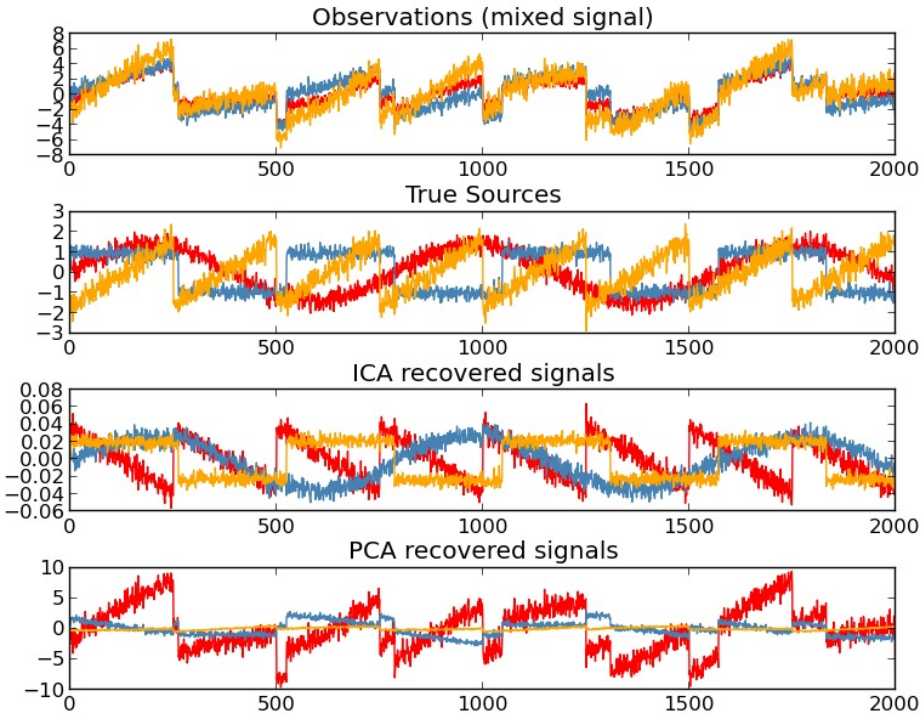
Examples:

- *Model selection with Probabilistic (PCA) and Factor Analysis (FA)*

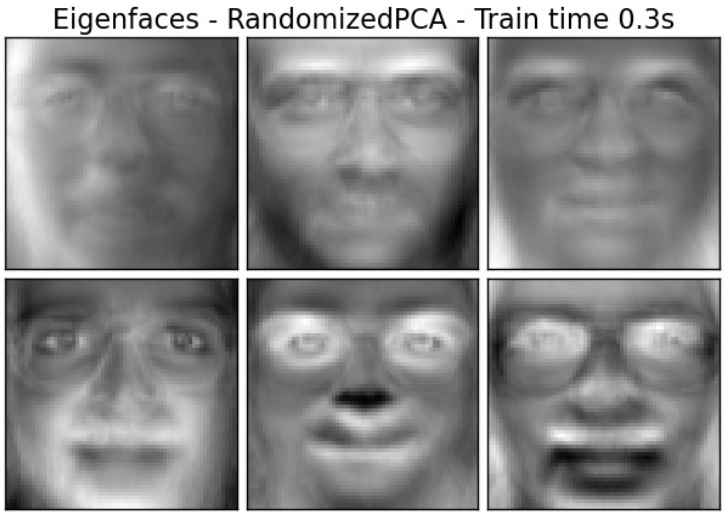
2.5.5. Independent component analysis (ICA)

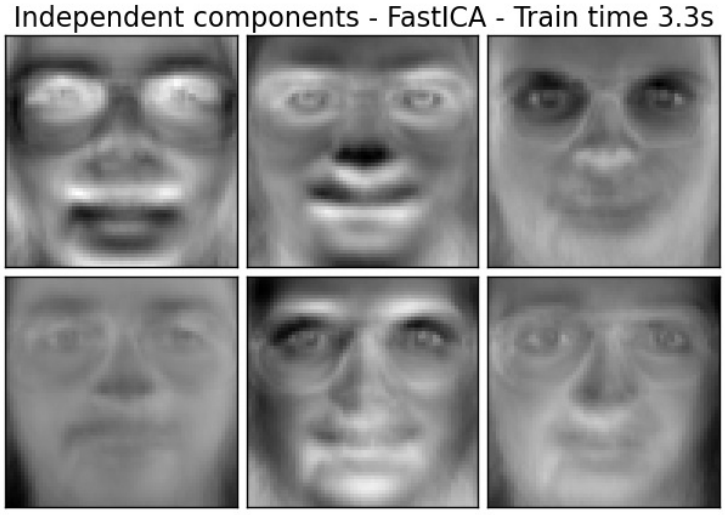
Independent component analysis separates a multivariate signal into additive subcomponents that are maximally independent. It is implemented in scikit-learn using the `Fast ICA` algorithm. Typically, ICA is not used for reducing dimensionality but for separating superimposed signals. Since the ICA model does not include a noise term, for the model to be correct, whitening must be applied. This can be done internally using the whiten argument or manually using one of the PCA variants.

It is classically used to separate mixed signals (a problem known as *blind source separation*), as in the example below:



ICA can also be used as yet another non linear decomposition that finds components with some sparsity:





Examples:

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

2.5.6. Non-negative matrix factorization (NMF or NNMF)

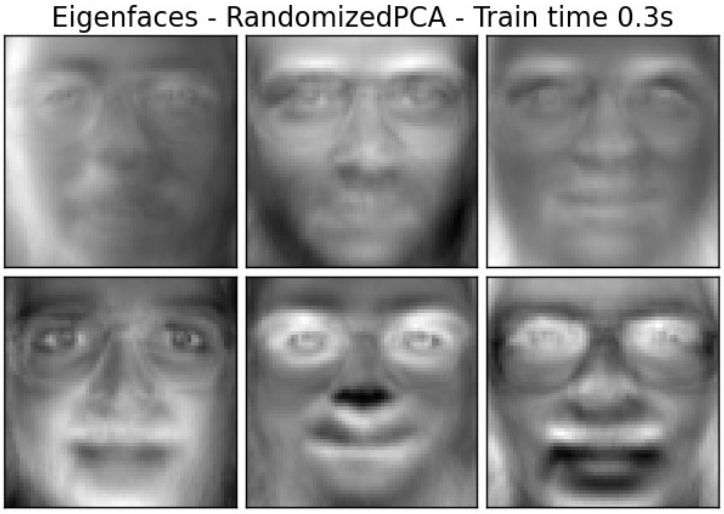
NMF is an alternative approach to decomposition that assumes that the data and the components are non-negative. **NMF** can be plugged in instead of **PCA** or its variants, in the cases where the data matrix does not contain negative values. It finds a decomposition of samples X into two matrices V and H of non-negative elements, by optimizing for the squared Frobenius norm:

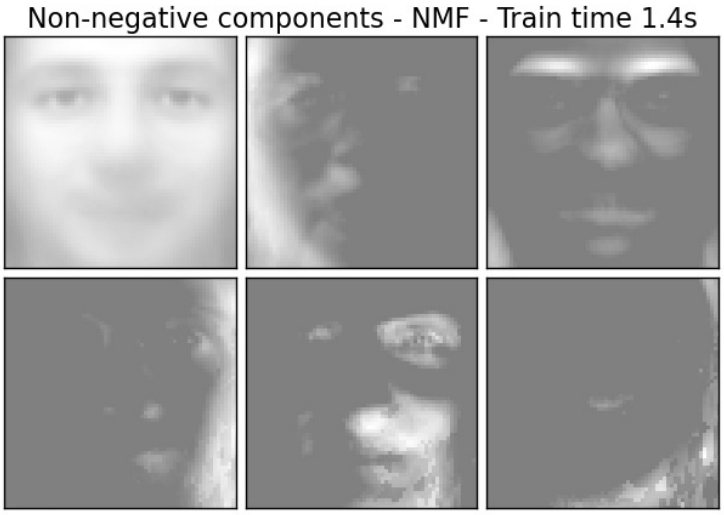
$$\arg\min_{W,H} ||X - WH||^2 = \sum_{i,j} X_{ij} - WH_{ij}$$

This norm is an obvious extension of the Euclidean norm to matrices. (Other optimization objectives have been suggested in the NMF literature, in particular Kullback-Leibler divergence, but these are not currently implemented.)

Unlike **PCA**, the representation of a vector is obtained in an additive fashion, by superimposing the components, without subtracting. Such additive models are efficient for representing images and text.

It has been observed in [Hoyer, 04] that, when carefully constrained, **NMF** can produce a parts-based representation of the dataset, resulting in interpretable models. The following example displays 16 sparse components found by **NMF** from the images in the Olivetti faces dataset, in comparison with the PCA eigenfaces.





The **`init`** attribute determines the initialization method applied, which has a great impact on the performance of the method. **`NMF`** implements the method Nonnegative Double Singular Value Decomposition. NNDSVD is based on two SVD processes, one approximating the data matrix, the other approximating positive sections of the resulting partial SVD factors utilizing an algebraic property of unit rank matrices. The basic NNDSVD algorithm is better fit for sparse factorization. Its variants NNDSVDa (in which all zeros are set equal to the mean of all elements of the data), and NNDSVDar (in which the zeros are set to random perturbations less than the mean of the data divided by 100) are recommended in the dense case.

`NMF` can also be initialized with random non-negative matrices, by passing an integer seed or a `RandomState` to **`init`**.

In **`NMF`**, sparseness can be enforced by setting the attribute **`sparseness`** to `"data"` or `"components"`. Sparse components lead to localized features, and sparse data leads to a more efficient representation of the data.

Examples:

- *Faces dataset decompositions*
- *Topics extraction with Non-Negative Matrix Factorization*

References:

- “Learning the parts of objects by non-negative matrix factorization” D. Lee, S. Seung, 1999
- “Non-negative Matrix Factorization with Sparseness Constraints” P. Hoyer, 2004
- “Projected gradient methods for non-negative matrix factorization” C.-J. Lin, 2007
- “SVD based initialization: A head start for nonnegative matrix factorization” C. Boutsidis, E. Gallopoulos, 2008

2.6. Covariance estimation

Many statistical problems require at some point the estimation of a population's covariance matrix, which can be seen as an estimation of data set scatter plot shape. Most of the time, such an estimation has to be done on a sample whose properties (size, structure, homogeneity) has a large influence on the estimation's quality. The *sklearn.covariance* package aims at providing tools affording an accurate estimation of a population's covariance matrix under various settings.

We assume that the observations are independent and identically distributed (i.i.d.).

2.6.1. Empirical covariance

The covariance matrix of a data set is known to be well approximated with the classical *maximum likelihood estimator* (or "empirical covariance"), provided the number of observations is large enough compared to the number of features (the variables describing the observations). More precisely, the Maximum Likelihood Estimator of a sample is an unbiased estimator of the corresponding population covariance matrix.

The empirical covariance matrix of a sample can be computed using the `empirical_covariance` function of the package, or by fitting an `EmpiricalCovariance` object to the data sample with the `EmpiricalCovariance.fit` method. Be careful that depending whether the data are centered or not, the result will be different, so one may want to use the `assume_centered` parameter accurately. More precisely if one uses `assume_centered=False`, then the test set is supposed to have the same mean vector as the training set. If not so, both should be centered by the user, and `assume_centered=True` should be used.

Examples:

- See *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood* for an example on how to fit an `EmpiricalCovariance` object to data.

2.6.2. Shrunk Covariance

2.6.2.1. Basic shrinkage

Despite being an unbiased estimator of the covariance matrix, the Maximum Likelihood Estimator is not a good estimator of the eigenvalues of the covariance matrix, so the precision matrix obtained from its inversion is not accurate. Sometimes, it even occurs that the empirical covariance matrix cannot be inverted for numerical reasons. To avoid such an inversion problem, a transformation of the empirical covariance matrix has been introduced: the `shrinkage`.

In the scikit-learn, this transformation (with a user-defined shrinkage coefficient) can be directly applied to a pre-computed covariance with the `shrunk_covariance` method. Also, a shrunk estimator of the covariance can be fitted to data with a `ShrunkCovariance` object and its `ShrunkCovariance.fit` method. Again,

depending whether the data are centered or not, the result will be different, so one may want to use the `assume_centered` parameter accurately.

Mathematically, this shrinkage consists in reducing the ratio between the smallest and the largest eigenvalue of the empirical covariance matrix. It can be done by simply shifting every eigenvalue according to a given offset, which is equivalent of finding the l2-penalized Maximum Likelihood Estimator of the covariance matrix. In practice, shrinkage boils down to a simple a convex transformation : $\Sigma_{\text{shrunk}} = (1-\alpha)\hat{\Sigma} + \alpha\frac{\text{Tr}\hat{\Sigma}}{p}\text{Id}$.

Choosing the amount of shrinkage, α amounts to setting a bias/variance trade-off, and is discussed below.

Examples:

- See *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood* for an example on how to fit a **`ShrunkCovariance`** object to data.

2.6.2.2. Ledoit-Wolf shrinkage

In their 2004 paper [1], O. Ledoit and M. Wolf propose a formula so as to compute the optimal shrinkage coefficient α that minimizes the Mean Squared Error between the estimated and the real covariance matrix.

The Ledoit-Wolf estimator of the covariance matrix can be computed on a sample with the **`ledoit_wolf`** function of the *sklearn.covariance* package, or it can be otherwise obtained by fitting a **`LedoitWolf`** object to the same sample.

Examples:

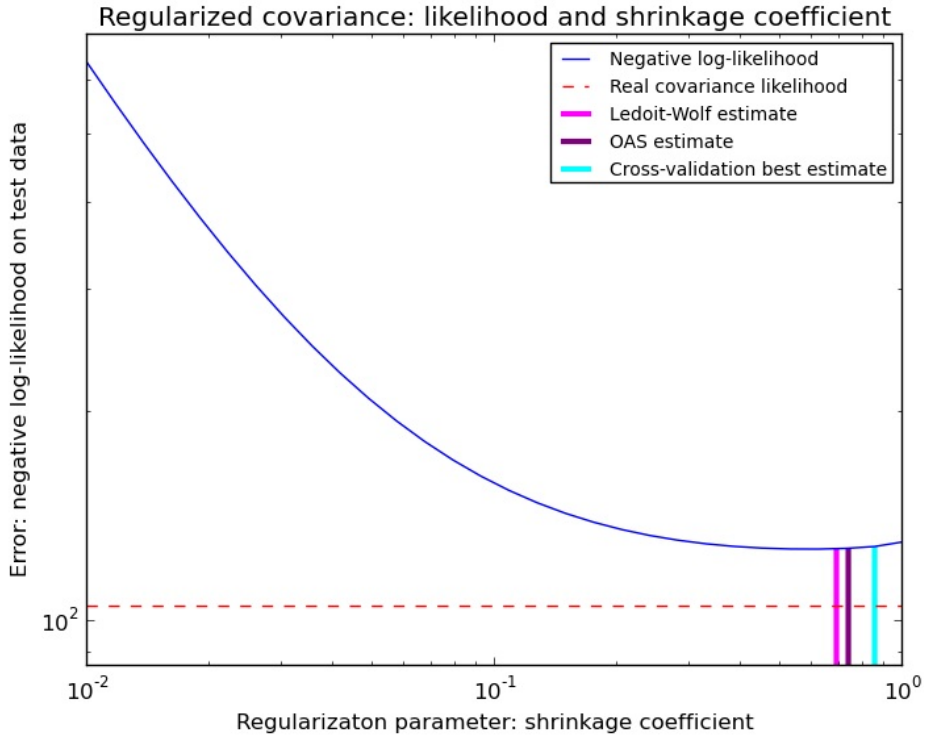
- See *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood* for an example on how to fit a **`LedoitWolf`** object to data and for visualizing the performances of the Ledoit-Wolf estimator in terms of likelihood.

[1] O. Ledoit and M. Wolf, “A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices”, Journal of Multivariate Analysis, Volume 88, Issue 2, February 2004, pages 365-411.

2.6.2.3. Oracle Approximating Shrinkage

Under the assumption that the data are Gaussian distributed, Chen et al. [2] derived a formula aimed at choosing a shrinkage coefficient that yields a smaller Mean Squared Error than the one given by Ledoit and Wolf’s formula. The resulting estimator is known as the Oracle Shrinkage Approximating estimator of the covariance.

The OAS estimator of the covariance matrix can be computed on a sample with the **`oas`** function of the *sklearn.covariance* package, or it can be otherwise obtained by fitting an **`OAS`** object to the same sample.

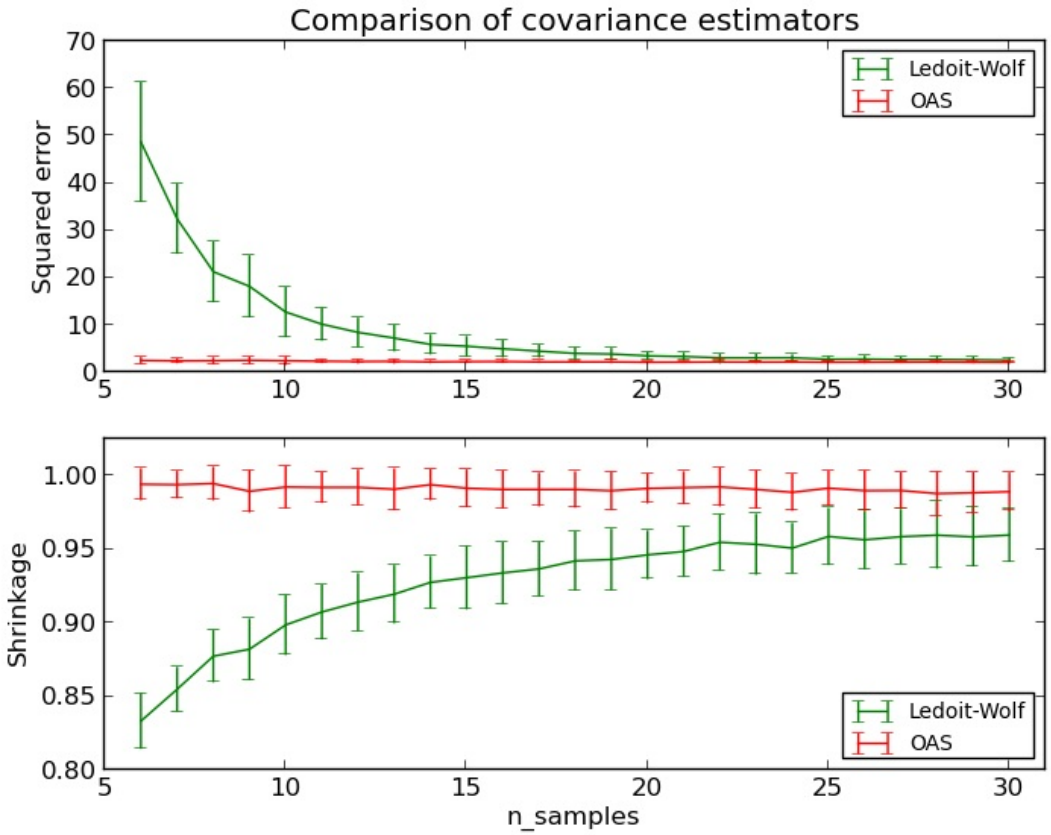


Bias-variance trade-off when setting the shrinkage: comparing the choices of Ledoit-Wolf and OAS estimators

[2] Chen et al., “Shrinkage Algorithms for MMSE Covariance Estimation”,
IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

Examples:

- See *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood* for an example on how to fit an `OAS` object to data.
- See *Ledoit-Wolf vs OAS estimation* to visualize the Mean Squared Error difference between a `LedoitWolf` and an `OAS` estimator of the covariance.

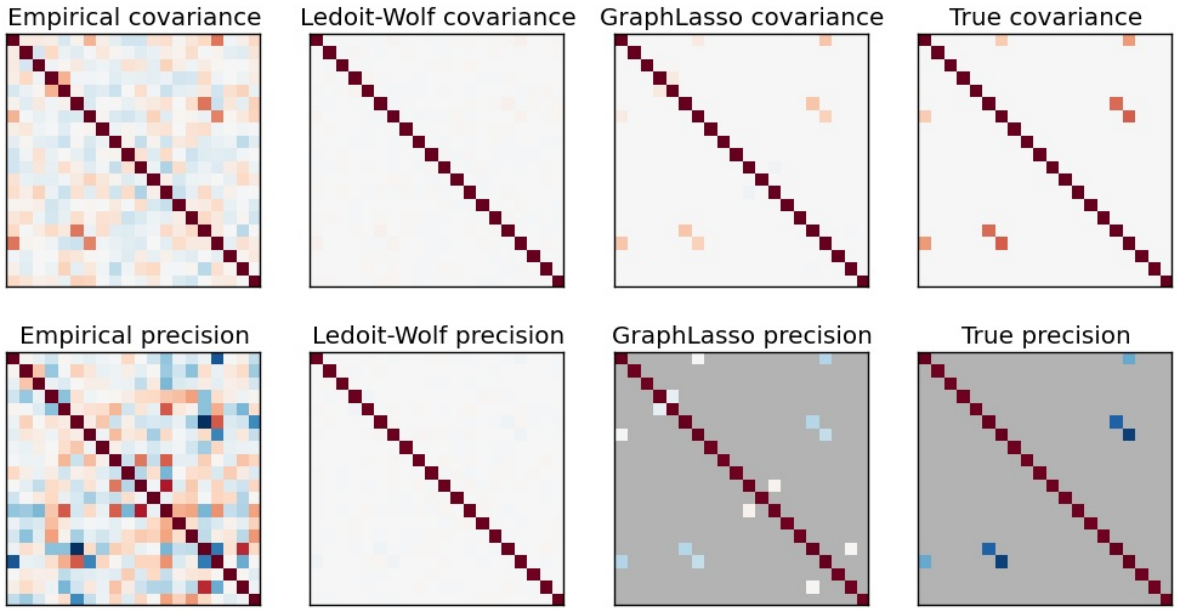


2.6.3. Sparse inverse covariance

The matrix inverse of the covariance matrix, often called the precision matrix, is proportional to the partial correlation matrix. It gives the partial independence relationship. In other words, if two features are independent conditionally on the others, the corresponding coefficient in the precision matrix will be zero. This is why it makes sense to estimate a sparse precision matrix: by learning independence relations from the data, the estimation of the covariance matrix is better conditioned. This is known as *covariance selection*.

In the small-samples situation, in which `n_samples` is on the order of `n_features` or smaller, sparse inverse covariance estimators tend to work better than shrunk covariance estimators. However, in the opposite situation, or for very correlated data, they can be numerically unstable. In addition, unlike shrinkage estimators, sparse estimators are able to recover off-diagonal structure.

The **`GraphLasso`** estimator uses an l1 penalty to enforce sparsity on the precision matrix: the higher its `alpha` parameter, the more sparse the precision matrix. The corresponding **`GraphLassoCV`** object uses cross-validation to automatically set the `alpha` parameter.



A comparison of maximum likelihood, shrinkage and sparse estimates of the covariance and precision matrix in the very small samples settings.

Note: Structure recovery

Recovering a graphical structure from correlations in the data is a challenging thing. If you are interested in such recovery keep in mind that:

- Recovery is easier from a correlation matrix than a covariance matrix: standardize your observations before running `GraphLasso`
- If the underlying graph has nodes with much more connections than the average node, the algorithm will miss some of these connections.
- If your number of observations is not large compared to the number of edges in your underlying graph, you will not recover it.
- Even if you are in favorable recovery conditions, the alpha parameter chosen by cross-validation (e.g. using the `GraphLassoCV` object) will lead to selecting too many edges. However, the relevant edges will have heavier weights than the irrelevant ones.

The mathematical formulation is the following:

$$\hat{K} = \mathrm{argmin}_K \big(\mathrm{tr} S K - \mathrm{logdet} K + \alpha \|K\|_1 \big)$$

Where K is the precision matrix to be estimated, and S is the sample covariance matrix. $\|K\|_1$ is the sum of the absolute values of off-diagonal coefficients of K. The algorithm employed to solve this problem is the GLasso algorithm, from the Friedman 2008 Biostatistics paper. It is the same algorithm as in the R `glasso` package.

Examples:

- *Sparse inverse covariance estimation*: example on synthetic data showing some recovery of a structure, and comparing to other covariance estimators.

- *Visualizing the stock market structure*: example on real stock market data, finding which symbols are most linked.

References:

- Friedman et al, "Sparse inverse covariance estimation with the graphical lasso", Biostatistics 9, pp 432, 2008

2.6.4. Robust Covariance Estimation

Real data set are often subjects to measurement or recording errors. Regular but uncommon observations may also appear for a variety of reason. Every observation which is very uncommon is called an outlier. The empirical covariance estimator and the shrunk covariance estimators presented above are very sensitive to the presence of outlying observations in the data. Therefore, one should use robust covariance estimators to estimate the covariance of its real data sets. Alternatively, robust covariance estimators can be used to perform outlier detection and discard/downweight some observations according to further processing of the data.

The `sklearn.covariance` package implements a robust estimator of covariance, the Minimum Covariance Determinant [3].

2.6.4.1. Minimum Covariance Determinant

The Minimum Covariance Determinant estimator is a robust estimator of a data set's covariance introduced by P.J. Rousseeuw in [3]. The idea is to find a given proportion (h) of "good" observations which are not outliers and compute their empirical covariance matrix. This empirical covariance matrix is then rescaled to compensate the performed selection of observations ("consistency step"). Having computed the Minimum Covariance Determinant estimator, one can give weights to observations according to their Mahalanobis distance, leading the a reweighted estimate of the covariance matrix of the data set ("reweighting step").

Rousseeuw and Van Driessen [4] developed the FastMCD algorithm in order to compute the Minimum Covariance Determinant. This algorithm is used in scikit-learn when fitting an MCD object to data. The FastMCD algorithm also computes a robust estimate of the data set location at the same time.

Raw estimates can be accessed as `raw_location_` and `raw_covariance_` attributes of a **`MinCovDet`** robust covariance estimator object.

[3] P. J. Rousseeuw. Least median of squares regression.
J. Am Stat Ass, 79:871, 1984.

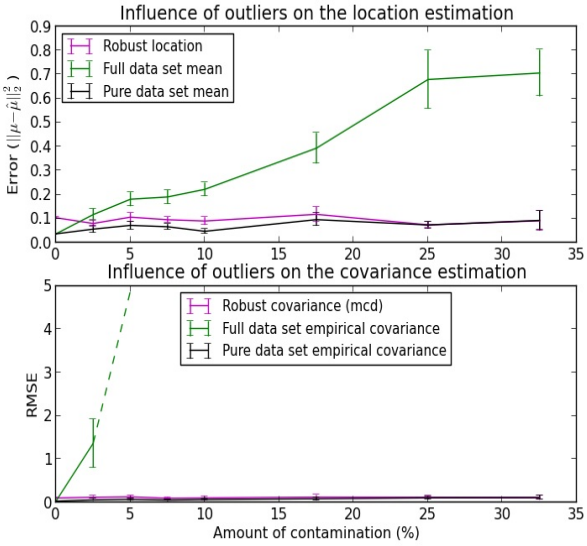
[4] A Fast Algorithm for the Minimum Covariance Determinant Estimator,
1999, American Statistical Association and the American Society for Quality, TECHNOMETRICS.

Examples:

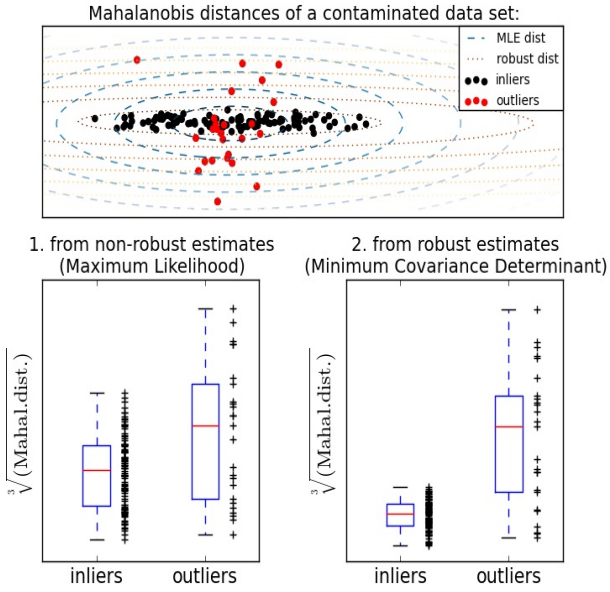
- See *Robust vs Empirical covariance estimate* for an example on how to fit a **`MinCovDet`** object to data and see how the estimate remains accurate despite the presence of outliers.
- See *Robust covariance estimation and Mahalanobis distances relevance* to visualize the difference

between `EmpiricalCovariance` and `MinCovDet` covariance estimators in terms of Mahalanobis distance (so we get a better estimate of the precision matrix too).

Influence of outliers on location and covariance estimates



Separating inliers from outliers using a Mahalanobis distance



Previous Next

2.7. Novelty and Outlier Detection

Many applications require being able to decide whether a new observation belongs to the same distribution as existing observations (it is an *inlier*), or should be considered as different (it is an outlier). Often, this ability is used to clean real data sets. Two important distinction must be made:

novelty detection:
The training data is not polluted by outliers, and we are interested in detecting anomalies in new observations.

outlier detection:
The training data contains outliers, and we need to fit the central mode of the training data, ignoring the deviant observations.

The scikit-learn project provides a set of machine learning tools that can be used both for novelty or outliers detection. This strategy is implemented with objects learning in an unsupervised way from the data:

```
estimator.fit(X_train)
```

new observations can then be sorted as inliers or outliers with a *predict* method:

```
estimator.predict(X_test)
```

Inliers are labeled 1, while outliers are labeled -1.

2.7.1. Novelty Detection

Consider a data set of n observations from the same distribution described by p features. Consider now that we add one more observation to that data set. Is the new observation so different from the others that we can doubt it is regular? (i.e. does it come from the same distribution?) Or on the contrary, is it so similar to the other that we cannot distinguish it from the original observations? This is the question addressed by the novelty detection tools and methods.

In general, it is about to learn a rough, close frontier delimiting the contour of the initial observations distribution, plotted in embedding p-dimensional space. Then, if further observations lay within the frontier-delimited subspace, they are considered as coming from the same population than the initial observations. Otherwise, if they lay outside the frontier, we can say that they are abnormal with a given confidence in our assessment.

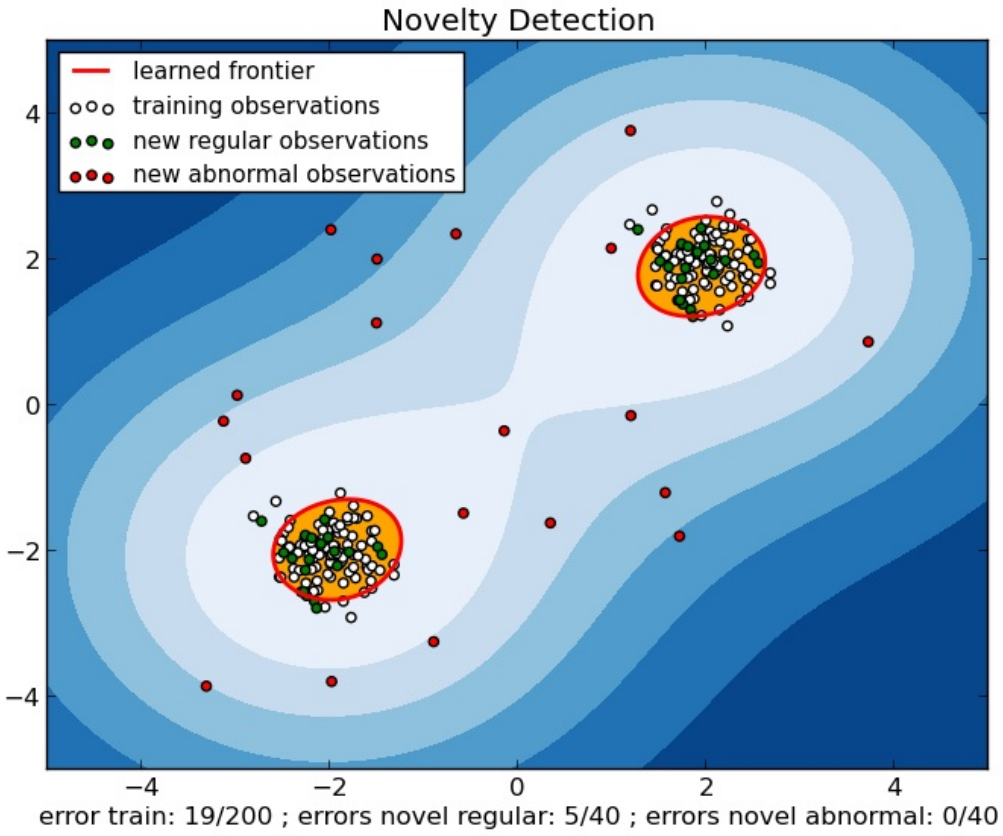
The One-Class SVM has been introduced by Schölkopf et al. for that purpose and implemented in the *Support Vector Machines* module in the `svm.OneClassSVM` object. It requires the choice of a kernel and a scalar parameter to define a frontier. The RBF kernel is usually chosen although there exists no exact formula or algorithm to set its bandwidth parameter. This is the default in the scikit-learn implementation. The ν parameter, also known as the margin of the One-Class SVM, corresponds to the probability of finding a new, but regular, observation outside the frontier.

References:

- Estimating the support of a high-dimensional distribution Schölkopf, Bernhard, et al. Neural computation 13.7 (2001): 1443-1471.

Examples:

- See *One-class SVM with non-linear kernel (RBF)* for visualizing the frontier learned around some data by a `svm.OneClassSVM` object.



2.7.2. Outlier Detection

Outlier detection is similar to novelty detection in the sense that the goal is to separate a core of regular observations from some polluting ones, called “outliers”. Yet, in the case of outlier detection, we don’t have a clean data set representing the population of regular observations that can be used to train any tool.

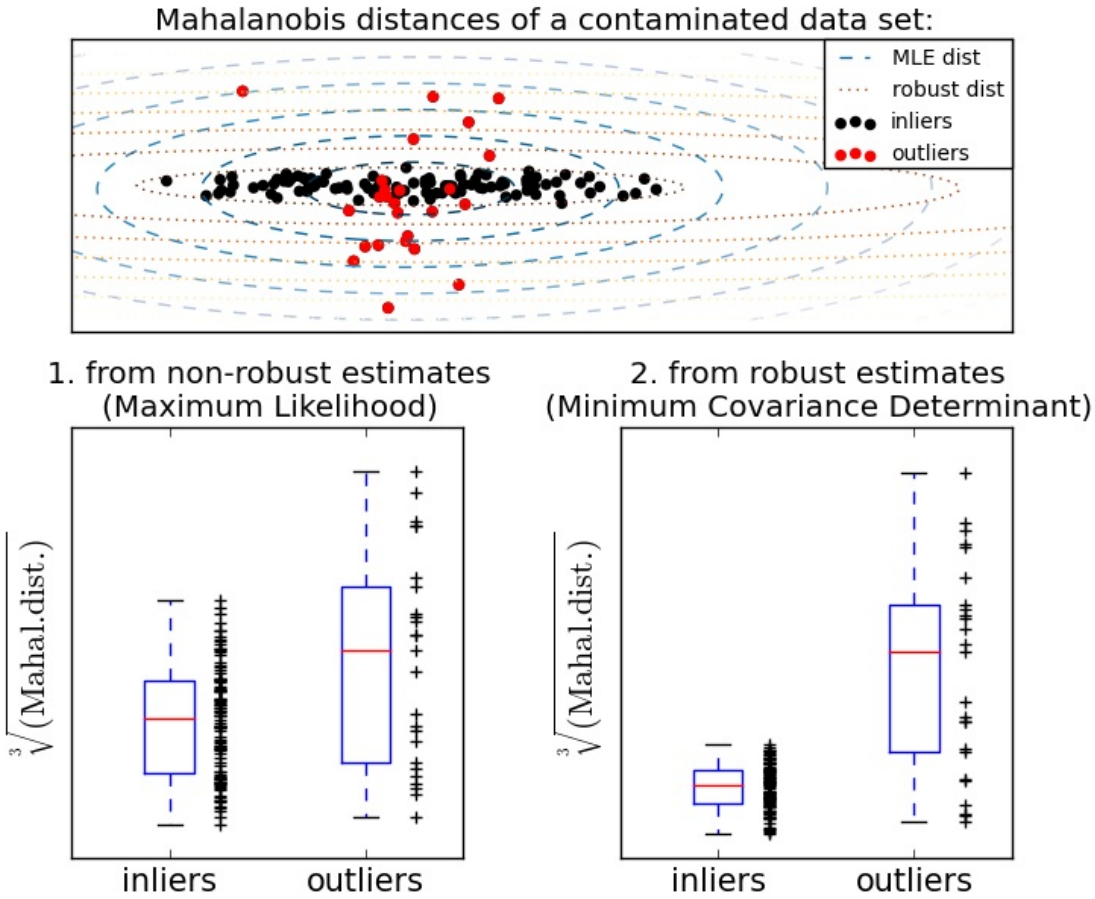
2.7.2.1. Fitting an elliptic envelope

One common way of performing outlier detection is to assume that the regular data come from a known distribution (e.g. data are Gaussian distributed). From this assumption, we generally try to define the “shape” of the data, and can define outlying observations as observations which stand far enough from the fit shape.

The scikit-learn provides an object `covariance.EllipticEnvelope` that fits a robust covariance estimate to the data, and thus fits an ellipse to the central data points, ignoring points outside the central mode.

For instance, assuming that the inlier data are Gaussian distributed, it will estimate the inlier location and covariance in a robust way (i.e. whithout being influenced by outliers). The Mahalanobis distances obtained

from this estimate is used to derive a measure of outlyingness. This strategy is illustrated below.



Examples:

- See *Robust covariance estimation and Mahalanobis distances relevance* for an illustration of the difference between using a standard (`covariance.EmpiricalCovariance`) or a robust estimate (`covariance.MinCovDet`) of location and covariance to assess the degree of outlyingness of an observation.

References:

[RD1999] Rousseeuw, P.J., Van Driessen, K. “A fast algorithm for the minimum covariance determinant estimator” Technometrics 41(3), 212 (1999)

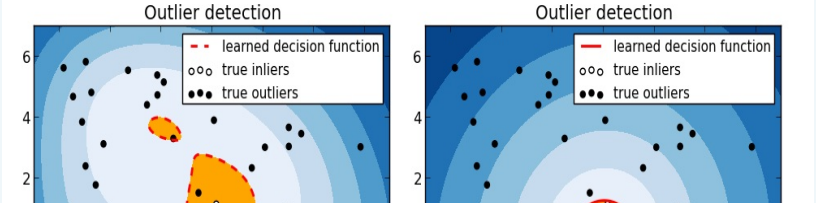
2.7.2.2. One-class SVM versus elliptic envelope

Strictly-speaking, the One-class SVM is not an outlier-detection method, but a novelty-detection method: its training set should not be contaminated by outliers as it may fit them. That said, outlier detection in high-dimension, or without any assumptions on the distribution of the inlying data is very challenging, and a One-class SVM gives useful results in these situations.

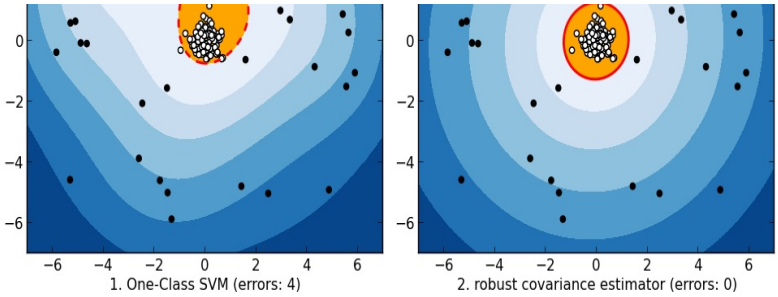
The examples below illustrate how the performance of the `covariance.EllipticEnvelope` degrades as the data is less and less unimodal. `svm.OneClassSVM` works better on data with multiple modes.

Comparing One-class SVM approach, and elliptic envelope

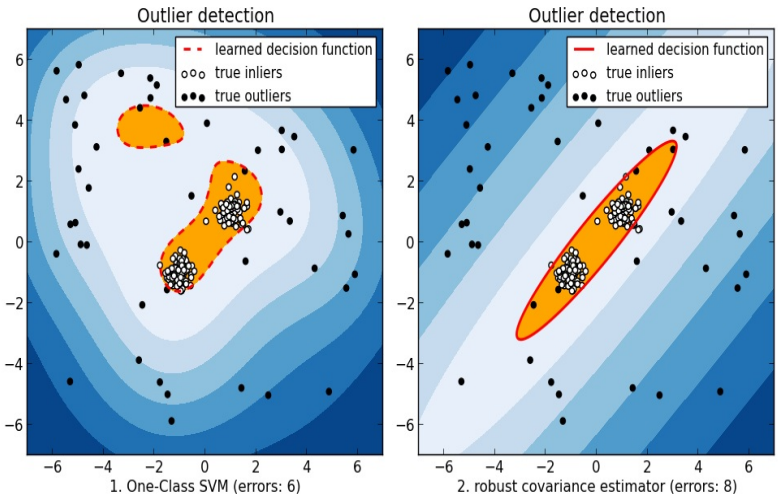
For a inlier mode well-centered and elliptic, the `svm.OneClassSVM` is not able to benefit from the rotational symmetry of the inlier population. In addition, it fits a bit the outliers present in the training set. On the opposite, the decision rule based



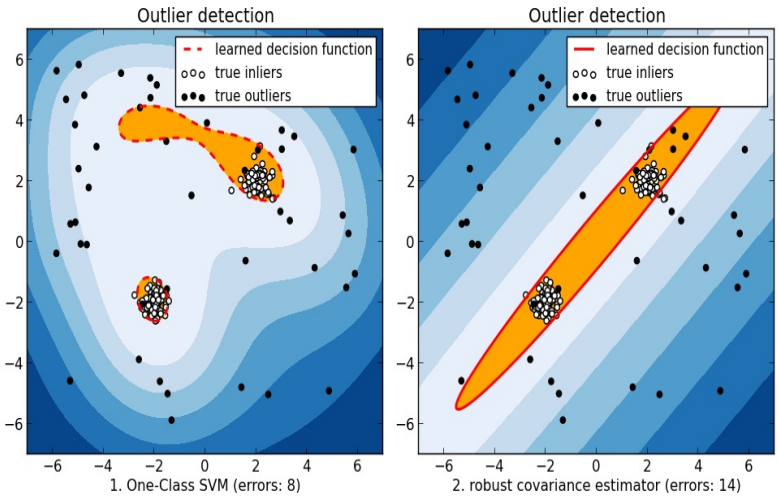
on fitting an `covariance.EllipticEnvelope` learns an ellipse, which fits well the inlier distribution.



As the inlier distribution becomes bimodal, the `covariance.EllipticEnvelope` does not fit well the inliers. However, we can see that the `svm.OneClassSVM` tends to overfit: because it has not model of inliers, it interprets a region where, by chance some outliers are clustered, as inliers.



If the inlier distribution is strongly non Gaussian, the `svm.OneClassSVM` is able to recover a reasonable approximation, whereas the `covariance.EllipticEnvelope` completely fails.



Examples:

- See *Outlier detection with several methods.* for a comparison of the `svm.OneClassSVM` (tuned to perform like an outlier detection method) and a covariance-based outlier detection with `covariance.MinCovDet`.

Previous Next

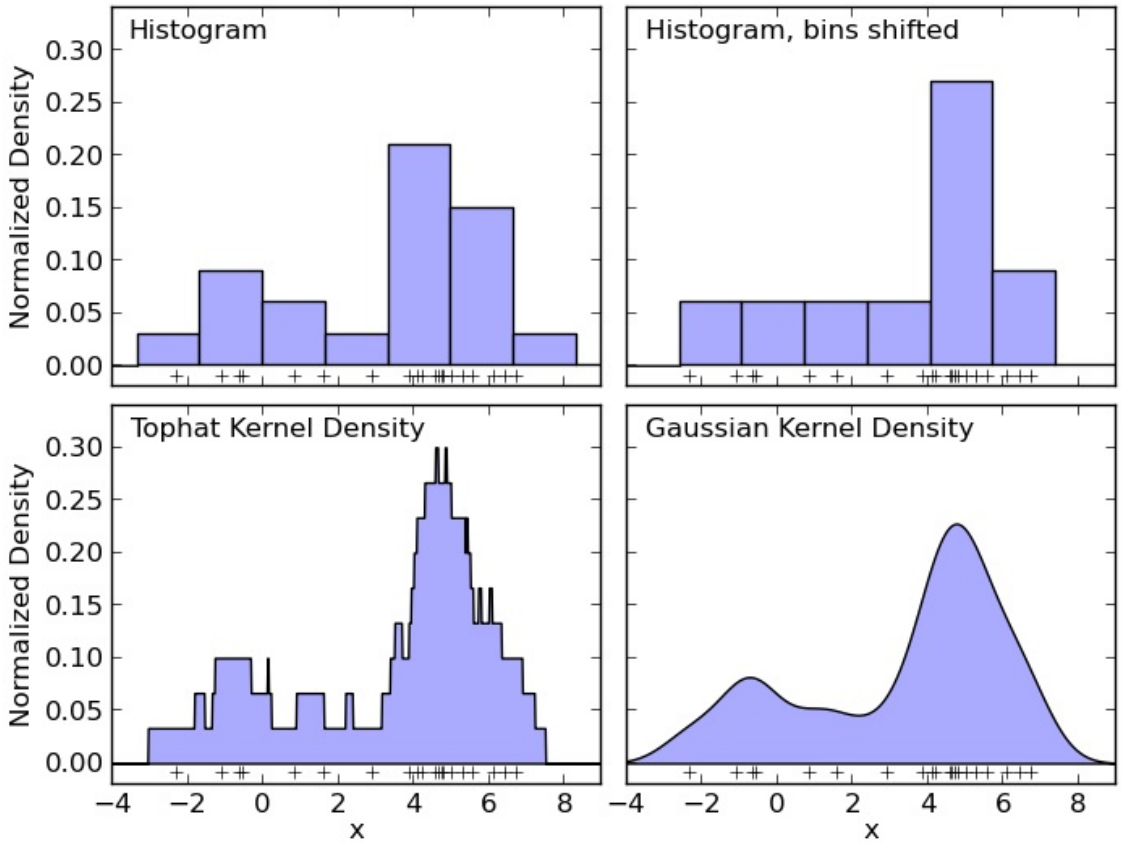
2.8. Density Estimation

Density estimation walks the line between unsupervised learning, feature engineering, and data modeling. Some of the most popular and useful density estimation techniques are mixture models such as Gaussian Mixtures (`sklearn.mixture.GMM`), and neighbor-based approaches such as the kernel density estimate (`sklearn.neighbors.KernelDensity`). Gaussian Mixtures are discussed more fully in the context of *clustering*, because the technique is also useful as an unsupervised clustering scheme.

Density estimation is a very simple concept, and most people are already familiar with one common density estimation technique: the histogram.

2.8.1. Density Estimation: Histograms

A histogram is a simple visualization of data where bins are defined, and the number of data points within each bin is tallied. An example of a histogram can be seen in the upper-left panel of the following figure:



A major problem with histograms, however, is that the choice of binning can have a disproportionate effect on the resulting visualization. Consider the upper-right panel of the above figure. It shows a histogram over the same data, with the bins shifted right. The results of the two visualizations look entirely different, and

might lead to different interpretations of the data.

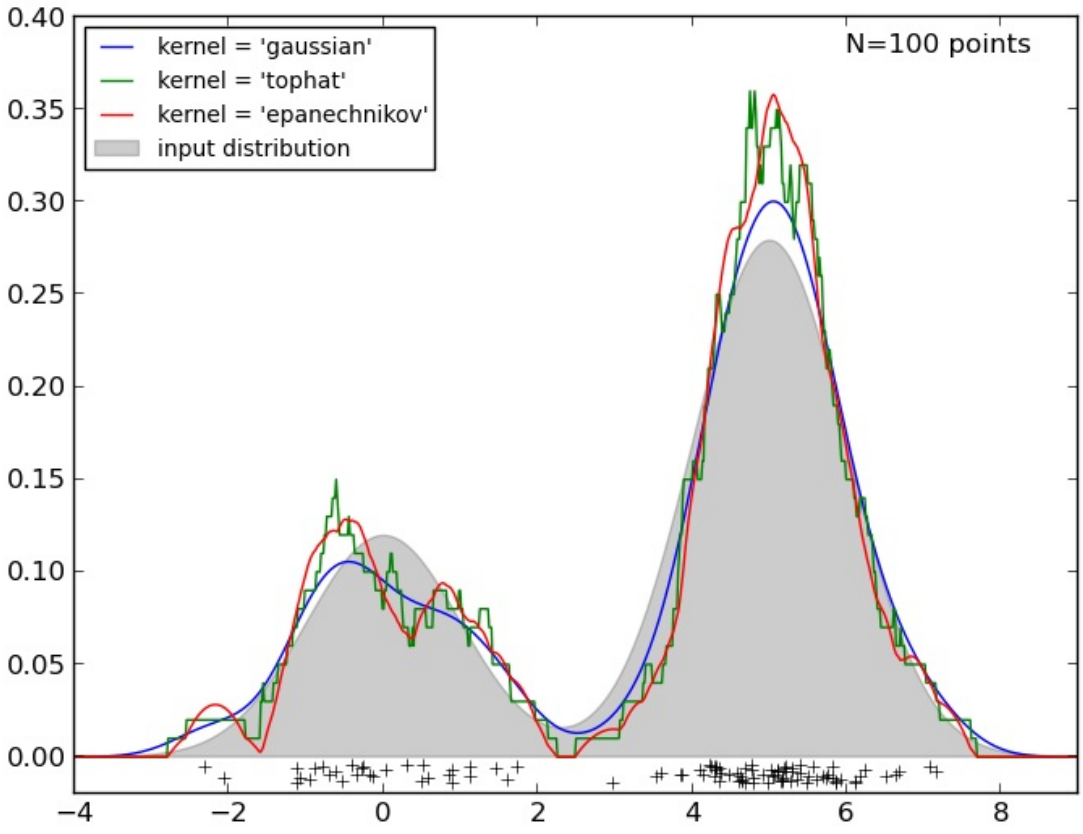
Intuitively, one can also think of a histogram as a stack of blocks, one block per point. By stacking the blocks in the appropriate grid space, we recover the histogram. But what if, instead of stacking the blocks on a regular grid, we center each block on the point it represents, and sum the total height at each location? This idea leads to the lower-left visualization. It is perhaps not as clean as a histogram, but the fact that the data drive the block locations mean that it is a much better representation of the underlying data.

This visualization is an example of a *kernel density estimation*, in this case with a top-hat kernel (i.e. a square block at each point). We can recover a smoother distribution by using a smoother kernel. The bottom-right plot shows a Gaussian kernel density estimate, in which each point contributes a Gaussian curve to the total. The result is a smooth density estimate which is derived from the data, and functions as a powerful non-parametric model of the distribution of points.

2.8.2. Kernel Density Estimation

Kernel density estimation in scikit-learn is implemented in the `sklearn.neighbors.KernelDensity` estimator, which uses the Ball Tree or KD Tree for efficient queries (see *Nearest Neighbors* for a discussion of these). Though the above example uses a 1D data set for simplicity, kernel density estimation can be performed in any number of dimensions, though in practice the curse of dimensionality causes its performance to degrade in high dimensions.

In the following figure, 100 points are drawn from a bimodal distribution, and the kernel density estimates are shown for three choices of kernels:



It's clear how the kernel shape affects the smoothness of the resulting distribution. The scikit-learn kernel density estimator can be used as follows:

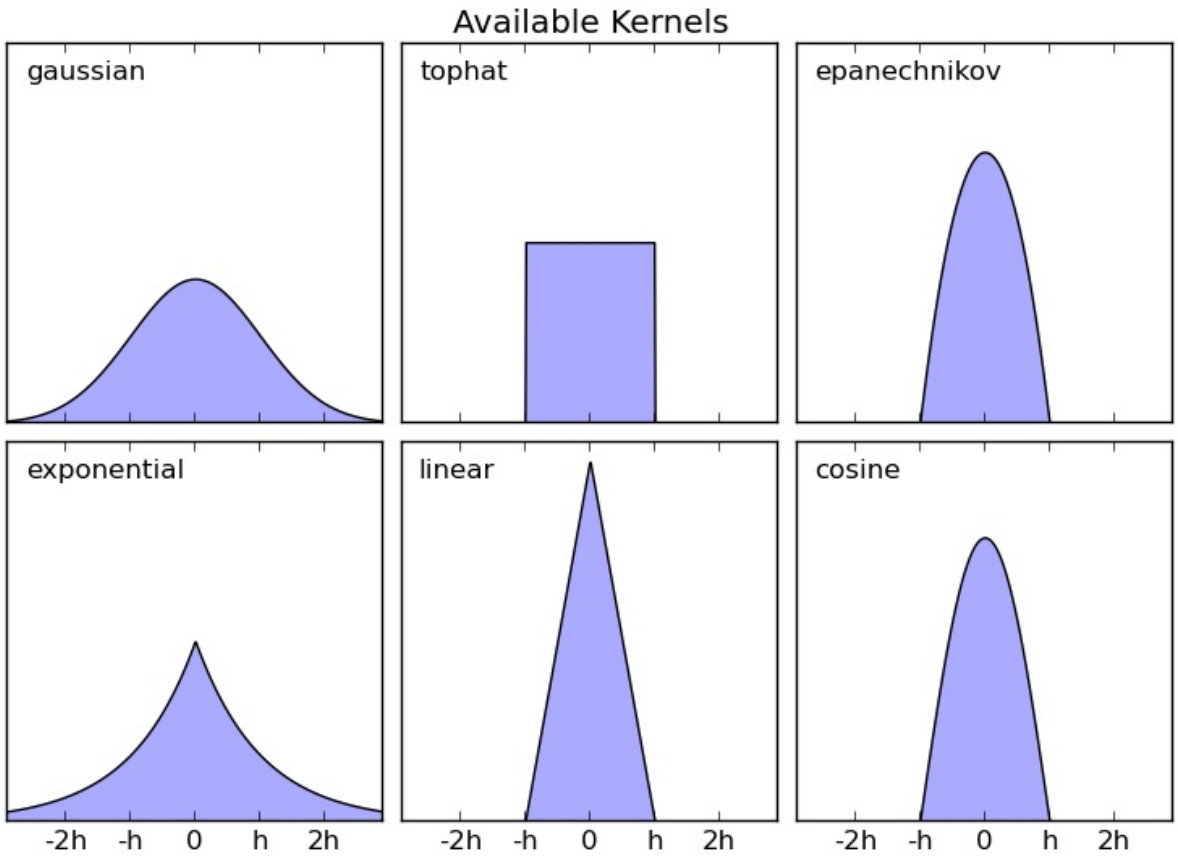
```
>>> from sklearn.neighbors.kde import KernelDensity
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> kde = KernelDensity(kernel='gaussian', bandwidth=0.2).fit(X)
>>> kde.score_samples(X)
array([-0.41075698, -0.41075698, -0.41076071, -0.41075698, -0.41075698,
       -0.41076071])
```

Here we have used `kernel='gaussian'`, as seen above. Mathematically, a kernel is a positive function $K(x;h)$ which is controlled by the bandwidth parameter h. Given this kernel form, the density estimate at a point y within a group of points $x_i; i = 1 \cdots N$ is given by:

$$\rho_K(y) = \sum_{i=1}^{N} K((y - x_i)/h)$$

The bandwidth here acts as a smoothing parameter, controlling the tradeoff between bias and variance in the result. A large bandwidth leads to a very smooth (i.e. high-bias) density distribution. A small bandwidth leads to an unsmooth (i.e. high-variance) density distribution.

`sklearn.neighbors.KernelDensity` implements several common kernel forms, which are shown in the following figure:



The form of these kernels is as follows:

- Gaussian kernel (`kernel = 'gaussian'`)

$K(x; h) \propto \exp(-\frac{x^2}{2h^2})$

- Tophat kernel (`kernel = 'tophat'`)

$K(x; h) \propto 1$ if $x < h$

- Epanechnikov kernel (`kernel = 'epanechnikov'`)

$K(x; h) \propto 1 - \frac{x^2}{h^2}$

- Exponential kernel (`kernel = 'exponential'`)

$K(x; h) \propto \exp(-x/h)$

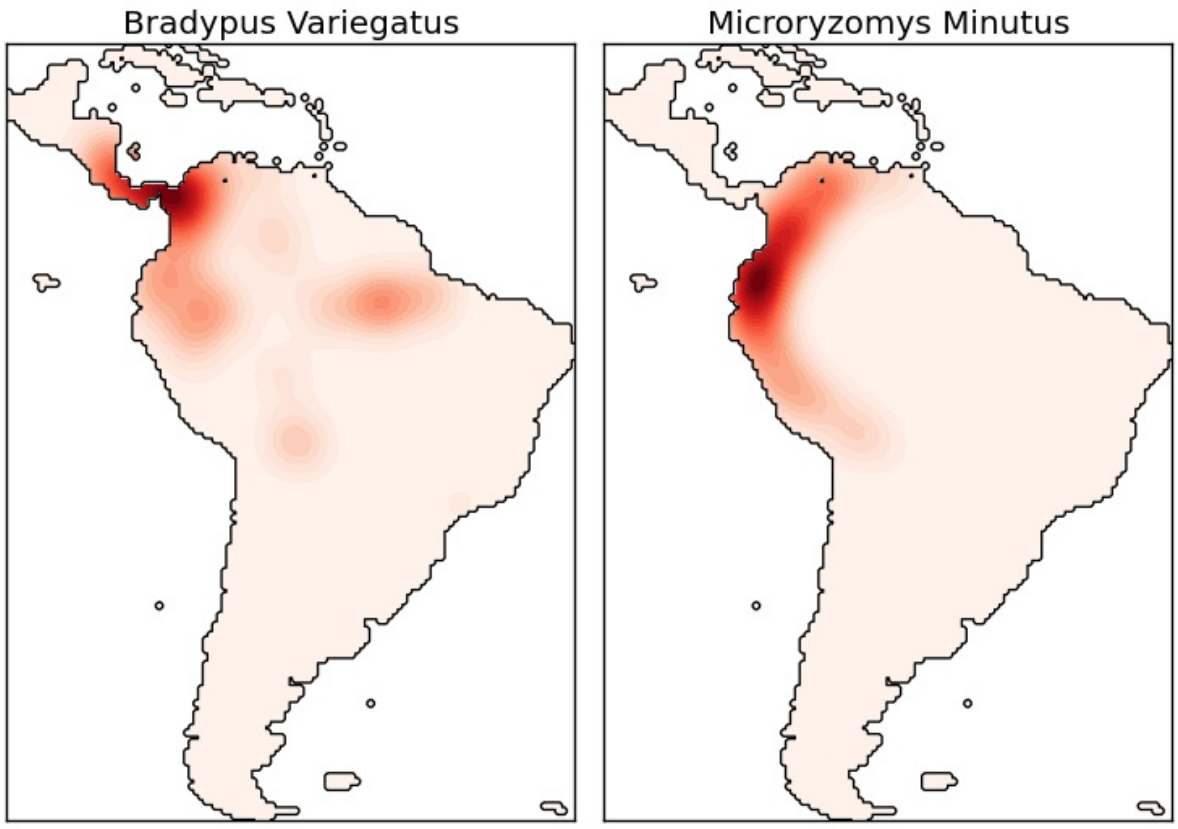
- Linear kernel (`kernel = 'linear'`)

$K(x; h) \propto 1 - x/h$ if $x < h$

- Cosine kernel (`kernel = 'cosine'`)

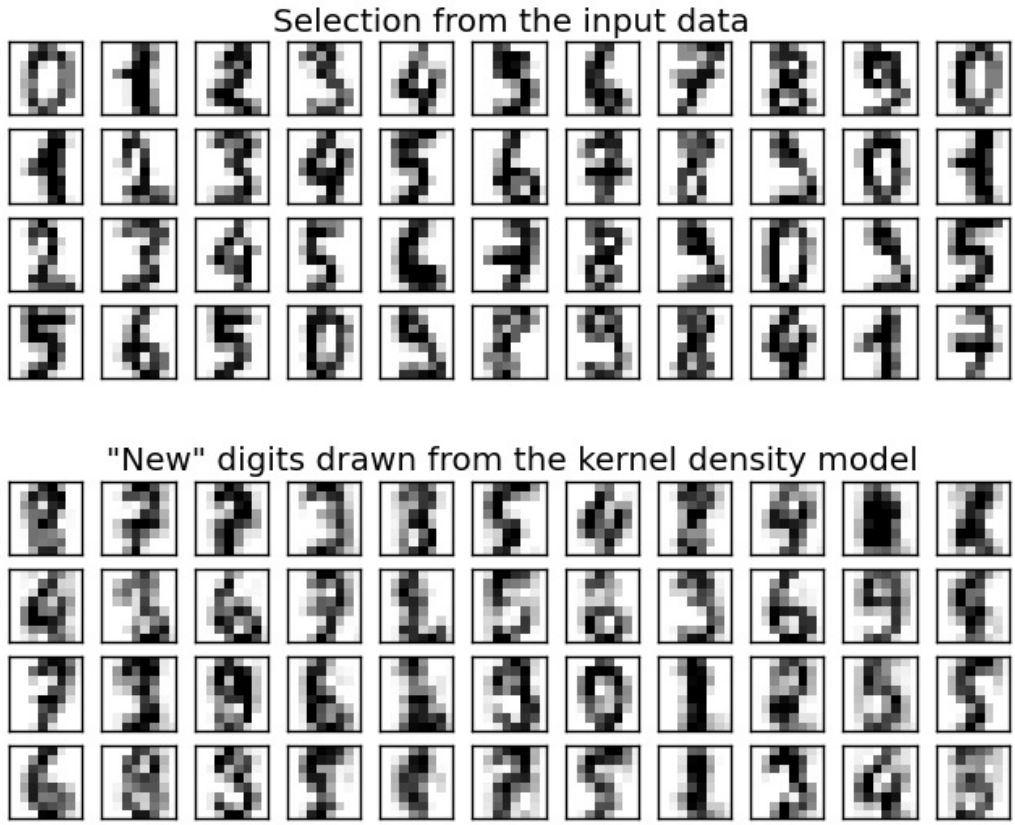
$K(x; h) \propto \cos(\frac{\pi x}{2h})$ if $x < h$

The kernel density estimator can be used with any of the valid distance metrics (see **`sklearn.neighbors.DistanceMetric`** for a list of available metrics), though the results are properly normalized only for the Euclidean metric. One particularly useful metric is the Haversine distance which measures the angular distance between points on a sphere. Here is an example of using a kernel density estimate for a visualization of geospatial data, in this case the distribution of observations of two different species on the South American continent:



One other useful application of kernel density estimation is to learn a non-parametric generative model of a

dataset in order to efficiently draw new samples from this generative model. Here is an example of using this process to create a new set of hand-written digits, using a Gaussian kernel learned on a PCA projection of the data:



The “new” data consists of linear combinations of the input data, with weights probabilistically drawn given the KDE model.

Examples:

- *Simple 1D Kernel Density Estimation*: computation of simple kernel density estimates in one dimension.
- *Kernel Density Estimation*: an example of using Kernel Density estimation to learn a generative model of the hand-written digits data, and drawing new samples from this model.
- *Kernel Density Estimate of Species Distributions*: an example of Kernel Density estimation using the Haversine distance metric to visualize geospatial data

Previous Next

2.9. Neural network models (unsupervised)

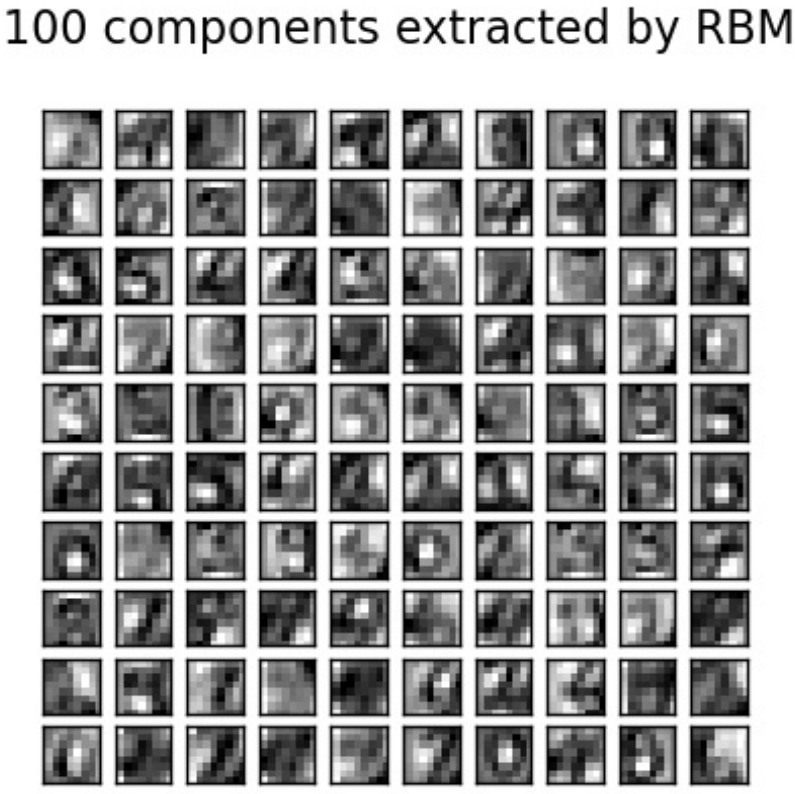
2.9.1. Restricted Boltzmann machines

Restricted Boltzmann machines (RBM) are unsupervised nonlinear feature learners based on a probabilistic model. The features extracted by an RBM or a hierarchy of RBMs often give good results when fed into a linear classifier such as a linear SVM or a perceptron.

The model makes assumptions regarding the distribution of inputs. At the moment, scikit-learn only provides `BernoulliRBM`, which assumes the inputs are either binary values or values between 0 and 1, each encoding the probability that the specific feature would be turned on.

The RBM tries to maximize the likelihood of the data using a particular graphical model. The parameter learning algorithm used (*Stochastic Maximum Likelihood*) prevents the representations from straying far from the input data, which makes them capture interesting regularities, but makes the model less useful for small datasets, and usually not useful for density estimation.

The method gained popularity for initializing deep neural networks with the weights of independent RBMs. This method is known as unsupervised pre-training.

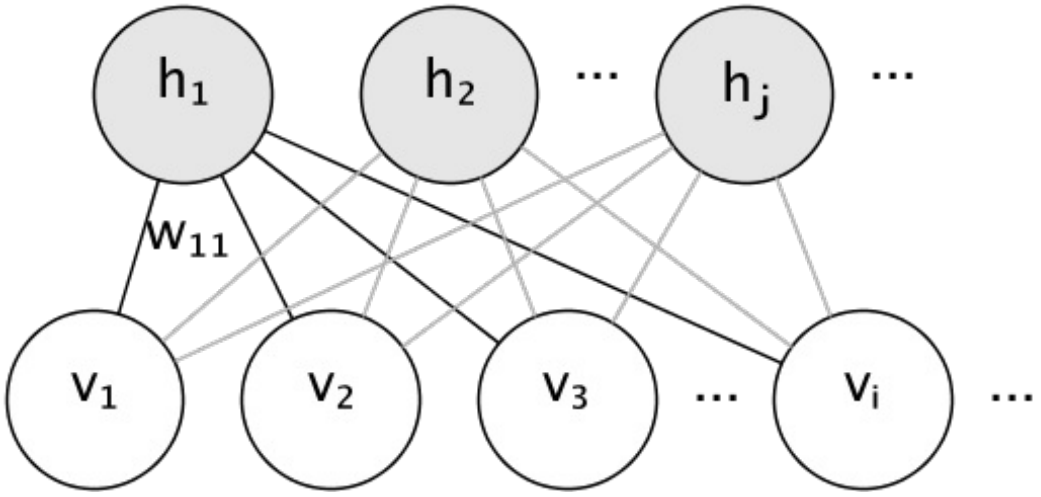


Examples:

- *Restricted Boltzmann Machine features for digit classification*

2.9.1.1. Graphical model and parametrization

The graphical model of an RBM is a fully-connected bipartite graph.



The nodes are random variables whose states depend on the state of the other nodes they are connected to. The model is therefore parameterized by the weights of the connections, as well as one intercept (bias) term for each visible and hidden unit, ommited from the image for simplicity.

The energy function measures the quality of a joint assignment:

$$E(\mathbf{v}, \mathbf{h}) = \sum_i \sum_j w_{ij} v_i h_j + \sum_i b_i v_i + \sum_j c_j h_j$$

In the formula above, $\mathbf{b}$ and $\mathbf{c}$ are the intercept vectors for the visible and hidden layers, respectively. The joint probability of the model is defined in terms of the energy:

$$P(\mathbf{v}, \mathbf{h}) = \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{Z}$$

The word *restricted* refers to the bipartite structure of the model, which prohibits direct interaction between hidden units, or between visible units. This means that the following conditional independencies are assumed:

$$h_i \perp h_j | \mathbf{v}$$
$$v_i \perp v_j | \mathbf{h}$$

The bipartite structure allows for the use of efficient block Gibbs sampling for inference.

2.9.1.2. Bernoulli Restricted Boltzmann machines

In the `BernoulliRBM`, all units are binary stochastic units. This means that the input data should either be binary, or real-valued between 0 and 1 signifying the probability that the visible unit would turn on or off. This is a good model for character recognition, where the interest is on which pixels are active and which aren't. For images of natural scenes it no longer fits because of background, depth and the tendency of neighbouring pixels to take the same values.

The conditional probability distribution of each unit is given by the logistic sigmoid activation function of the input it receives:

$$P(v_i = 1|\mathbf{h}) = \sigma(\sum_j w_{ij}h_j + b_i)$$
$$P(h_i = 1|\mathbf{v}) = \sigma(\sum_i w_{ij}v_i + c_j)$$

where σ is the logistic sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

2.9.1.3. Stochastic Maximum Likelihood learning

The training algorithm implemented in `BernoulliRBM` is known as Stochastic Maximum Likelihood (SML) or Persistent Contrastive Divergence (PCD). Optimizing maximum likelihood directly is infeasible because of the form of the data likelihood:

$$\log P(v) = \log \sum_h e^{-E(v,h)} - \log \sum_{x,y} e^{-E(x,y)}$$

For simplicity the equation above is written for a single training example. The gradient with respect to the weights is formed of two terms corresponding to the ones above. They are usually known as the positive gradient and the negative gradient, because of their respective signs. In this implementation, the gradients are estimated over mini-batches of samples.

In maximizing the log-likelihood, the positive gradient makes the model prefer hidden states that are compatible with the observed training data. Because of the bipartite structure of RBMs, it can be computed efficiently. The negative gradient, however, is intractable. Its goal is to lower the energy of joint states that the model prefers, therefore making it stay true to the data. It can be approximated by Markov chain Monte Carlo using block Gibbs sampling by iteratively sampling each of v and h given the other, until the chain mixes. Samples generated in this way are sometimes refered as fantasy particles. This is inefficient and it is difficult to determine whether the Markov chain mixes.

The Contrastive Divergence method suggests to stop the chain after a small number of iterations, k, usually even 1. This method is fast and has low variance, but the samples are far from the model distribution.

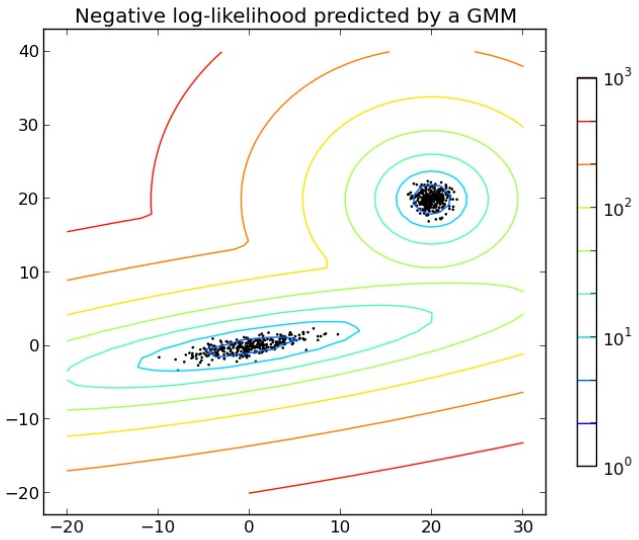
Persistent Contrastive Divergence addresses this. Instead of starting a new chain each time the gradient is needed, and performing only one Gibbs sampling step, in PCD we keep a number of chains (fantasy particles) that are updated k Gibbs steps after each weight update. This allows the particles to explore the space more thoroughly.

References:

- "A fast learning algorithm for deep belief nets" G. Hinton, S. Osindero, Y.-W. Teh, 2006
- "Training Restricted Boltzmann Machines using Approximations to the Likelihood Gradient" T. Tieleman, 2008

2.1. Gaussian mixture models

sklearn.mixture is a package which enables one to learn Gaussian Mixture Models (diagonal, spherical, tied and full covariance matrices supported), sample them, and estimate them from data. Facilities to help determine the appropriate number of components are also provided.



Two-component Gaussian mixture model: *data points, and equi-probability surfaces of the model.*

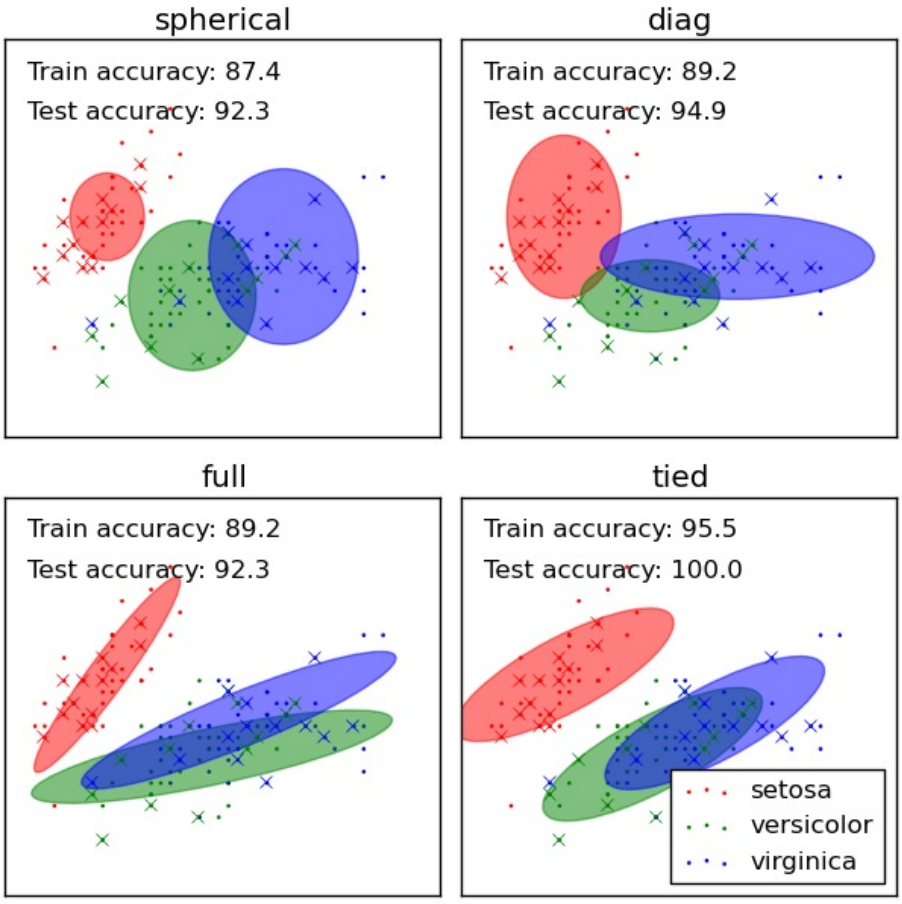
A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think of mixture models as generalizing k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians.

Scikit-learn implements different classes to estimate Gaussian mixture models, that correspond to different estimation strategies, detailed below.

2.1.1. GMM classifier

The `GMM` object implements the *expectation-maximization* (EM) algorithm for fitting mixture-of-Gaussian models. It can also draw confidence ellipsoids for multivariate models, and compute the Bayesian Information Criterion to assess the number of clusters in the data. A `GMM.fit` method is provided that learns a Gaussian Mixture Model 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.

The `GMM` comes with different options to constrain the covariance of the difference classes estimated: spherical, diagonal, tied or full covariance.



Examples:

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

2.1.1.1. Pros and cons of class `GMM`: expectation-maximization inference

2.1.1.1.1. Pros

Speed:	it is the fastest algorithm for learning mixture models
Agnostic:	as this algorithm maximizes only the likelihood, it will not bias the means towards zero, or bias the cluster sizes to have specific structures that might or might not apply.

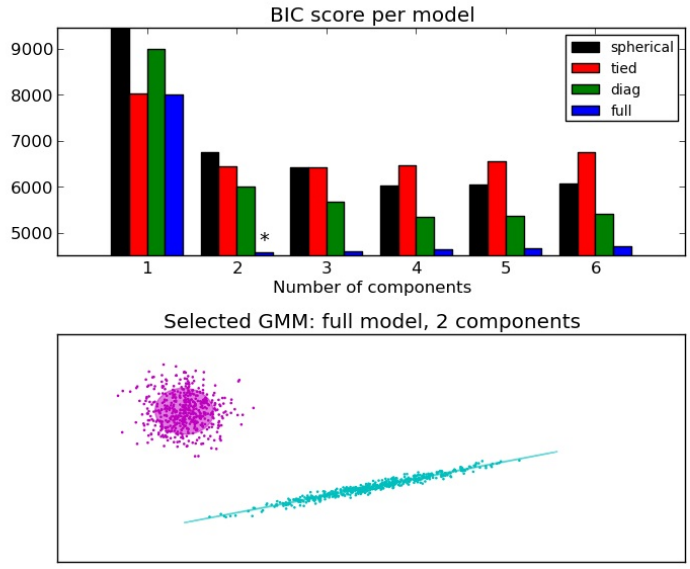
2.1.1.1.2. Cons

Singularities:	when one has insufficiently many points per mixture, estimating the covariance matrices becomes difficult, and the algorithm is known to diverge and find solutions with infinite likelihood unless one regularizes the covariances artificially.
Number of components:	
	this algorithm will always use all the components it has access to, needing held-out data or information theoretical criteria to decide how many components to use in the absence of external cues.

2.1.1.2. Selecting the number of components in a classical GMM

The BIC criterion can be used to select the number of components in a GMM in an efficient way. In theory, it recovers the true number of components only in the asymptotic regime (i.e. if much data is available). Note

that using a *DPGMM* avoids the specification of the number of components for a Gaussian mixture model.



Examples:

- See *Gaussian Mixture Model Selection* for an example of model selection performed with classical GMM.

2.1.1.3. Estimation algorithm Expectation-maximization

The main difficulty in learning Gaussian mixture models from unlabeled data is that it is one usually doesn't know which points came from which latent component (if one has access to this information it gets very easy to fit a separate Gaussian distribution to each set of points). Expectation-maximization is a well-fundamented statistical algorithm to get around this problem by an iterative process. First one assumes random components (randomly centered on data points, learned from k-means, or even just normally distributed around the origin) and computes for each point a probability of being generated by each component of the model. Then, one tweaks the parameters to maximize the likelihood of the data given those assignments. Repeating this process is guaranteed to always converge to a local optimum.

2.1.2. VBGMM classifier: variational Gaussian mixtures

The `VBGMM` object implements a variant of the Gaussian mixture model with *variational inference* algorithms. The API is identical to `GMM`. It is essentially a middle-ground between `GMM` and `DPGMM`, as it has some of the properties of the Dirichlet process.

2.1.2.1. Pros and cons of class `VBGMM`: variational inference

2.1.2.1.1. Pros

Regularization:	due to the incorporation of prior information, variational solutions have less pathological special cases than expectation-maximization solutions. One can then use full covariance matrices in high dimensions or in cases where some components might be centered around a single point without risking divergence.

2.1.2.1.2. Cons

Bias:	to regularize a model one has to add biases. The variational algorithm will bias all the means towards the origin (part of the prior information adds a “ghost point” in the origin to every mixture component) and it will bias the covariances to be more spherical. It will also, depending on the concentration parameter, bias the cluster structure either towards uniformity or towards a rich-get-richer scenario.
Hyperparameters:	
	this algorithm needs an extra hyperparameter that might need experimental tuning via cross-validation.

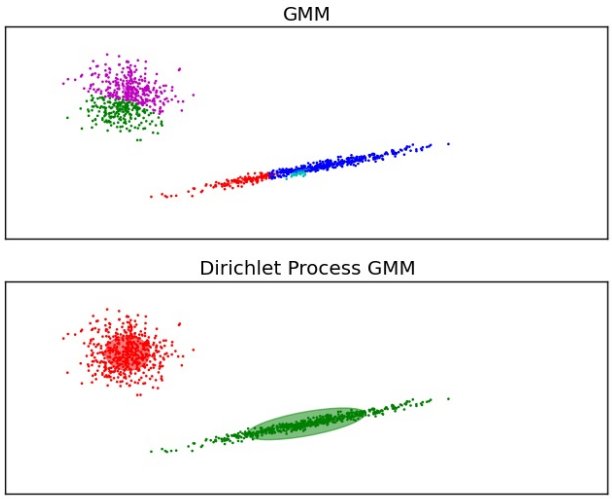
2.1.2.2. Estimation algorithm: variational inference

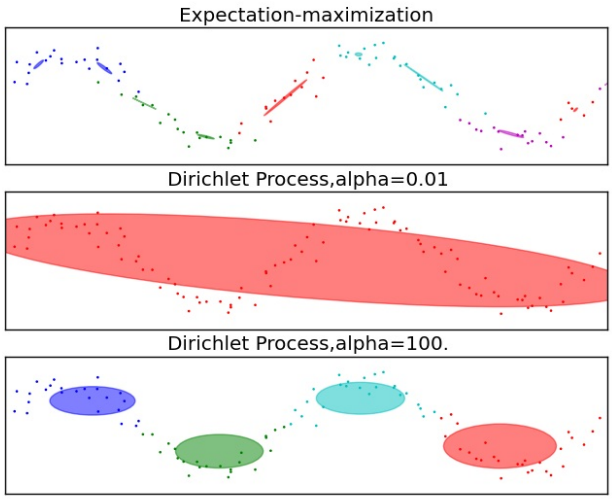
Variational inference is an extension of expectation-maximization that maximizes a lower bound on model evidence (including priors) instead of data likelihood. The principle behind variational methods is the same as expectation-maximization (that is both are iterative algorithms that alternate between finding the probabilities for each point to be generated by each mixture and fitting the mixtures to these assigned points), but variational methods add regularization by integrating information from prior distributions. This avoids the singularities often found in expectation-maximization solutions but introduces some subtle biases to the model. Inference is often notably slower, but not usually as much so as to render usage unpractical.

Due to its Bayesian nature, the variational algorithm needs more hyper-parameters than expectation-maximization, the most important of these being the concentration parameter `alpha`. Specifying a high value of alpha leads more often to uniformly-sized mixture components, while specifying small (between 0 and 1) values will lead to some mixture components getting almost all the points while most mixture components will be centered on just a few of the remaining points.

2.1.3. DPGMM classifier: Infinite Gaussian mixtures

The **DPGMM** object implements a variant of the Gaussian mixture model with a variable (but bounded) number of components using the Dirichlet Process. The API is identical to **GMM**. This class doesn’t require the user to choose the number of components, and at the expense of extra computational time the user only needs to specify a loose upper bound on this number and a concentration parameter.





The examples above compare Gaussian mixture models with fixed number of components, to DPGMM models. **On the left** the GMM is fitted with 5 components on a dataset composed of 2 clusters. We can see that the DPGMM is able to limit itself to only 2 components whereas the GMM fits the data fit too many components. Note that with very little observations, the DPGMM can take a conservative stand, and fit only one component. **On the right** we are fitting a dataset not well-depicted by a mixture of Gaussian. Adjusting the *alpha* parameter of the DPGMM controls the number of components used to fit this data.

Examples:

- See *Gaussian Mixture Model Ellipsoids* for an example on plotting the confidence ellipsoids for both `GMM` and `DPGMM`.
- *Gaussian Mixture Model Sine Curve* shows using `GMM` and `DPGMM` to fit a sine wave

2.1.3.1. Pros and cons of class `DPGMM`: Dirichlet process mixture model

2.1.3.1.1. Pros

Less sensitivity to the number of parameters:
unlike finite models, which will almost always use all components as much as they can, and hence will produce wildly different solutions for different numbers of components, the Dirichlet process solution won't change much with changes to the parameters, leading to more stability and less tuning.
No need to specify the number of components:
only an upper bound of this number needs to be provided. Note however that the DPMM is not a formal model selection procedure, and thus provides no guarantee on the result.

2.1.3.1.2. Cons

Speed:	the extra parametrization necessary for variational inference and for the structure of the Dirichlet process can and will make inference slower, although not by much.
Bias:	as in variational techniques, but only more so, there are many implicit biases in the Dirichlet process and the inference algorithms, and whenever there is a mismatch between these biases and the data it might be possible to fit better models using a finite mixture.

2.1.3.2. The Dirichlet Process

Here we describe variational inference algorithms on Dirichlet process mixtures. The Dirichlet process is a

prior probability distribution on *clusterings with an infinite, unbounded, number of partitions*. Variational techniques let us incorporate this prior structure on Gaussian mixture models at almost no penalty in inference time, comparing with a finite Gaussian mixture model.

An important question is how can the Dirichlet process use an infinite, unbounded number of clusters and still be consistent. While a full explanation doesn't fit this manual, one can think of its chinese restaurant process analogy to help understanding it. The chinese restaurant process is a generative story for the Dirichlet process. Imagine a chinese restaurant with an infinite number of tables, at first all empty. When the first customer of the day arrives, he sits at the first table. Every following customer will then either sit on an occupied table with probability proportional to the number of customers in that table or sit in an entirely new table with probability proportional to the concentration parameter *alpha*. After a finite number of customers has sat, it is easy to see that only finitely many of the infinite tables will ever be used, and the higher the value of alpha the more total tables will be used. So the Dirichlet process does clustering with an unbounded number of mixture components by assuming a very asymmetrical prior structure over the assignments of points to components that is very concentrated (this property is known as rich-get-richer, as the full tables in the Chinese restaurant process only tend to get fuller as the simulation progresses).

Variational inference techniques for the Dirichlet process still work with a finite approximation to this infinite mixture model, but instead of having to specify a priori how many components one wants to use, one just specifies the concentration parameter and an upper bound on the number of mixture components (this upper bound, assuming it is higher than the "true" number of components, affects only algorithmic complexity, not the actual number of components used).

Derivation:

- See here the full derivation of this algorithm.

3.1. Cross-validation: evaluating estimator performance

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called **overfitting**. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a **test set** `X_test, y_test`. Note that the word "experiment" is not intended to denote academic use only, because even in commercial settings machine learning usually starts out experimentally.

In scikit-learn a random split into training and test sets can be quickly computed with the `train_test_split` helper function. Let's load the iris data set to fit a linear support vector machine on it:

```
>>> import numpy as np
>>> from sklearn import cross_validation
>>> from sklearn import datasets
>>> from sklearn import svm

>>> iris = datasets.load_iris()
>>> iris.data.shape, iris.target.shape
((150, 4), (150,))
```

We can now quickly sample a training set while holding out 40% of the data for testing (evaluating) our classifier:

```
>>> X_train, X_test, y_train, y_test = cross_validation.train_test_split(
...     iris.data, iris.target, test_size=0.4, random_state=0)

>>> X_train.shape, y_train.shape
((90, 4), (90,))
>>> X_test.shape, y_test.shape
((60, 4), (60,))

>>> clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.96...
```

When evaluating different settings ("hyperparameters") for estimators, such as the `C` setting that must be manually set for an SVM, there is still a risk of overfitting *on the test set* because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set can "leak" into the model and evaluation metrics no longer report on generalization performance. To solve this problem, yet another part of the dataset can be held out as a so-called "validation set": training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

A solution to this problem is a procedure called cross-validation (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach,

called *k*-fold CV, the training set is split into *k* smaller sets (other approaches are described below, but generally follow the same principles). The following procedure is followed for each of the *k* “folds”:

- A model is trained using $k - 1$ of the folds as training data;
- the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

The performance measure reported by *k*-fold cross-validation is then the average of the values computed in the loop. 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.

3.1.1. Computing cross-validated metrics

The simplest way to use cross-validation is to call the **`cross_val_score`** helper function on the estimator and the dataset.

The following example demonstrates how to estimate the accuracy of a linear kernel support vector machine on the iris dataset by splitting the data, fitting a model and computing the score 5 consecutive times (with different splits each time):

```
>>> clf = svm.SVC(kernel='linear', C=1)
>>> scores = cross_validation.cross_val_score(
...     clf, iris.data, iris.target, cv=5)
...
>>> scores
array([ 0.96...,  1.  ...,  0.96...,  0.96...,  1.        ])
```

The mean score and the standard deviation of the score estimate are hence given by:

```
>>> print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
Accuracy: 0.98 (+/- 0.03)
```

By default, the score computed at each CV iteration is the `score` method of the estimator. It is possible to change this by using the scoring parameter:

```
>>> from sklearn import metrics
>>> scores = cross_validation.cross_val_score(clf, iris.data, iris.target,
...     cv=5, scoring='f1')
>>> scores
array([ 0.96...,  1.  ...,  0.96...,  0.96...,  1.        ])
```

See *The scoring parameter: defining model evaluation rules* for details. In the case of the Iris dataset, the samples are balanced across target classes hence the accuracy and the F1-score are almost equal.

When the `cv` argument is an integer, **`cross_val_score`** uses the **`KFold`** or **`StratifiedKFold`** strategies by default (depending on the absence or presence of the target array).

It is also possible to use other cross validation strategies by passing a cross validation iterator instead, for instance:

```
>>> n_samples = iris.data.shape[0]
>>> cv = cross_validation.ShuffleSplit(n_samples, n_iter=3,
...     test_size=0.3, random_state=0)

>>> cross_validation.cross_val_score(clf, iris.data, iris.target, cv=cv)
...
array([ 0.97...,  0.97...,  1.        ])
```

The available cross validation iterators are introduced in the following.

Examples

- *Receiver Operating Characteristic (ROC) with cross validation*,
- *Recursive feature elimination with cross-validation*,
- *Parameter estimation using grid search with cross-validation*,
- *Sample pipeline for text feature extraction and evaluation*,

3.1.2. Cross validation iterators

The following sections list utilities to generate indices that can be used to generate dataset splits according to different cross validation strategies.

3.1.2.1. K-fold

`KFold` divides all the samples in k groups of samples, called folds (if $k = n$, this is equivalent to the *Leave One Out* strategy), 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 cross-validation on a dataset with 4 samples:

```
>>> import numpy as np
>>> from sklearn.cross_validation import KFold

>>> kf = KFold(4, n_folds=2)
>>> for train, test in kf:
...     print("%s %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```

Each fold is constituted 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 numpy indexing:

```
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> y = np.array([0, 1, 0, 1])
>>> X_train, X_test, y_train, y_test = X[train], X[test], y[train], y[test]
```

3.1.2.2. Stratified k-fold

`StratifiedKFold` is a variation of *k-fold* which returns *stratified* folds: each set contains approximately the same percentage of samples of each target class as the complete set.

Example of stratified 2-fold cross-validation on a dataset with 10 samples from two slightly unbalanced classes:

```
>>> from sklearn.cross_validation import StratifiedKFold

>>> labels = [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
>>> skf = StratifiedKFold(labels, 3)
>>> for train, test in skf:
...     print("%s %s" % (train, test))
[2 3 6 7 8 9] [0 1 4 5]
[0 1 3 4 5 8 9] [2 6 7]
[0 1 2 4 5 6 7] [3 8 9]
```

3.1.2.3. Leave-One-Out - LOO

`LeaveOneOut` (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 training sets and n different tests set. This cross-validation procedure does not waste much data as only one sample is removed from the training set:

```
>>> from sklearn.cross_validation import LeaveOneOut

>>> loo = LeaveOneOut(4)
>>> for train, test in loo:
...     print("%s %s" % (train, test))
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

Potential users of LOO for model selection should weigh a few known caveats. When compared with k-fold cross validation, one builds n models from n samples instead of k models, where $n > k$. Moreover, each is trained on $n - 1$ samples rather than $(k - 1)n/k$. In both ways, assuming k is not too large and $k < n$, LOO is more computationally expensive than k-fold cross validation.

In terms of accuracy, LOO often results in high variance as an estimator for the test error. Intuitively, since $n - 1$ of the n samples are used to build each model, models constructed from folds are virtually identical to each other and to the model built from the entire training set.

However, if the learning curve is steep for the training size in question, then 5- or 10- fold cross validation can overestimate the generalization error.

As a general rule, most authors, and empirical evidence, suggest that 5- or 10- fold cross validation should be preferred to LOO.

References:

- http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-12.html
- T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, Springer 2009
- L. Breiman, P. Spector Submodel selection and evaluation in regression: The X-random case, International Statistical Review 1992
- R. Kohavi, A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection, Intl. Jnt. Conf. AI
- R. Bharat Rao, G. Fung, R. Rosales, On the Dangers of Cross-Validation. An Experimental Evaluation, SIAM 2008
- G. James, D. Witten, T. Hastie, R Tibshirani, An Introduction to Statistical Learning, Springer 2013

3.1.2.4. Leave-P-Out - LPO

`LeavePOut` is very similar to `LeaveOneOut` as it creates all the possible training/test sets by removing p samples from the complete set. For n samples, this produces $\binom{n}{p}$ train-test pairs. Unlike `LeaveOneOut` and `KFold`, the test sets will overlap for $p > 1$.

Example of Leave-2-Out on a dataset with 4 samples:

```
>>> from sklearn.cross_validation import LeavePOut
```

```
>>> lpo = LeavePOut(4, p=2)
>>> for train, test in lpo:
...     print("%s %s" % (train, test))
[2 3] [0 1]
[1 3] [0 2]
[1 2] [0 3]
[0 3] [1 2]
[0 2] [1 3]
[0 1] [2 3]
```

3.1.2.5. Leave-One-Label-Out - LOLO

`LeaveOneLabelOut` (LOLO) is a cross-validation scheme which holds out the samples according to a third-party provided array of integer labels. This label information can be used to encode arbitrary domain specific pre-defined cross-validation folds.

Each training set is thus constituted 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 sklearn.cross_validation import LeaveOneLabelOut

>>> labels = [1, 1, 2, 2]
>>> lolo = LeaveOneLabelOut(labels)
>>> for train, test in lolo:
...     print("%s %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```

Another common application is to use time information: for instance the labels could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

Warning: Contrary to `StratifiedKFold`, **the ``labels`` of :class:`LeaveOneLabelOut` should not encode the target class to predict**: the goal of `StratifiedKFold` is to rebalance dataset classes across the train / test split to ensure that the train and test folds have approximately the same percentage of samples of each class while `LeaveOneLabelOut` will do the opposite by ensuring that the samples of the train and test fold will not share the same label value.

3.1.2.6. Leave-P-Label-Out

`LeavePLabelOut` 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 sklearn.cross_validation import LeavePLabelOut

>>> labels = [1, 1, 2, 2, 3, 3]
>>> lplo = LeavePLabelOut(labels, p=2)
>>> for train, test in lplo:
...     print("%s %s" % (train, test))
[4 5] [0 1 2 3]
[2 3] [0 1 4 5]
[0 1] [2 3 4 5]
```

3.1.2.7. Random permutations cross-validation a.k.a. Shuffle & Split

ShuffleSplit

The **ShuffleSplit** iterator will generate a user defined number of independent train / test dataset splits. Samples are first shuffled and then split into a pair of train and test sets.

It is possible to control the randomness for reproducibility of the results by explicitly seeding the `random_state` pseudo random number generator.

Here is a usage example:

```
>>> ss = cross_validation.ShuffleSplit(5, n_iter=3, test_size=0.25,
...     random_state=0)
>>> for train_index, test_index in ss:
...     print("%s %s" % (train_index, test_index))
...
[1 3 4] [2 0]
[1 4 3] [0 2]
[4 0 2] [1 3]
```

ShuffleSplit is thus a good alternative to **KFold** cross validation that allows a finer control on the number of iterations and the proportion of samples in on each side of the train / test split.

3.1.2.8. See also

StratifiedShuffleSplit is a variation of *ShuffleSplit*, which returns stratified splits, *i.e* which creates splits by preserving the same percentage for each target class as in the complete set.

3.1.3. A note on shuffling

If the data ordering is not arbitrary (e.g. samples with the same label are contiguous), shuffling it first may be essential to get a meaningful cross- validation result. However, the opposite may be true if the samples are not independently and identically distributed. For example, if samples correspond to news articles, and are ordered by their time of publication, then shuffling the data will likely lead to a model that is overfit and an inflated validation score: it will be tested on samples that are artificially similar (close in time) to training samples.

Some cross validation iterators, such as **KFold**, have an inbuilt option to shuffle the data indices before splitting them. Note that:

- This consumes less memory than shuffling the data directly.
- By default no shuffling occurs, including for the (stratified) K fold cross- validation performed by specifying `cv=some_integer` to **cross_val_score**, grid search, etc. Keep in mind that **train_test_split** still returns a random split.
- The `random_state` parameter defaults to `None`, meaning that the shuffling will be different every time `KFold(..., shuffle=True)` is iterated. However, `GridSearchCV` will use the same shuffling for each set of parameters validated by a single call to its `fit` method.
- To ensure results are repeatable (*on the same platform*), use a fixed value for `random_state`.

3.1.4. Cross validation and model selection

Cross validation iterators can also be used to directly perform model selection using Grid Search for the

optimal hyperparameters of the model. This is the topic if the next section: *Grid Search: Searching for estimator parameters*.

Previous Next

3.2. Grid Search: Searching for estimator parameters

Parameters that are not directly learnt within estimators can be set by searching a parameter space for the best *Cross-validation: evaluating estimator performance* score. Typical examples include `C`, `kernel` and `gamma` for Support Vector Classifier, `alpha` for Lasso, etc.

Any parameter provided when constructing an estimator may be optimized in this manner. Specifically, to find the names and current values for all parameters for a given estimator, use:

```
estimator.get_params()
```

Such parameters are often referred to as *hyperparameters* (particularly in Bayesian learning), distinguishing them from the parameters optimised in a machine learning procedure.

A search consists of:

- an estimator (regressor or classifier such as `sklearn.svm.SVC()`);
- a parameter space;
- a method for searching or sampling candidates;
- a cross-validation scheme; and
- a *score function*.

Two generic approaches to sampling search candidates are provided in scikit-learn: for given values, **`GridSearchCV`** exhaustively considers all parameter combinations, while **`RandomizedSearchCV`** can sample a given number of candidates from a parameter space with a specified distribution.

3.2.1. Exhaustive Grid Search

The grid search provided by **`GridSearchCV`** exhaustively generates candidates from a grid of parameter values specified with the `param_grid` parameter. For instance, the following `param_grid`:

```
param_grid = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
  {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']},
 ]
```

specifies that two grids should be explored: one with a linear kernel and C values in [1, 10, 100, 1000], and the second one with an RBF kernel, and the cross-product of C values ranging in [1, 10, 100, 1000] and gamma values in [0.001, 0.0001].

The **`GridSearchCV`** instance implements the usual estimator API: when “fitting” it on a dataset all the possible combinations of parameter values are evaluated and the best combination is retained.

Model selection: development and evaluation

Model selection with `GridSearchCV` can be seen as a way to use the labeled data to "train" the parameters of the grid.

When evaluating the resulting model it is important to do it on held-out samples that were not seen during the grid search process: it is recommended to split the data into a **development set** (to be fed to the `GridSearchCV` instance) and an **evaluation set** to compute performance metrics.

This can be done by using the **`cross_validation.train_test_split`** utility function.

3.2.1.1. Scoring functions for parameter search

By default, **`GridSearchCV`** uses the `score` function of the estimator to evaluate a parameter setting. These are the **`sklearn.metrics.accuracy_score`** for classification and **`sklearn.metrics.r2_score`** for regression. For some applications, other scoring functions are better suited (for example in unbalanced classification, the accuracy score is often uninformative). An alternative scoring function can be specified via the `scoring` parameter to **`GridSearchCV`**. See *The scoring parameter: defining model evaluation rules* for more details.

Examples:

- See *Parameter estimation using grid search with 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) using a **`pipeline.Pipeline`** instance.

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

3.2.2. Randomized Parameter Optimization

While using a grid of parameter settings is currently the most widely used method for parameter optimization, other search methods have more favourable properties. **`RandomizedSearchCV`** implements a randomized search over parameters, where each setting is sampled from a distribution over possible parameter values. This has two main benefits over an exhaustive search:

- A budget can be chosen independent of the number of parameters and possible values.
- Adding parameters that do not influence the performance does not decrease efficiency.

Specifying how parameters should be sampled is done using a dictionary, very similar to specifying parameters for **`GridSearchCV`**. Additionally, a computation budget, being the number of sampled candidates or sampling iterations, is specified using the `n_iter` parameter. For each parameter, either a distribution over possible values or a list of discrete choices (which will be sampled uniformly) can be specified:

```
[{'C': scipy.stats.expon(scale=100), 'gamma': scipy.stats.expon(scale=.1),
  'kernel': ['rbf'], 'class_weight':['auto', None]}]
```

This example uses the `scipy.stats` module, which contains many useful distributions for sampling

parameters, such as `expon`, `gamma`, `uniform` or `randint`. In principle, any function can be passed that provides a `rvs` (random variate sample) method to sample a value. A call to the `rvs` function should provide independent random samples from possible parameter values on consecutive calls.

> **Warning:** The distributions in `scipy.stats` do not allow specifying a random state. Instead, they use the global numpy random state, that can be seeded via `np.random.seed` or set using `np.random.set_state`.

For continuous parameters, such as `C` above, it is important to specify a continuous distribution to take full advantage of the randomization. This way, increasing `n_iter` will always lead to a finer search.

Examples:

- *Comparing randomized search and grid search for hyperparameter estimation* compares the usage and efficiency of randomized search and grid search.

References:

- Bergstra, J. and Bengio, Y., Random search for hyper-parameter optimization, The Journal of Machine Learning Research (2012)

3.2.3. Alternatives to brute force parameter search

3.2.3.1. Model specific cross-validation

Some models can fit data for a range of value of some parameter almost as efficiently as fitting the estimator for a single value of the parameter. This feature can be leveraged to perform a more efficient cross-validation used for model selection of this parameter.

The most common parameter amenable to this strategy is the parameter encoding the strength of the regularizer. In this case we say that we compute the **regularization path** of the estimator.

Here is the list of such models:

`linear_model.RidgeCV`([alphas, ...])	Ridge regression with built-in cross-validation.
`linear_model.RidgeClassifierCV`([alphas, ...])	Ridge classifier with built-in cross-validation.
`linear_model.LarsCV`([fit_intercept, ...])	Cross-validated Least Angle Regression model
`linear_model.LassoLarsCV`([fit_intercept, ...])	Cross-validated Lasso, using the LARS algorithm
`linear_model.LassoCV`([eps, n_alphas, ...])	Lasso linear model with iterative fitting along a regularization path
`linear_model.ElasticNetCV`([l1_ratio, eps, ...])	Elastic Net model with iterative fitting along a regularization path

3.2.3.2. Information Criterion

Some models can offer an information-theoretic closed-form formula of the optimal estimate of the regularization parameter by computing a single regularization path (instead of several when using cross-validation).

Here is the list of models benefitting from the Aikike Information Criterion (AIC) or the Bayesian Information Criterion (BIC) for automated model selection:

`linear_model.LassoLarsIC`([criterion, ...])	Lasso model fit with Lars using BIC or AIC for model selection

3.2.3.3. Out of Bag Estimates

When using ensemble methods base upon bagging, i.e. generating new training sets using sampling with replacement, part of the training set remains unused. For each classifier in the ensemble, a different part of the training set is left out.

This left out portion can be used to estimate the generalization error without having to rely on a separate validation set. This estimate comes “for free” as no additional data is needed and can be used for model selection.

This is currently implemented in the following classes:

`ensemble.RandomForestClassifier`([...])	A random forest classifier.
`ensemble.RandomForestRegressor`([...])	A random forest regressor.
`ensemble.ExtraTreesClassifier`([...])	An extra-trees classifier.
`ensemble.ExtraTreesRegressor`([n_estimators, ...])	An extra-trees regressor.
`ensemble.GradientBoostingClassifier`([loss, ...])	Gradient Boosting for classification.
`ensemble.GradientBoostingRegressor`([loss, ...])	Gradient Boosting for regression.

Previous Next

3.3. Pipeline: chaining estimators

`Pipeline` can be used to chain multiple estimators into one. This is useful as there is often a fixed sequence of steps in processing the data, for example feature selection, normalization and classification. `Pipeline` serves two purposes here:

> **Convenience**: You only have to call `fit` and `predict` once on your data to fit a whole sequence of estimators.
>
> **Joint parameter selection**: You can *grid search* over parameters of all estimators in the pipeline at once.

All estimators in a pipeline, except the last one, must be transformers (i.e. must have a `transform` method). The last estimator may be any type (transformer, classifier, etc.).

3.3.1. Usage

The `Pipeline` is build using a list of `(key, value)` pairs, where the `key` a string containing the name you want to give this step and `value` is an estimator object:

```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.svm import SVC
>>> from sklearn.decomposition import PCA
>>> estimators = [('reduce_dim', PCA()), ('svm', SVC())]
>>> clf = Pipeline(estimators)
>>> clf
Pipeline(steps=[('reduce_dim', PCA(copy=True, n_components=None,
    whiten=False)), ('svm', SVC(C=1.0, cache_size=200, class_weight=None,
    coef0=0.0, degree=3, gamma=0.0, kernel='rbf', max_iter=-1,
    probability=False, random_state=None, shrinking=True, tol=0.001,
    verbose=False))])
```

The utility function `make_pipeline` is a shorthand for constructing pipelines; it takes a variable number of estimators and returns a pipeline, filling in the names automatically:

```
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.naive_bayes import MultinomialNB
>>> from sklearn.preprocessing import Binarizer
>>> make_pipeline(Binarizer(), MultinomialNB())
Pipeline(steps=[('binarizer', Binarizer(copy=True, threshold=0.0)),
                ('multinomialnb', MultinomialNB(alpha=1.0,
                                                class_prior=None,
                                                fit_prior=True))])
```

The estimators of a pipeline are stored as a list in the `steps` attribute:

```
>>> clf.steps[0]
('reduce_dim', PCA(copy=True, n_components=None, whiten=False))
```

and as a `dict` in `named_steps`:

```
>>> clf.named_steps['reduce_dim']
PCA(copy=True, n_components=None, whiten=False)
```

Parameters of the estimators in the pipeline can be accessed using the `<estimator>__<parameter>` syntax:

```
>>> clf.set_params(svm__C=10)
Pipeline(steps=[('reduce_dim', PCA(copy=True, n_components=None,
    whiten=False)), ('svm', SVC(C=10, cache_size=200, class_weight=None,
    coef0=0.0, degree=3, gamma=0.0, kernel='rbf', max_iter=-1,
    probability=False, random_state=None, shrinking=True, tol=0.001,
    verbose=False))])
```

This is particularly important for doing grid searches:

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = dict(reduce_dim__n_components=[2, 5, 10],
...               svm__C=[0.1, 10, 100])
>>> grid_search = GridSearchCV(clf, param_grid=params)
```

Examples:

- *Pipeline Anova SVM*
- *Sample pipeline for text feature extraction and evaluation*
- *Pipelining: chaining a PCA and a logistic regression*
- *Explicit feature map approximation for RBF kernels*
- *SVM-Anova: SVM with univariate feature selection*

3.3.2. Notes

Calling `fit` on the pipeline is the same as calling `fit` on each estimator in turn, `transform` the input and pass it on to the next step. The pipeline has all the methods that the last estimator in the pipeline has, i.e. if the last estimator is a classifier, the **`Pipeline`** can be used as a classifier. If the last estimator is a transformer, again, so is the pipeline.

3.4. FeatureUnion: Combining feature extractors

`FeatureUnion` combines several transformer objects into a new transformer that combines their output. A **`FeatureUnion`** takes a list of transformer objects. During fitting, each of these is fit to the data independently. For transforming data, the transformers are applied in parallel, and the sample vectors they output are concatenated end-to-end into larger vectors.

`FeatureUnion` serves the same purposes as **`Pipeline`** - convenience and joint parameter estimation and validation.

`FeatureUnion` and **`Pipeline`** can be combined to create complex models.

(A **`FeatureUnion`** has no way of checking whether two transformers might produce identical features. It only produces a union when the feature sets are disjoint, and making sure they are is the caller's responsibility.)

3.4.1. Usage

A **`FeatureUnion`** is built using a list of `(key, value)` pairs, where the `key` is the name you want to give to a given transformation (an arbitrary string; it only serves as an identifier) and `value` is an estimator object:

```
>>> from sklearn.pipeline import FeatureUnion
>>> from sklearn.decomposition import PCA
>>> from sklearn.decomposition import KernelPCA
>>> estimators = [('linear_pca', PCA()), ('kernel_pca', KernelPCA())]
>>> combined = FeatureUnion(estimators)
>>> combined
FeatureUnion(n_jobs=1, transformer_list=[('linear_pca', PCA(copy=True,
    n_components=None, whiten=False)), ('kernel_pca', KernelPCA(alpha=1.0,
    coef0=1, degree=3, eigen_solver='auto', fit_inverse_transform=False,
    gamma=None, kernel='linear', kernel_params=None, max_iter=None,
    n_components=None, remove_zero_eig=False, tol=0))],
    transformer_weights=None)
```

Like pipelines, feature unions have a shorthand constructor called `make_union` that does require manual naming of the components.

Examples:

- *Concatenating multiple feature extraction methods*

Previous Next

3.5. Model evaluation: quantifying the quality of predictions

There are 3 different approaches to evaluate the quality of predictions of a model:

- **Estimator score method**: Estimators have a `score` method providing a default evaluation criterion for the problem they are designed to solve. This is not discussed on this page, but in each estimator's documentation.
- **Scoring parameter**: Model-evaluation tools using *cross-validation* (such as `cross_validation.cross_val_score` and `grid_search.GridSearchCV`) rely on an internal *scoring* strategy. This is discussed on section *The scoring parameter: defining model evaluation rules*.
- **Metric functions**: The `metrics` module implements functions assessing prediction errors for specific purposes. This is discussed in the section *Function for prediction-error metrics*.

Finally, *Dummy estimators* are useful to get a baseline value of those metrics for random predictions.

See also: For "pairwise" metrics, between *samples* and not estimators or predictions, see the *Pairwise metrics, Affinities and Kernels* section.

3.5.1. The `scoring` parameter: defining model evaluation rules

Model selection and evaluation using tools, such as `grid_search.GridSearchCV` and `cross_validation.cross_val_score`, take a `scoring` parameter that controls what metric they apply to estimators evaluated.

3.5.1.1. Common cases: predefined values

For the most common usecases, you can simply provide a string as the `scoring` parameter. Possible values are:

Scoring	Function
Classification	
'accuracy'	`sklearn.metrics.accuracy_score`
'average_precision'	`sklearn.metrics.average_precision_score`
'f1'	`sklearn.metrics.f1_score`
'precision'	`sklearn.metrics.precision_score`
'recall'	`sklearn.metrics.recall_score`
'roc_auc'	`sklearn.metrics.roc_auc_score`
Clustering	
'adjusted_rand_score'	`sklearn.metrics.adjusted_rand_score`
Regression	
'mean_absolute_error'	`sklearn.metrics.mean_absolute_error`
'mean_squared_error'	`sklearn.metrics.mean_squared_error`
'r2'	`sklearn.metrics.r2_score`

Setting the `scoring` parameter to a wrong value should give you a list of acceptable values:

```
>>> from sklearn import svm, cross_validation, datasets
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> model = svm.SVC()
>>> cross_validation.cross_val_score(model, X, y, scoring='wrong_choice')
Traceback (most recent call last):
ValueError: 'wrong_choice' is not a valid scoring value. Valid options are ['accuracy', 'adjusted
```

Note: The corresponding scorer objects are stored in the dictionary `sklearn.metrics.SCORERS`.

The above choices correspond to error-metric functions that can be applied to predicted values. These are detailed below, in the next sections.

3.5.1.2. Defining your scoring strategy from score functions

The scoring parameter can be a callable that takes model predictions and ground truth.

However, if you want to use a scoring function that takes additional parameters, such as **`fbeta_score`**, you need to generate an appropriate scoring object. The simplest way to generate a callable object for scoring is by using **`make_scorer`**. That function converts score functions (discussed below in *Function for prediction-error metrics*) into callables that can be used for model evaluation.

One typical use case is to wrap an existing scoring function from the library with non default value for its parameters such as the `beta` parameter for the **`fbeta_score`** function:

```
>>> from sklearn.metrics import fbeta_score, make_scorer
>>> ftwo_scorer = make_scorer(fbeta_score, beta=2)
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> grid = GridSearchCV(LinearSVC(), param_grid={'C': [1, 10]}, scoring=ftwo_scorer)
```

The second use case is to build a completely new and custom scorer object from a simple python function:

```
>>> def my_custom_loss_func(ground_truth, predictions):
...     diff = np.abs(ground_truth - predictions).max()
...     return np.log(1 + diff)
...
>>> my_custom_scorer = make_scorer(my_custom_loss_func, greater_is_better=False)
>>> grid = GridSearchCV(LinearSVC(), param_grid={'C': [1, 10]}, scoring=my_custom_scorer)
```

`make_scorer` takes as parameters:

- the function you want to use
- whether it is a score (`greater_is_better=True`) or a loss (`greater_is_better=False`),
- whether the function you provided takes predictions as input (`needs_threshold=False`) or needs confidence scores (`needs_threshold=True`)
- any additional parameters, such as `beta` in an **`f1_score`**.

3.5.1.3. Implementing your own scoring object

You can generate even more flexible model scores by constructing your own scoring object from scratch, without using the **`make_scorer`** factory. For a callable to be a scorer, it needs to meet the protocol specified by the following two rules:

- It can be called with parameters `(estimator, X, y)`, where `estimator` is the model that should be evaluated, `X` is validation data, and `y` is the ground truth target for `X` (in the supervised case) or `None` (in the unsupervised case).
- It returns a floating point number that quantifies the quality of `estimator`'s predictions on `X` which reference to `y`. Again, higher numbers are better.

3.5.2. Function for prediction-error metrics

The module **`sklearn.metric`** also exposes a set of simple functions measuring a prediction error given ground truth and prediction:

- functions ending with `_score` return a value to maximize (the higher the better).
- functions ending with `_error` or `_loss` return a value to minimize (the lower the better).

3.5.2.1. Classification metrics

The **`sklearn.metrics`** implements several losses, scores and utility functions to measure classification performance. Some metrics might require probability estimates of the positive class, confidence values or binary decisions values.

Some of these are restricted to the binary classification case:

`hinge_loss`(y_true, pred_decision[, ...])	Average hinge loss (non-regularized)
`matthews_corrcoef`(y_true, y_pred)	Compute the Matthews correlation coefficient (MCC) for binary classes
`precision_recall_curve`(y_true, probas_pred)	Compute precision-recall pairs for different probability thresholds
`roc_curve`(y_true, y_score[, pos_label, ...])	Compute Receiver operating characteristic (ROC)

Others also work in the multiclass case:

`confusion_matrix`(y_true, y_pred[, labels])	Compute confusion matrix to evaluate the accuracy of a classification

And some also work in the multilabel case:

`accuracy_score`(y_true, y_pred[, normalize, ...])	Accuracy classification score.
`classification_report`(y_true, y_pred[, ...])	Build a text report showing the main classification metrics
`f1_score`(y_true, y_pred[, labels, ...])	Compute the F1 score, also known as balanced F-score or F-measure
`fbeta_score`(y_true, y_pred, beta[, labels, ...])	Compute the F-beta score
`hamming_loss`(y_true, y_pred[, classes])	Compute the average Hamming loss.
`jaccard_similarity_score`(y_true, y_pred[, ...])	Jaccard similarity coefficient score
`log_loss`(y_true, y_pred[, eps, normalize])	Log loss, aka logistic loss or cross-entropy loss.
`precision_recall_fscore_support`(y_true, y_pred)	Compute precision, recall, F-measure and support for each class
`precision_score`(y_true, y_pred[, labels, ...])	Compute the precision
`recall_score`(y_true, y_pred[, labels, ...])	Compute the recall
`zero_one_loss`(y_true, y_pred[, normalize, ...])	Zero-one classification loss.

And some work with binary and multilabel indicator format:

`average_precision_score`(y_true, y_score[, ...])	Compute average precision (AP) from prediction scores

`roc_auc_score`(y_true, y_score[, average, ...])	Compute Area Under the Curve (AUC) from prediction scores

In the following sub-sections, we will describe each of those functions.

3.5.2.1.1. Accuracy score

The `accuracy_score` function computes the accuracy, the fraction (default) or the number of correct predictions.

In multilabel classification, the function returns the subset accuracy: if the entire set of predicted labels for a sample strictly match with the true set of labels, then the subset accuracy is 1.0, otherwise it is 0.0.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the fraction of correct predictions over n_{samples} is defined as

$$\texttt{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y}_i = y_i)$$

where $1(x)$ is the indicator function.

```
>>> import numpy as np
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
0.5
>>> accuracy_score(y_true, y_pred, normalize=False)
2
```

In the multilabel case with binary label indicators:

```
>>> accuracy_score(np.array([[0.0, 1.0], [1.0, 1.0]]), np.ones((2, 2)))
0.5
```

Example:

- See *Test with permutations the significance of a classification score* for an example of accuracy score usage using permutations of the dataset.

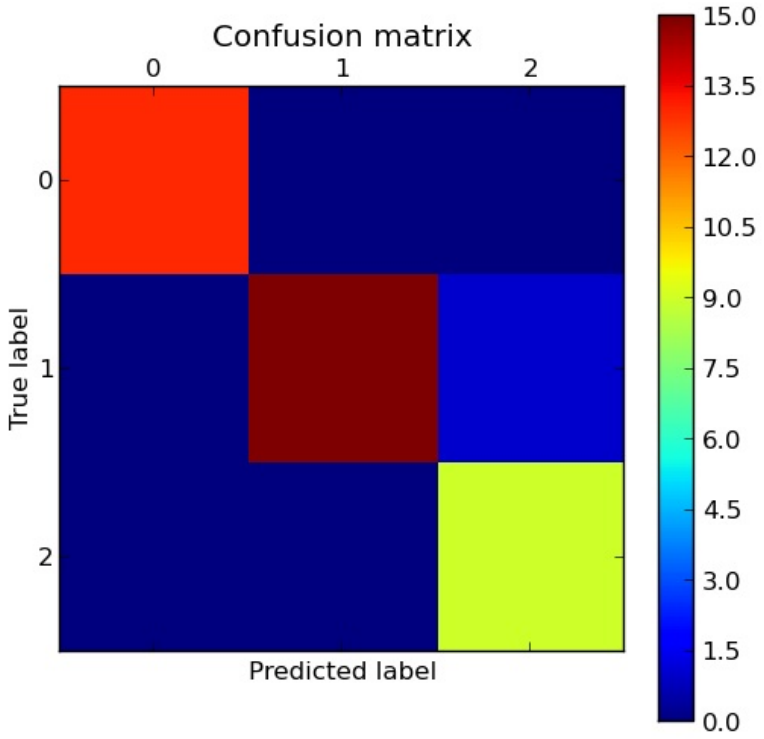
3.5.2.1.2. Confusion matrix

The `confusion_matrix` function computes the confusion matrix to evaluate the accuracy on a classification problem.

By definition, a confusion matrix C is such that $C_{i,j}$ is equal to the number of observations known to be in group i but predicted to be in group j. Here an example of such confusion matrix:

```
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

Here a visual representation of such confusion matrix (this figure comes from the *Confusion matrix* example):



Example:

- See *Confusion matrix* for an example of confusion matrix usage to evaluate the quality of the output of a classifier.
- See *Recognizing hand-written digits* for an example of confusion matrix usage in the classification of hand-written digits.
- See *Classification of text documents using sparse features* for an example of confusion matrix usage in the classification of text documents.

3.5.2.1.3. Classification report

The `classification_report` function builds a text report showing the main classification metrics. Here a small example with custom `target_names` and inferred labels:

```
>>> from sklearn.metrics import classification_report
>>> y_true = [0, 1, 2, 2, 0]
>>> y_pred = [0, 0, 2, 2, 0]
>>> target_names = ['class 0', 'class 1', 'class 2']
>>> print(classification_report(y_true, y_pred, target_names=target_names))
             precision    recall  f1-score   support

    class 0       0.67      1.00      0.80         2
    class 1       0.00      0.00      0.00         1
    class 2       1.00      1.00      1.00         2

avg / total       0.67      0.80      0.72         5
```

Example:

- See *Recognizing hand-written digits* for an example of classification report usage in the classification of the hand-written digits.

- See *Classification of text documents using sparse features* for an example of classification report usage in the classification of text documents.
- See *Parameter estimation using grid search with cross-validation* for an example of classification report usage in parameter estimation using grid search with a nested cross-validation.

3.5.2.1.4. Hamming loss

The **`hamming_loss`** computes the average Hamming loss or Hamming distance between two sets of samples.

If $\hat{y}_j$ is the predicted value for the j-th labels of a given sample, y_j is the corresponding true value and n_{labels} is the number of class or labels, then the Hamming loss $L_{Hamming}$ between two samples is defined as:

$$L_{Hamming}(y, \hat{y}) = \frac{1}{n_{\text{labels}}} \sum_{j=0}^{n_{\text{labels}}-1} 1(\hat{y}_j \neq y_j)$$

where $1(x)$ is the indicator function.

```
>>> from sklearn.metrics import hamming_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> hamming_loss(y_true, y_pred)
0.25
```

In the multilabel case with binary label indicators:

```
>>> hamming_loss(np.array([[0.0, 1.0], [1.0, 1.0]]), np.zeros((2, 2)))
0.75
```

Note: In multiclass classification, the Hamming loss correspond to the Hamming distance between `y_true` and `y_pred` which is equivalent to the *Zero one loss* function.

In multilabel classification, the Hamming loss is different from the zero-one loss. The zero-one loss penalizes any predictions that don't exactly match the true required set of labels, while Hamming loss will penalize the individual labels. So, predicting a subset or superset of the true labels will give a Hamming loss strictly between zero and one.

The Hamming loss is upperbounded by the zero-one loss. When normalized over samples, the Hamming loss is always between zero and one.

3.5.2.1.5. Jaccard similarity coefficient score

The **`jaccard_similarity_score`** function computes the average (default) or sum of Jaccard similarity coefficients, also called Jaccard index, between pairs of label sets.

The Jaccard similarity coefficient of the i-th samples with a ground truth label set y_i and a predicted label set $\hat{y}_i$ is defined as

$$J(y_i, \hat{y}_i) = \frac{|y_i \cap \hat{y}_i|}{|y_i \cup \hat{y}_i|}.$$

In binary and multiclass classification, the Jaccard similarity coefficient score is equal to the classification accuracy.

```
>>> import numpy as np
>>> from sklearn.metrics import jaccard_similarity_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> jaccard_similarity_score(y_true, y_pred)
0.5
>>> jaccard_similarity_score(y_true, y_pred, normalize=False)
2
```

In the multilabel case with binary label indicators:

```
>>> jaccard_similarity_score(np.array([[0.0, 1.0], [1.0, 1.0]]), np.ones((2, 2)))
0.75
```

3.5.2.1.6. Precision, recall and F-measures

The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is intuitively the ability of the classifier to find all the positive samples.

The F-measure (F_β and F_1 measures) can be interpreted as a weighted harmonic mean of the precision and recall. A F_β measure reaches its best value at 1 and worst score at 0. With $\beta = 1$, the F_β measure leads to the F_1 measure, wheres the recall and the precision are equally important.

The **precision_recall_curve** computes a precision-recall curve from the ground truth label and a score given by the classifier by varying a decision threshold.

The **average_precision_score** function computes the average precision (AP) from prediction scores. This score corresponds to the area under the precision-recall curve.

Several functions allow you to analyze the precision, recall and F-measures score:

average_precision_score(y_true, y_score[, ...])	Compute average precision (AP) from prediction scores
f1_score(y_true, y_pred[, labels, ...])	Compute the F1 score, also known as balanced F-score or F-measure
fbeta_score(y_true, y_pred, beta[, labels, ...])	Compute the F-beta score
precision_recall_curve(y_true, probas_pred)	Compute precision-recall pairs for different probability thresholds
precision_recall_fscore_support(y_true, y_pred)	Compute precision, recall, F-measure and support for each class
precision_score(y_true, y_pred[, labels, ...])	Compute the precision
recall_score(y_true, y_pred[, labels, ...])	Compute the recall

Note that the **precision_recall_curve** function is restricted to the binary case. The **average_precision_score** function works only in binary classification and multilabel indicator format.

Examples:

- See *Classification of text documents using sparse features* for an example of **f1_score** usage with classification of text documents.
- See *Parameter estimation using grid search with cross-validation* for an example of **precision_score** and **recall_score** usage in parameter estimation using grid search with a nested cross-validation.

- See *Precision-Recall* for an example of precision-Recall metric to evaluate the quality of the output of a classifier with **precision_recall_curve**.
- See *Sparse recovery: feature selection for sparse linear models* for an example of **precision_recall_curve** usage in feature selection for sparse linear models.

3.5.2.1.6.1. Binary classification

In a binary classification task, the terms ''positive'' and ''negative'' refer to the classifier's prediction and the terms ''true'' and ''false'' refer to whether that prediction corresponds to the external judgment (sometimes known as the ''observation''). Given these definitions, we can formulate the following table:

	Actual class (observation)	
Predicted class (expectation)	tp (true positive) Correct result	fp (false positive) Unexpected result
	fn (false negative) Missing result	tn (true negative) Correct absence of result

In this context, we can define the notions of precision, recall and F-measure:

$$\text{precision} = \frac{tp}{tp + fp},$$

$$\text{recall} = \frac{tp}{tp + fn},$$

$$F_\beta = (1 + \beta^2) \frac{\text{precision} \times \text{recall}}{\beta^2 \text{precision} + \text{recall}}.$$

Here some small examples in binary classification:

```
>>> from sklearn import metrics
>>> y_pred = [0, 1, 0, 0]
>>> y_true = [0, 1, 0, 1]
>>> metrics.precision_score(y_true, y_pred)
1.0
>>> metrics.recall_score(y_true, y_pred)
0.5
>>> metrics.f1_score(y_true, y_pred)
0.66...
>>> metrics.fbeta_score(y_true, y_pred, beta=0.5)
0.83...
>>> metrics.fbeta_score(y_true, y_pred, beta=1)
0.66...
>>> metrics.fbeta_score(y_true, y_pred, beta=2)
0.55...
>>> metrics.precision_recall_fscore_support(y_true, y_pred, beta=0.5)
(array([ 0.66...,  1.        ]), array([ 1. ,  0.5]), array([ 0.71...,  0.83...]), array([2, 2]..


>>> import numpy as np
>>> from sklearn.metrics import precision_recall_curve
>>> from sklearn.metrics import average_precision_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> precision, recall, threshold = precision_recall_curve(y_true, y_scores)
>>> precision
array([ 0.66...,  0.5       ,  1.        ,  1.        ])
>>> recall
array([ 1. ,  0.5,  0.5,  0. ])
>>> threshold
array([ 0.35,  0.4 ,  0.8 ])
>>> average_precision_score(y_true, y_scores)
0.79...
```

3.5.2.1.6.2. Multiclass and multilabel classification

In multiclass and multilabel classification task, the notions of precision, recall and F-measures can be applied to each label independently. There are a few ways to combine results across labels, specified by the `average` argument to the **`average_precision_score`** (multilabel only), **`f1_score`**, **`fbeta_score`**, **`precision_recall_fscore_support`**, **`precision_score`** and **`recall_score`** functions:

- `"micro"`: calculate metrics globally by counting the total true positives, false negatives and false positives. Except in the multi-label case this implies that precision, recall and F are equal.
- `"samples"`: calculate metrics for each sample, comparing sets of labels assigned to each, and find the mean across all samples. This is only meaningful and available in the multilabel case.
- `"macro"`: calculate metrics for each label, and find their mean. This does not take label imbalance into account.
- `"weighted"`: calculate metrics for each label, and find their average weighted by the number of occurrences of the label in the true data. This alters `"macro"` to account for label imbalance; it may produce an F-score that is not between precision and recall.
- `None`: calculate metrics for each label and do not average them.

To make this more explicit, consider the following notation:

- y the set of *predicted* $(sample, label)$ pairs
- $\hat{y}$ the set of *true* $(sample, label)$ pairs
- L the set of labels
- S the set of samples
- y_s the subset of y with sample s, i.e. $y_s := \{(s', l) \in y | s' = s\}$
- y_l the subset of y with label l
- similarly, $\hat{y}_s$ and $\hat{y}_l$ are subsets of $\hat{y}$
- $P(A, B) := \frac{|A \cap B|}{|A|}$
- $R(A, B) := \frac{|A \cap B|}{|B|}$ (Conventions vary on handling $B = \emptyset$; this implementation uses $R(A, B) := 0$, and similar for P.)
- $F_\beta(A, B) := (1 + \beta^2) \frac{P(A,B) \times R(A,B)}{\beta^2 P(A,B) + R(A,B)}$

Then the metrics are defined as:

`average`	Precision	Recall	F_beta
`"micro"`	$P(y, \hat{y})$	$R(y, \hat{y})$	$F_\beta(y, \hat{y})$
`"samples"`	$\frac{1}{\lvert S \rvert} \sum_{s \in S} P(y_s, \hat{y}_s)$	$\frac{1}{\lvert S \rvert} \sum_{s \in S} R(y_s, \hat{y}_s)$	$\frac{1}{\lvert S \rvert} \sum_{s \in S} F_\beta(y_s, \hat{y}_s)$
`"macro"`	$\frac{1}{\lvert L \rvert} \sum_{l \in L} P(y_l, \hat{y}_l)$	$\frac{1}{\lvert L \rvert} \sum_{l \in L} R(y_l, \hat{y}_l)$	$\frac{1}{\lvert L \rvert} \sum_{l \in L} F_\beta(y_l, \hat{y}_l)$
`"weighted"`	$\frac{1}{\sum_{l \in L} \lvert \hat{y}_l \rvert} \sum_{l \in L} \lvert \hat{y}_l \rvert P(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l \in L} \lvert \hat{y}_l \rvert} \sum_{l \in L} \lvert \hat{y}_l \rvert R(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l \in L} \lvert \hat{y}_l \rvert} \sum_{l \in L} \lvert \hat{y}_l \rvert F_\beta(y_l, \hat{y}_l)$
`None`	$\langle P(y_l, \hat{y}_l) \mid l \in L \rangle$	$\langle R(y_l, \hat{y}_l) \mid l \in L \rangle$	$\langle F_\beta(y_l, \hat{y}_l) \mid l \in L \rangle$

```
>>> from sklearn import metrics
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> metrics.precision_score(y_true, y_pred, average='macro')
0.22...
>>> metrics.recall_score(y_true, y_pred, average='micro')
...
0.33...
>>> metrics.f1_score(y_true, y_pred, average='weighted')
0.26...
```

```
>>> metrics.fbeta_score(y_true, y_pred, average='macro', beta=0.5)
0.23...
>>> metrics.precision_recall_fscore_support(y_true, y_pred, beta=0.5, average=None)
...
(array([ 0.66...,  0.        ,  0.        ]), array([ 1.,  0.,  0.]), array([ 0.71...,  0.
```

3.5.2.1.7. Hinge loss

The **hinge_loss** function computes the average hinge loss function. The hinge loss is used in maximal margin classification as support vector machines.

If the labels are encoded with +1 and -1, y: is the true value and w is the predicted decisions as output by `decision_function`, then the hinge loss is defined as:

$$L_{\text{Hinge}}(y, w) = \max\{1 - wy, 0\} = |1 - wy|_+$$

Here a small example demonstrating the use of the **hinge_loss** function with a svm classifier:

```
>>> from sklearn import svm
>>> from sklearn.metrics import hinge_loss
>>> X = [[0], [1]]
>>> y = [-1, 1]
>>> est = svm.LinearSVC(random_state=0)
>>> est.fit(X, y)
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='l2', multi_class='ovr', penalty='l2',
     random_state=0, tol=0.0001, verbose=0)
>>> pred_decision = est.decision_function([[-2], [3], [0.5]])
>>> pred_decision
array([-2.18...,  2.36...,  0.09...])
>>> hinge_loss([-1, 1, 1], pred_decision)
0.3...
```

3.5.2.1.8. Log loss

The log loss, also called logistic regression loss or cross-entropy loss, is a loss function defined on probability estimates. It is commonly used in (multinomial) logistic regression and neural networks, as well as some variants of expectation-maximization, and can be used to evaluate the probability outputs (`predict_proba`) of a classifier, rather than its discrete predictions.

For binary classification with a true label $y \in \{0, 1\}$ and a probability estimate $p = \Pr(y = 1)$, the log loss per sample is the negative log-likelihood of the classifier given the true label:

$$L_{\log}(y, p) = -\log \Pr(y|p) = -(y \log p) + (1 - y) \log(1 - p))$$

This extends to the multiclass case as follows. Let the true labels for a set of samples be encoded as a 1-of-K binary indicator matrix Y, i.e. $y_{i,k} = 1$ if sample i has label k taken from a set of K labels. Let P be a matrix of probability estimates, with $p_{i,k} = \Pr(t_{i,k} = 1)$. Then the log loss of the whole set is

$$L_{\log}(Y, P) = -\log \Pr(Y|P) = -\frac{1}{N} \sum_{i=0}^{N-1} \sum_{k=0}^{K-1} y_{i,k} \log p_{i,k}$$

To see how this generalizes the binary log loss given above, note that in the binary case, $p_{i,0} = 1 - p_{i,1}$ and $y_{i,0} = 1 - y_{i,1}$, so expanding the inner sum over $y_{i,k} \in \{0, 1\}$ gives the binary log loss.

The function **log_loss** computes log loss given a list of ground-truth labels and a probability matrix, as

returned by an estimator's `predict_proba` method.

```
>>> from sklearn.metrics import log_loss
>>> y_true = [0, 0, 1, 1]
>>> y_pred = [[.9, .1], [.8, .2], [.3, .7], [.01, .99]]
>>> log_loss(y_true, y_pred)
0.1738...
```

The first `[.9, .1]` in `y_pred` denotes 90% probability that the first sample has label 0. The log loss is non-negative.

3.5.2.1.9. Matthews correlation coefficient

The **`matthews_corrcoef`** function computes the Matthew's correlation coefficient (MCC) for binary classes (quoting the Wikipedia article on the Matthew's correlation coefficient):

> "The Matthews correlation coefficient is used in machine learning as a measure of the quality of binary (two-class) classifications. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes. The MCC is in essence a correlation coefficient value between -1 and +1. A coefficient of +1 represents a perfect prediction, 0 an average random prediction and -1 an inverse prediction. The statistic is also known as the phi coefficient."

If tp, tn, fp and fn are respectively the number of true positives, true negatives, false positives ans false negatives, the MCC coefficient is defined as

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}.$$

Here a small example illustrating the usage of the **`matthews_corrcoef`** function:

```
>>> from sklearn.metrics import matthews_corrcoef
>>> y_true = [+1, +1, +1, -1]
>>> y_pred = [+1, -1, +1, +1]
>>> matthews_corrcoef(y_true, y_pred)
-0.33...
```

3.5.2.1.10. Receiver operating characteristic (ROC)

The function **`roc_curve`** computes the receiver operating characteristic curve, or ROC curve (quoting Wikipedia):

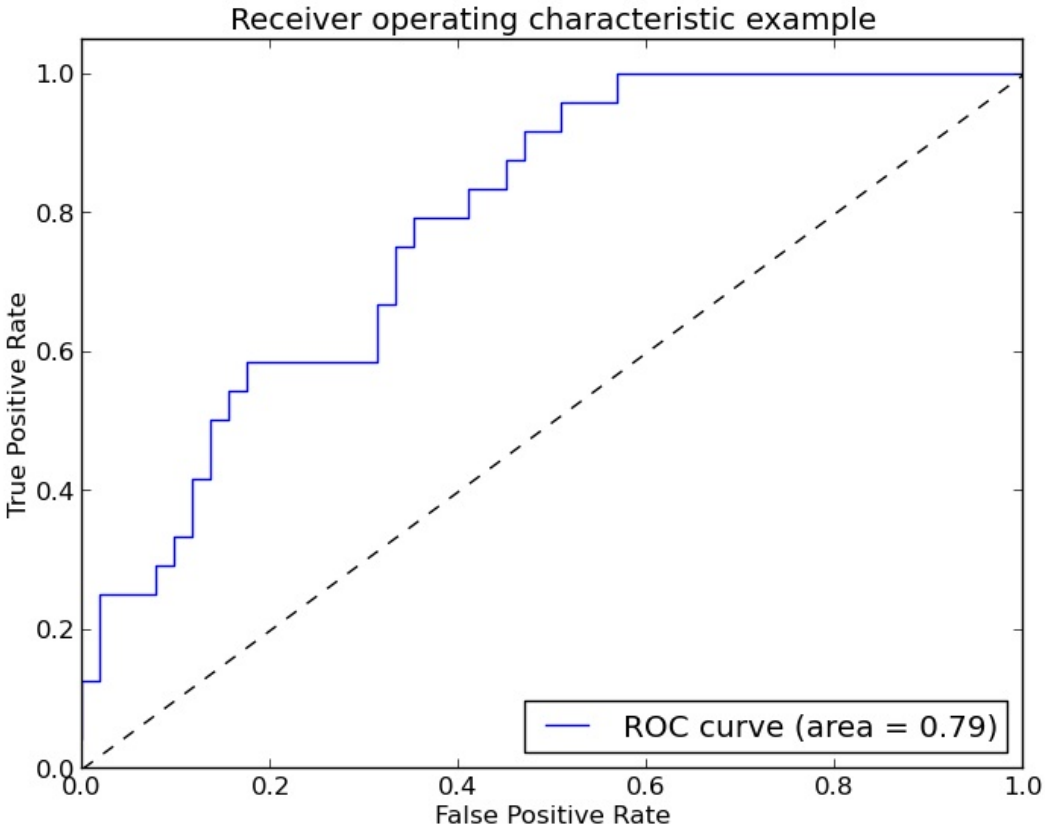
> "A receiver operating characteristic (ROC), or simply ROC curve, is a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied. It is created by plotting the fraction of true positives out of the positives (TPR = true positive rate) vs. the fraction of false positives out of the negatives (FPR = false positive rate), at various threshold settings. TPR is also known as sensitivity, and FPR is one minus the specificity or true negative rate."

This function requires the true binary value and the target scores, which can either be probability estimates of the positive class, confidence values, or binary decisions. Here a small example of how to use the **`roc_curve`** function:

```
>>> import numpy as np
>>> from sklearn.metrics import roc_curve
>>> y = np.array([1, 1, 2, 2])
>>> scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = roc_curve(y, scores, pos_label=2)
>>> fpr
array([ 0. ,  0.5,  0.5,  1. ])
```

```
>>> tpr
array([ 0.5,  0.5,  1. ,  1. ])
>>> thresholds
array([ 0.8 ,  0.4 ,  0.35,  0.1 ])
```

The following figure shows an example of such ROC curve.



The `roc_auc_score` function computes the area under the receiver operating characteristic (ROC) curve, which is also denoted by AUC or AUROC. By computing the area under the roc curve, the curve information is summarized in one number. For more information see the Wikipedia article on AUC.

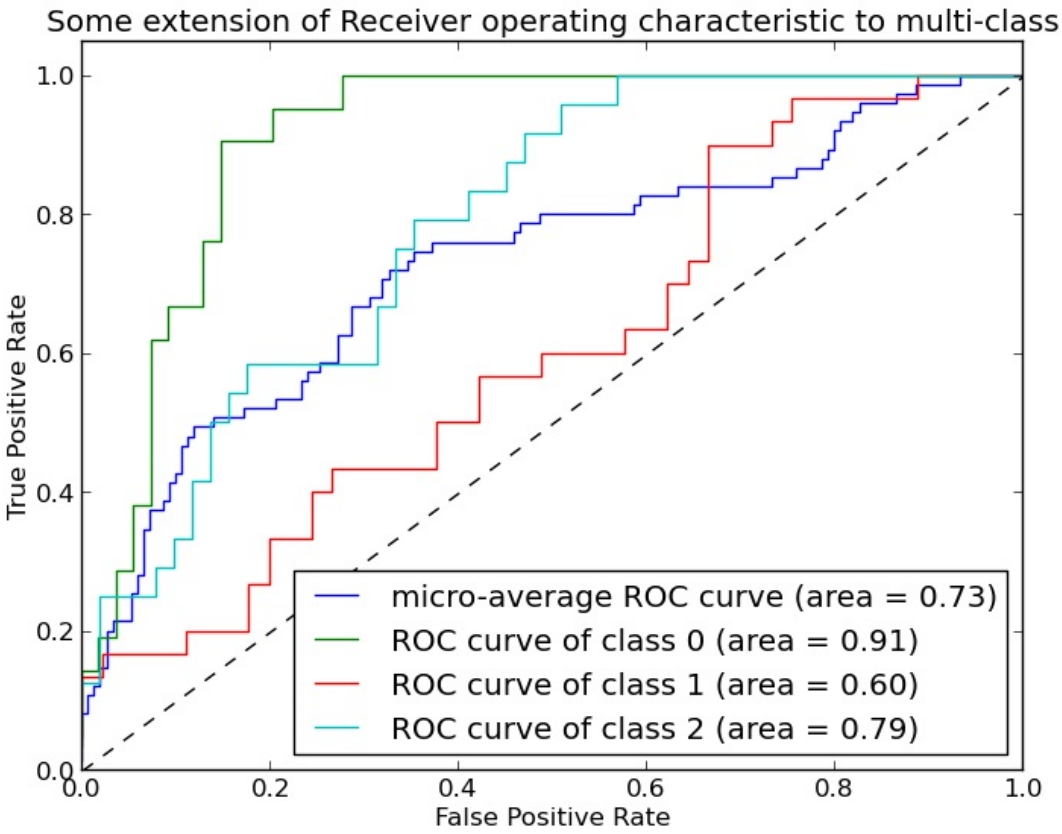
```
>>> import numpy as np
>>> from sklearn.metrics import roc_auc_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> roc_auc_score(y_true, y_scores)
0.75
```

In multi-label classification, the `roc_auc_score` function is extended by averaging over the labels:

- `"micro"`: computes the area under the ROC curve globally obtained by considering each element of the label indicator matrix as a label.
- `"samples"`: computes the area under the ROC curve on each sample, comparing the set of labels and scores assigned to each, and find the mean across all samples.
- `"macro"`: computes the area under the ROC curve for each label, and find their mean.
- `"weighted"`: computes the area under the ROC curve for each label, and find their average weighted by the number of occurrences of the label in the true data.
- `None`: this returns an array of scores with scores with shape (n_classes,) instead of an aggregate scalar score.

Compared to metrics such as the subset accuracy, the hamming loss or the F1 score, ROC AUC doesn't

require to optimize a threshold for each label. The `roc_auc_score` function can also be used in multi-class classification if predicted outputs have been binarized.



Examples:

- See *Receiver Operating Characteristic (ROC)* for an example of receiver operating characteristic (ROC) metric to evaluate the quality of the output of a classifier.
- See *Receiver Operating Characteristic (ROC) with cross validation* for an example of receiver operating characteristic (ROC) metric to evaluate the quality of the output of a classifier using cross-validation.
- See *Species distribution modeling* for an example of receiver operating characteristic (ROC) metric to model species distribution.

3.5.2.1.11. Zero one loss

The `zero_one_loss` function computes the sum or the average of the 0-1 classification loss (L_{0-1}) over n_{samples}. By defaults, the function normalizes over the sample. To get the sum of the L_{0-1}, set `normalize` to `False`.

In multilabel classification, the `zero_one_loss` function corresponds to the subset zero-one loss: the subset of labels must be correctly predict.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the 0-1 loss L_{0-1} is defined as:

$$L_{0-1}(y_i, \hat{y}_i) = 1(\hat{y}_i \neq y_i)$$

where $1(x)$ is the indicator function.

```
>>> from sklearn.metrics import zero_one_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> zero_one_loss(y_true, y_pred)
0.25
>>> zero_one_loss(y_true, y_pred, normalize=False)
1
```

In the multilabel case with binary label indicators:

```
>>> zero_one_loss(np.array([[0.0, 1.0], [1.0, 1.0]]), np.ones((2, 2)))
0.5
```

Example:

- See *Recursive feature elimination with cross-validation* for an example of the zero one loss usage to perform recursive feature elimination with cross-validation.

3.5.2.2. Regression metrics

The **sklearn.metrics** implements several losses, scores and utility functions to measure regression performance. Some of those have been enhanced to handle the multioutput case: **mean_absolute_error**, **mean_absolute_error** and **r2_score**.

3.5.2.2.1. Explained variance score

The **explained_variance_score** computes the explained variance regression score.

If $\hat{y}$ is the estimated target output and y is the corresponding (correct) target output, then the explained variance is estimated as follow:

$$\texttt{explained\_variance}(y, \hat{y}) = 1 - \frac{Var\{y - \hat{y}\}}{Var\{y\}}$$

The best possible score is 1.0, lower values are worse.

Here a small example of usage of the **explained_variance_score** function:

```
>>> from sklearn.metrics import explained_variance_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> explained_variance_score(y_true, y_pred)
0.957...
```

3.5.2.2.2. Mean absolute error

The **mean_absolute_error** function computes the mean absolute error, which is a risk function corresponding to the expected value of the absolute error loss or $l1$-norm loss.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the mean absolute error (MAE) estimated over n_{samples} is defined as

$$\text{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \left| y_i - \hat{y}_i \right| .$$

Here a small example of usage of the `mean_absolute_error` function:

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_absolute_error(y_true, y_pred)
0.5
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_absolute_error(y_true, y_pred)
0.75
```

3.5.2.2.3. Mean squared error

The `mean_squared_error` function computes the mean square error, which is a risk function corresponding to the expected value of the squared error loss or quadratic loss.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the mean squared error (MSE) estimated over n_{samples} is defined as

$$\text{MSE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2.$$

Here a small example of usage of the `mean_squared_error` function:

```
>>> from sklearn.metrics import mean_squared_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred)
0.375
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.7083...
```

Examples:

- See *Gradient Boosting regression* for an example of mean squared error usage to evaluate gradient boosting regression.

3.5.2.2.4. R² score, the coefficient of determination

The `r2_score` function computes R², the coefficient of determination. It provides a measure of how well future samples are likely to be predicted by the model.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the score R² estimated over n_{samples} is defined as

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2}{\sum_{i=0}^{n_{\text{samples}}-1} (y_i - \bar{y})^2}$$

where $\bar{y} = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} y_i$.

Here a small example of usage of the `r2_score` function:

```
>>> from sklearn.metrics import r2_score
```

```
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> r2_score(y_true, y_pred)
0.948...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred)
0.938...
```

Example:

- See *Lasso and Elastic Net for Sparse Signals* for an example of R^2 score usage to evaluate Lasso and Elastic Net on sparse signals.

3.5.3. Clustering metrics

The **`sklearn.metrics`** implements several losses, scores and utility function for more information see the *Clustering performance evaluation* section.

3.5.4. Biclustering metrics

The **`sklearn.metrics`** module implements bicluster scoring metrics. For more information see the *Biclustering evaluation* section.

3.5.4.1. Clustering metrics

The **`sklearn.metrics`** implements several losses, scores and utility functions. For more information see the *Clustering performance evaluation* section.

3.5.5. Dummy estimators

When doing supervised learning, a simple sanity check consists in comparing one's estimator against simple rules of thumb. **`DummyClassifier`** implements three such simple strategies for classification:

- `stratified` generates randomly predictions by respecting the training set's class distribution,
- `most_frequent` always predicts the most frequent label in the training set,
- `uniform` generates predictions uniformly at random.
- `constant` always predicts a constant label that is provided by the user.

  A major motivation of this method is F1-scoring when the positive class is in the minority.

Note that with all these strategies, the `predict` method completely ignores the input data!

To illustrate **`DummyClassifier`**, first let's create an imbalanced dataset:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.cross_validation import train_test_split
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> y[y != 1] = -1
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Next, let's compare the accuracy of `SVC` and `most_frequent`:

```
>>> from sklearn.dummy import DummyClassifier
>>> from sklearn.svm import SVC
>>> clf = SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.63...
>>> clf = DummyClassifier(strategy='most_frequent',random_state=0)
>>> clf.fit(X_train, y_train)
DummyClassifier(constant=None, random_state=0, strategy='most_frequent')
>>> clf.score(X_test, y_test)
0.57...
```

We see that `SVC` doesn't do much better than a dummy classifier. Now, let's change the kernel:

```
>>> clf = SVC(kernel='rbf', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.97...
```

We see that the accuracy was boosted to almost 100%. For a better estimate of the accuracy, it is recommended to use a cross validation strategy, if it is not too CPU costly. For more information see the *Cross-validation: evaluating estimator performance* section. Moreover if you want to optimize over the parameter space, it is highly recommended to use an appropriate methodology see the *Grid Search: Searching for estimator parameters* section.

More generally, when the accuracy of a classifier is too close to random classification, it probably means that something went wrong: features are not helpful, a hyper parameter is not correctly tuned, the classifier is suffering from class imbalance, etc...

`DummyRegressor` also implements three simple rules of thumb for regression:

- `mean` always predicts the mean of the training targets.
- `median` always predicts the median of the training targests.
- `constant` always predicts a constant value that is provided by the user.

In all these strategies, the `predict` method completely ignores the input data.

3.6. Model persistence

After training a scikit-learn model, it is desirable to have a way to persist the model for future use without having to retrain. The following section gives you an example of how to persist a model with pickle. We'll also review a few security and maintainability issues when working with pickle serialization.

3.6.1. Persistence example

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

```
>>> from sklearn import svm
>>> from sklearn import datasets
>>> clf = svm.SVC()
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)

>>> 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 (`joblib.dump` & `joblib.load`), which is more efficient on objects that carry large numpy arrays internally as is often the case for fitted scikit-learn estimators, but can only pickle to the disk and not to a string:

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

Later you can load back the pickled model (possibly in another Python process) with:

```
>>> clf = joblib.load('filename.pkl')
```

Note: joblib.dump returns a list of filenames. Each individual numpy array contained in the `clf` object is serialized as a separate file on the filesystem. All files are required in the same folder when reloading the model with joblib.load.

3.6.2. Security & maintainability limitations

pickle (and joblib by extension), has some issues regarding maintainability and security. Because of this,

- Never unpickle untrusted data
- Models saved in one version of scikit-learn might not load in another version.

In order to rebuild a similar model with future versions of scikit-learn, additional metadata should be saved

along the pickled model:

- The training data, e.g. a reference to a immutable snapshot
- The python source code used to generate the model
- The versions of scikit-learn and its dependencies
- The cross validation score obtained on the training data

This should make it possible to check that the cross-validation score is in the same range as before.

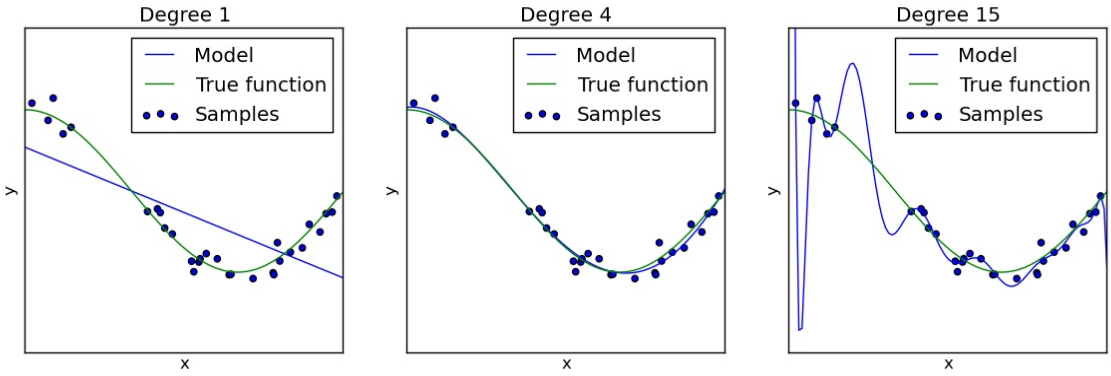
If you want to know more about these issues and explore other possible serialization methods, please refer to this talk by Alex Gaynor.

Previous Next

3.7. Validation curves: plotting scores to evaluate models

Every estimator has its advantages and drawbacks. Its generalization error can be decomposed in terms of bias, variance and noise. The **bias** of an estimator is its average error for different training sets. The **variance** of an estimator indicates how sensitive it is to varying training sets. Noise is a property of the data.

In the following plot, we see a function $f(x) = \cos(\frac{3}{2}\pi x)$ and some noisy samples from that function. We use three different estimators to fit the function: linear regression with polynomial features of degree 1, 4 and 15. We see that the first estimator can at best provide only a poor fit to the samples and the true function because it is too simple (high bias), the second estimator approximates it almost perfectly and the last estimator approximates the training data perfectly but does not fit the true function very well, i.e. it is very sensitive to varying training data (high variance).



Bias and variance are inherent properties of estimators and we usually have to select learning algorithms and hyperparameters so that both bias and variance are as low as possible (see Bias-variance dilemma). Another way to reduce the variance of a model is to use more training data. However, you should only collect more training data if the true function is too complex to be approximated by an estimator with a lower variance.

In the simple one-dimensional problem that we have seen in the example it is easy to see whether the estimator suffers from bias or variance. However, in high-dimensional spaces, models can become very difficult to visualize. For this reason, it is often helpful to use the tools described below.

Examples:

- *example_linear_model_plot_polynomial_regression.py*
- *Plotting Validation Curves*
- *Plotting Learning Curves*

3.7.1. Validation curve

To validate a model we need a scoring function (see *Model evaluation: quantifying the quality of predictions*),

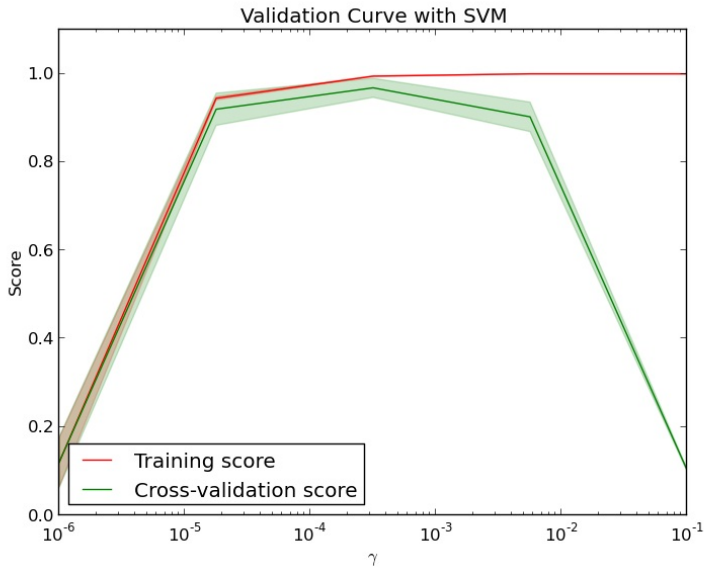
for example accuracy for classifiers. The proper way of choosing multiple hyperparameters of an estimator are of course grid search or similar methods (see *Grid Search: Searching for estimator parameters*) that select the hyperparameter with the maximum score on a validation set or multiple validation sets. Note that if we optimized the hyperparameters based on a validation score the validation score is biased and not a good estimate of the generalization any longer. To get a proper estimate of the generalization we have to compute the score on another test set.

However, it is sometimes helpful to plot the influence of a single hyperparameter on the training score and the validation score to find out whether the estimator is overfitting or underfitting for some hyperparameter values.

The function `validation_curve` can help in this case:

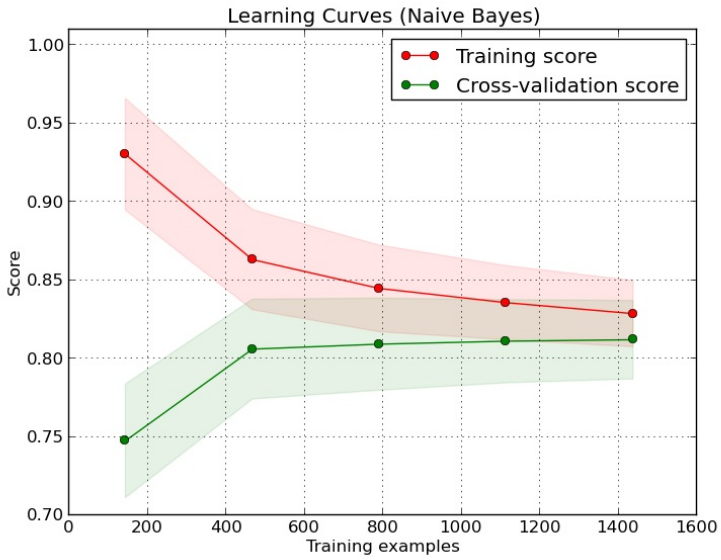
```
>>> import numpy as np
>>> from sklearn.learning_curve import validation_curve
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import Ridge

>>> np.random.seed(0)
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> indices = np.arange(y.shape[0])
>>> np.random.shuffle(indices)
>>> X, y = X[indices], y[indices]

>>> train_scores, valid_scores = validation_curve(Ridge(), X, y, "alpha",
...                                               np.logspace(-7, 3, 3))
>>> train_scores
array([[ 0.94...,  0.92...,  0.92...],
       [ 0.94...,  0.92...,  0.92...],
       [ 0.47...,  0.45...,  0.42...]])
>>> valid_scores
array([[ 0.90...,  0.92...,  0.94...],
       [ 0.90...,  0.92...,  0.94...],
       [ 0.44...,  0.39...,  0.45...]])
```

If the training score and the validation score are both low, the estimator will be underfitting. If the training score is high and the validation score is low, the estimator is overfitting and otherwise it is working very well. A low training score and a high validation score is usually not possible. All three cases can be found in the plot below where we vary the parameter γ of an SVM on the digits dataset.

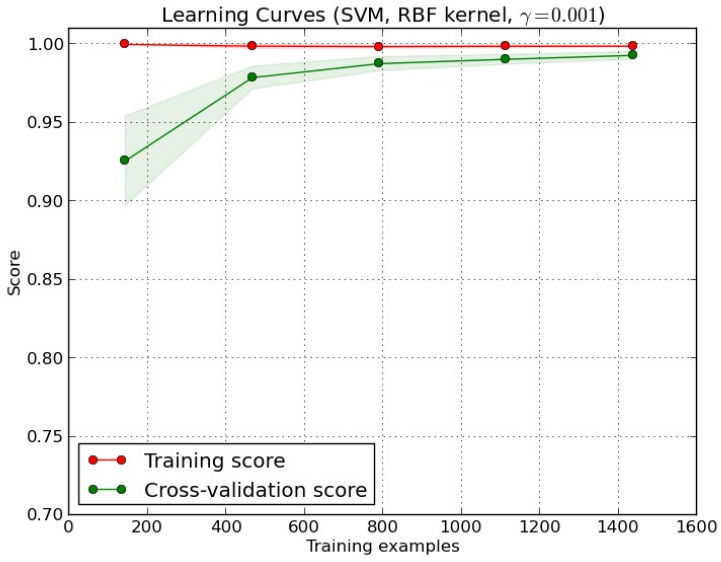


3.7.2. Learning curve

A learning curve shows the validation and training score of an estimator for varying numbers of training samples. It is a tool to find out how much we benefit from adding more training data and whether the estimator suffers more from a variance error or a bias error. If both the validation score and the training score converge to a value that is too low with increasing size of the training set, we will not benefit much from more training data. In the following plot you can see an example: naive Bayes roughly converges to a low score.



We will probably have to use an estimator or a parametrization of the current estimator that can learn more complex concepts (i.e. has a lower bias). If the training score is much greater than the validation score for the maximum number of training samples, adding more training samples will most likely increase generalization. In the following plot you can see that the SVM could benefit from more training examples.



We can use the function `learning_curve` to generate the values that are required to plot such a learning curve (number of samples that have been used, the average scores on the training sets and the average scores on the validation sets):

```
>>> from sklearn.learning_curve import learning_curve
>>> from sklearn.svm import SVC

>>> train_sizes, train_scores, valid_scores = learning_curve(
...     SVC(kernel='linear'), X, y, train_sizes=[50, 80, 110], cv=5)
>>> train_sizes
array([ 50, 80, 110])
>>> train_scores
```

```
array([[ 0.98...,  0.98 ,  0.98...,  0.98...,  0.98...],
       [ 0.98...,  1.   ,  0.98...,  0.98...,  0.98...],
       [ 0.98...,  1.   ,  0.98...,  0.98...,  0.99...]])
>>> valid_scores
array([[ 1. ,  0.93...,  1. ,  1. ,  0.96...],
       [ 1. ,  0.96...,  1. ,  1. ,  0.96...],
       [ 1. ,  0.96...,  1. ,  1. ,  0.96...]])
```

Previous Next

4.1. Feature extraction

The `sklearn.feature_extraction` module can be used to extract features in a format supported by machine learning algorithms from datasets consisting of formats such as text and image.

Note: Feature extraction is very different from *Feature selection*: the former consists in transforming arbitrary data, such as text or images, into numerical features usable for machine learning. The latter is a machine learning technique applied on these features.

4.1.1. Loading features from dicts

The class `DictVectorizer` can be used to convert feature arrays represented as lists of standard Python `dict` objects to the NumPy/SciPy representation used by scikit-learn estimators.

While not particularly fast to process, Python's `dict` has the advantages of being convenient to use, being sparse (absent features need not be stored) and storing feature names in addition to values.

`DictVectorizer` implements what is called one-of-K or "one-hot" coding for categorical (aka nominal, discrete) features. Categorical features are "attribute-value" pairs where the value is restricted to a list of discrete of possibilities without ordering (e.g. topic identifiers, types of objects, tags, names...).

In the following, "city" is a categorical attribute while "temperature" is a traditional numerical feature:

```
>>> measurements = [
...     {'city': 'Dubai', 'temperature': 33.},
...     {'city': 'London', 'temperature': 12.},
...     {'city': 'San Fransisco', 'temperature': 18.},
... ]

>>> from sklearn.feature_extraction import DictVectorizer
>>> vec = DictVectorizer()

>>> vec.fit_transform(measurements).toarray()
array([[  1.,   0.,   0.,  33.],
       [  0.,   1.,   0.,  12.],
       [  0.,   0.,   1.,  18.]])

>>> vec.get_feature_names()
['city=Dubai', 'city=London', 'city=San Fransisco', 'temperature']
```

`DictVectorizer` is also a useful representation transformation for training sequence classifiers in Natural Language Processing models that typically work by extracting feature windows around a particular word of interest.

For example, suppose that we have a first algorithm that extracts Part of Speech (PoS) tags that we want to use as complementary tags for training a sequence classifier (e.g. a chunker). The following dict could be such a window of features extracted around the word 'sat' in the sentence 'The cat sat on the mat.':

```
>>> pos_window = [
...     {
...         'word-2': 'the',
```

```
...         'pos-2': 'DT',
...         'word-1': 'cat',
...         'pos-1': 'NN',
...         'word+1': 'on',
...         'pos+1': 'PP',
...     },
...     # in a real application one would extract many such dictionaries
... ]
```

This description can be vectorized into a sparse two-dimensional matrix suitable for feeding into a classifier (maybe after being piped into a **text.TfidfTransformer** for normalization):

```
>>> vec = DictVectorizer()
>>> pos_vectorized = vec.fit_transform(pos_window)
>>> pos_vectorized
<1x6 sparse matrix of type '<... 'numpy.float64'>'
    with 6 stored elements in Compressed Sparse ... format>
>>> pos_vectorized.toarray()
array([[ 1.,  1.,  1.,  1.,  1.,  1.]])
>>> vec.get_feature_names()
['pos+1=PP', 'pos-1=NN', 'pos-2=DT', 'word+1=on', 'word-1=cat', 'word-2=the']
```

As you can imagine, if one extracts such a context around each individual word of a corpus of documents the resulting matrix will be very wide (many one-hot-features) with most of them being valued to zero most of the time. So as to make the resulting data structure able to fit in memory the `DictVectorizer` class uses a `scipy.sparse` matrix by default instead of a `numpy.ndarray`.

4.1.2. Feature hashing

The class **FeatureHasher** is a high-speed, low-memory vectorizer that uses a technique known as feature hashing, or the “hashing trick”. Instead of building a hash table of the features encountered in training, as the vectorizers do, instances of **FeatureHasher** apply a hash function to the features to determine their column index in sample matrices directly. The result is increased speed and reduced memory usage, at the expense of inspectability; the hasher does not remember what the input features looked like and has no `inverse_transform` method.

Since the hash function might cause collisions between (unrelated) features, a signed hash function is used and the sign of the hash value determines the sign of the value stored in the output matrix for a feature. This way, collisions are likely to cancel out rather than accumulate error, and the expected mean of any output feature’s value is zero.

If `non_negative=True` is passed to the constructor, the absolute value is taken. This undoes some of the collision handling, but allows the output to be passed to estimators like **MultinomialNB** or `chi2` feature selectors that expect non-negative inputs.

FeatureHasher accepts either mappings (like Python’s `dict` and its variants in the `collections` module), `(feature, value)` pairs, or strings, depending on the constructor parameter `input_type`. Mapping are treated as lists of `(feature, value)` pairs, while single strings have an implicit value of 1, so `['feat1', 'feat2', 'feat3']` is interpreted as `[('feat1', 1), ('feat2', 1), ('feat3', 1)]`. If a single feature occurs multiple times in a sample, the associated values will be summed (so `('feat', 2)` and `('feat', 3.5)` become `('feat', 5.5)`). The output from **FeatureHasher** is always a `scipy.sparse` matrix in the CSR format.

Feature hashing can be employed in document classification, but unlike **text.CountVectorizer**,

`FeatureHasher` does not do word splitting or any other preprocessing except Unicode-to-UTF-8 encoding; see *Vectorizing a large text corpus with the hashing trick*, below, for a combined tokenizer/hasher.

As an example, consider a word-level natural language processing task that needs features extracted from `(token, part_of_speech)` pairs. One could use a Python generator function to extract features:

```
def token_features(token, part_of_speech):
    if token.isdigit():
        yield "numeric"
    else:
        yield "token={}".format(token.lower())
        yield "token,pos={},{}".format(token, part_of_speech)
    if token[0].isupper():
        yield "uppercase_initial"
    if token.isupper():
        yield "all_uppercase"
    yield "pos={}".format(part_of_speech)
```

Then, the `raw_X` to be fed to `FeatureHasher.transform` can be constructed using:

```
raw_X = (token_features(tok, pos_tagger(tok)) for tok in corpus)
```

and fed to a hasher with:

```
hasher = FeatureHasher(input_type='string')
X = hasher.transform(raw_X)
```

to get a `scipy.sparse` matrix `X`.

Note the use of a generator comprehension, which introduces laziness into the feature extraction: tokens are only processed on demand from the hasher.

4.1.2.1. Implementation details

`FeatureHasher` uses the signed 32-bit variant of MurmurHash3. As a result (and because of limitations in `scipy.sparse`), the maximum number of features supported is currently $2^{31} - 1$.

The original formulation of the hashing trick by Weinberger et al. used two separate hash functions h and ξ to determine the column index and sign of a feature, respectively. The present implementation works under the assumption that the sign bit of MurmurHash3 is independent of its other bits.

Since a simple modulo is used to transform the hash function to a column index, it is advisable to use a power of two as the `n_features` parameter; otherwise the features will not be mapped evenly to the columns.

References:

- Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola and Josh Attenberg (2009). Feature hashing for large scale multitask learning. Proc. ICML.
- MurmurHash3.

4.1.3. Text feature extraction

4.1.3.1. The Bag of Words representation

Text Analysis is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length.

In order to address this, scikit-learn provides utilities for the most common ways to extract numerical features from text content, namely:

- **tokenizing** strings and giving an integer id for each possible token, for instance by using white-spaces and punctuation as token separators.
- **counting** the occurrences of tokens in each document.
- **normalizing** and weighting with diminishing importance tokens that occur in the majority of samples / documents.

In this scheme, features and samples are defined as follows:

- each **individual token occurrence frequency** (normalized or not) is treated as a **feature**.
- the vector of all the token frequencies for a given **document** is considered a multivariate **sample**.

A corpus of documents can thus be represented by a matrix with one row per document and one column per token (e.g. word) occurring in the corpus.

We call **vectorization** the general process of turning a collection of text documents into numerical feature vectors. This specific strategy (tokenization, counting and normalization) is called the **Bag of Words** or "Bag of n-grams" representation. Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.

4.1.3.2. Sparsity

As most documents will typically use a very small subset of the words used in the corpus, the resulting matrix will have many feature values that are zeros (typically more than 99% of them).

For instance a collection of 10,000 short text documents (such as emails) will use a vocabulary with a size in the order of 100,000 unique words in total while each document will use 100 to 1000 unique words individually.

In order to be able to store such a matrix in memory but also to speed up algebraic operations matrix / vector, implementations will typically use a sparse representation such as the implementations available in the `scipy.sparse` package.

4.1.3.3. Common Vectorizer usage

`CountVectorizer` implements both tokenization and occurrence counting in a single class:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
```

This model has many parameters, however the default values are quite reasonable (please see the *reference documentation* for the details):

```
>>> vectorizer = CountVectorizer(min_df=1)
>>> vectorizer
CountVectorizer(analyzer=...'word', binary=False, charset=None,
        charset_error=None, decode_error=...'strict',
        dtype=<... 'numpy.int64'>, encoding=...'utf-8', input=...'content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
```

```
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern=...'(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

Let's use it to tokenize and count the word occurrences of a minimalistic corpus of text documents:

```
>>> corpus = [
...     'This is the first document.',
...     'This is the second second document.',
...     'And the third one.',
...     'Is this the first document?',
... ]
>>> X = vectorizer.fit_transform(corpus)
>>> X
<4x9 sparse matrix of type '<... 'numpy.int64'>'
    with 19 stored elements in Compressed Sparse ... format>
```

The default configuration tokenizes the string by extracting words of at least 2 letters. The specific function that does this step can be requested explicitly:

```
>>> analyze = vectorizer.build_analyzer()
>>> analyze("This is a text document to analyze.") == (
...     ['this', 'is', 'text', 'document', 'to', 'analyze'])
True
```

Each term found by the analyzer during the fit is assigned a unique integer index corresponding to a column in the resulting matrix. This interpretation of the columns can be retrieved as follows:

```
>>> vectorizer.get_feature_names() == (
...     ['and', 'document', 'first', 'is', 'one',
...      'second', 'the', 'third', 'this'])
True

>>> X.toarray()
array([[0, 1, 1, 1, 0, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 2, 1, 0, 1],
       [1, 0, 0, 0, 1, 0, 1, 1, 0],
       [0, 1, 1, 1, 0, 0, 1, 0, 1]]...)
```

The converse mapping from feature name to column index is stored in the `vocabulary_` attribute of the vectorizer:

```
>>> vectorizer.vocabulary_.get('document')
1
```

Hence words that were not seen in the training corpus will be completely ignored in future calls to the transform method:

```
>>> vectorizer.transform(['Something completely new.']).toarray()
...
array([[0, 0, 0, 0, 0, 0, 0, 0, 0]]...)
```

Note that in the previous corpus, the first and the last documents have exactly the same words hence are encoded in equal vectors. In particular we lose the information that the last document is an interrogative form. To preserve some of the local ordering information we can extract 2-grams of words in addition to the 1-grams (individual words):

```
>>> bigram_vectorizer = CountVectorizer(ngram_range=(1, 2),
...                                     token_pattern=r'\b\w+\b', min_df=1)
>>> analyze = bigram_vectorizer.build_analyzer()
>>> analyze('Bi-grams are cool!') == (
...     ['bi', 'grams', 'are', 'cool', 'bi grams', 'grams are', 'are cool'])
True
```

The vocabulary extracted by this vectorizer is hence much bigger and can now resolve ambiguities encoded

in local positioning patterns:

```
>>> X_2 = bigram_vectorizer.fit_transform(corpus).toarray()
>>> X_2
...
array([[0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0],
       [0, 0, 1, 0, 0, 1, 1, 0, 0, 2, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0],
       [1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1]]...)
```

In particular the interrogative form “Is this” is only present in the last document:

```
>>> feature_index = bigram_vectorizer.vocabulary_.get('is this')
>>> X_2[:, feature_index]
array([0, 0, 0, 1]...)
```

4.1.3.4. Tf–idf term weighting

In a large text corpus, some words will be very present (e.g. “the”, “a”, “is” in English) hence carrying very little meaningful information about the actual contents of the document. If we were to feed the direct count data directly to a classifier those very frequent terms would shadow the frequencies of rarer yet more interesting terms.

In order to re-weight the count features into floating point values suitable for usage by a classifier it is very common to use the tf–idf transform.

Tf means **term-frequency** while tf–idf means term-frequency times **inverse document-frequency**. This is a originally a term weighting scheme developed for information retrieval (as a ranking function for search engines results), that has also found good use in document classification and clustering.

This normalization is implemented by the **`text.TfidfTransformer`** class:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> transformer = TfidfTransformer()
>>> transformer
TfidfTransformer(norm=...'l2', smooth_idf=True, sublinear_tf=False,
                 use_idf=True)
```

Again please see the *reference documentation* for the details on all the parameters.

Let’s take an example with the following counts. The first term is present 100% of the time hence not very interesting. The two other features only in less than 50% of the time hence probably more representative of the content of the documents:

```
>>> counts = [[3, 0, 1],
...           [2, 0, 0],
...           [3, 0, 0],
...           [4, 0, 0],
...           [3, 2, 0],
...           [3, 0, 2]]
...
>>> tfidf = transformer.fit_transform(counts)
>>> tfidf
<6x3 sparse matrix of type '<... 'numpy.float64'>'
    with 9 stored elements in Compressed Sparse ... format>

>>> tfidf.toarray()
array([[ 0.85...,  0.  ...,  0.52...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 0.55...,  0.83...,  0.  ...],
       [ 0.63...,  0.  ...,  0.77...]])
```

Each row is normalized to have unit euclidean norm. The weights of each feature computed by the `fit` method call are stored in a model attribute:

```
>>> transformer.idf_
array([ 1. ...,  2.25...,  1.84...])
```

As tf–idf is very often used for text features, there is also another class called **`TfidfVectorizer`** that combines all the options of **`CountVectorizer`** and **`TfidfTransformer`** in a single model:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> vectorizer = TfidfVectorizer(min_df=1)
>>> vectorizer.fit_transform(corpus)
...
<4x9 sparse matrix of type '<... 'numpy.float64'>'
    with 19 stored elements in Compressed Sparse ... format>
```

While the tf–idf normalization is often very useful, there might be cases where the binary occurrence markers might offer better features. This can be achieved by using the `binary` parameter of **`CountVectorizer`**. In particular, some estimators such as *Bernoulli Naive Bayes* explicitly model discrete boolean random variables. Also, very short texts are likely to have noisy tf–idf values while the binary occurrence info is more stable.

As usual the best way to adjust the feature extraction parameters is to use a cross-validated grid search, for instance by pipelining the feature extractor with a classifier:

- *Sample pipeline for text feature extraction and evaluation*

4.1.3.5. Decoding text files

Text is made of characters, but files are made of bytes. These bytes represent characters according to some *encoding*. To work with text files in Python, their bytes must be *decoded* to a character set called Unicode. Common encodings are ASCII, Latin-1 (Western Europe), KOI8-R (Russian) and the universal encodings UTF-8 and UTF-16. Many others exist.

Note: An encoding can also be called a ‘character set’, but this term is less accurate: several encodings can exist for a single character set.

The text feature extractors in scikit-learn know how to decode text files, but only if you tell them what encoding the files are in. The **`CountVectorizer`** takes an `encoding` parameter for this purpose. For modern text files, the correct encoding is probably UTF-8, which is therefore the default (`encoding="utf-8"`).

If the text you are loading is not actually encoded with UTF-8, however, you will get a `UnicodeDecodeError`. The vectorizers can be told to be silent about decoding errors by setting the `decode_error` parameter to either `"ignore"` or `"replace"`. See the documentation for the Python function `bytes.decode` for more details (type `help(bytes.decode)` at the Python prompt).

If you are having trouble decoding text, here are some things to try:

- Find out what the actual encoding of the text is. The file might come with a header or README that tells you the encoding, or there might be some standard encoding you can assume based on where the text comes from.
- You may be able to find out what kind of encoding it is in general using the UNIX command `file`. The Python `chardet` module comes with a script called `chardetect.py` that will guess the specific

encoding, though you cannot rely on its guess being correct.

- You could try UTF-8 and disregard the errors. You can decode byte strings with `bytes.decode(errors='replace')` to replace all decoding errors with a meaningless character, or set `decode_error='replace'` in the vectorizer. This may damage the usefulness of your features.
- Real text may come from a variety of sources that may have used different encodings, or even be sloppily decoded in a different encoding than the one it was encoded with. This is common in text retrieved from the Web. The Python package ftfy can automatically sort out some classes of decoding errors, so you could try decoding the unknown text as `latin-1` and then using `ftfy` to fix errors.
- If the text is in a mish-mash of encodings that is simply too hard to sort out (which is the case for the 20 Newsgroups dataset), you can fall back on a simple single-byte encoding such as `latin-1`. Some text may display incorrectly, but at least the same sequence of bytes will always represent the same feature.

For example, the following snippet uses `chardet` (not shipped with scikit-learn, must be installed separately) to figure out the encoding of three texts. It then vectorizes the texts and prints the learned vocabulary. The output is not shown here.

```
>>> import chardet
>>> text1 = b"Sei mir gegr\xc3\xbc\xc3\x9ft mein Sauerkraut"
>>> text2 = b"holdselig sind deine Ger\xfcche"
>>> text3 = b"\xff\xfeA\x00u\x00f\x00 \x00F\x00l\x00\xfc\x00g\x00e\x00l\x00n\x00 \x00d\x00e\x00s\
>>> decoded = [x.decode(chardet.detect(x)['encoding'])
...            for x in (text1, text2, text3)]
>>> v = CountVectorizer().fit(decoded).vocabulary_
>>> for term in v: print(v)
```

(Depending on the version of `chardet`, it might get the first one wrong.)

For an introduction to Unicode and character encodings in general, see Joel Spolsky's Absolute Minimum Every Software Developer Must Know About Unicode.

4.1.3.6. Applications and examples

The bag of words representation is quite simplistic but surprisingly useful in practice.

In particular in a **supervised setting** it can be successfully combined with fast and scalable linear models to train **document classifiers**, for instance:

- *Classification of text documents using sparse features*

In an **unsupervised setting** it can be used to group similar documents together by applying clustering algorithms such as *K-means*:

- *Clustering text documents using k-means*

Finally it is possible to discover the main topics of a corpus by relaxing the hard assignment constraint of clustering, for instance by using *Non-negative matrix factorization (NMF or NNMF)*:

- *Topics extraction with Non-Negative Matrix Factorization*

4.1.3.7. Limitations of the Bag of Words representation

A collection of unigrams (what bag of words is) cannot capture phrases and multi-word expressions,

effectively disregarding any word order dependence. Additionally, the bag of words model doesn't account for potential misspellings or word derivations.

N-grams to the rescue! Instead of building a simple collection of unigrams (n=1), one might prefer a collection of bigrams (n=2), where occurrences of pairs of consecutive words are counted.

One might alternatively consider a collection of character n-grams, a representation resilient against misspellings and derivations.

For example, let's say we're dealing with a corpus of two documents: `['words', 'wprds']`. The second document contains a misspelling of the word 'words'. A simple bag of words representation would consider these two as very distinct documents, differing in both of the two possible features. A character 2-gram representation, however, would find the documents matching in 4 out of 8 features, which may help the preferred classifier decide better:

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(2, 2), min_df=1)
>>> counts = ngram_vectorizer.fit_transform(['words', 'wprds'])
>>> ngram_vectorizer.get_feature_names() == (
...     [' w', 'ds', 'or', 'pr', 'rd', 's ', 'wo', 'wp'])
True
>>> counts.toarray().astype(int)
array([[1, 1, 1, 0, 1, 1, 1, 0],
       [1, 1, 0, 1, 1, 1, 0, 1]])
```

In the above example, `'char_wb` analyzer is used, which creates n-grams only from characters inside word boundaries (padded with space on each side). The `'char'` analyzer, alternatively, creates n-grams that span across words:

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(5, 5), min_df=1)
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
...
<1x4 sparse matrix of type '<... 'numpy.int64'>'
   with 4 stored elements in Compressed Sparse ... format>
>>> ngram_vectorizer.get_feature_names() == (
...     [' fox ', ' jump', 'jumpy', 'umpy '])
True

>>> ngram_vectorizer = CountVectorizer(analyzer='char', ngram_range=(5, 5), min_df=1)
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
...
<1x5 sparse matrix of type '<... 'numpy.int64'>'
    with 5 stored elements in Compressed Sparse ... format>
>>> ngram_vectorizer.get_feature_names() == (
...     ['jumpy', 'mpy f', 'py fo', 'umpy ', 'y fox'])
True
```

The word boundaries-aware variant `char_wb` is especially interesting for languages that use white-spaces for word separation as it generates significantly less noisy features than the raw `char` variant in that case. For such languages it can increase both the predictive accuracy and convergence speed of classifiers trained using such features while retaining the robustness with regards to misspellings and word derivations.

While some local positioning information can be preserved by extracting n-grams instead of individual words, bag of words and bag of n-grams destroy most of the inner structure of the document and hence most of the meaning carried by that internal structure.

In order to address the wider task of Natural Language Understanding, the local structure of sentences and paragraphs should thus be taken into account. Many such models will thus be casted as "Structured output" problems which are currently outside of the scope of scikit-learn.

4.1.3.8. Vectorizing a large text corpus with the hashing trick

The above vectorization scheme is simple but the fact that it holds an **in- memory mapping from the string tokens to the integer feature indices** (the `vocabulary_` attribute) causes several **problems when dealing with large datasets**:

- the larger the corpus, the larger the vocabulary will grow and hence the memory use too,
- fitting requires the allocation of intermediate data structures of size proportional to that of the original dataset.
- building the word-mapping requires a full pass over the dataset hence it is not possible to fit text classifiers in a strictly online manner.
- pickling and un-pickling vectorizers with a large `vocabulary_` can be very slow (typically much slower than pickling / un-pickling flat data structures such as a NumPy array of the same size),
- it is not easily possible to split the vectorization work into concurrent sub tasks as the `vocabulary_` attribute would have to be a shared state with a fine grained synchronization barrier: the mapping from token string to feature index is dependent on ordering of the first occurrence of each token hence would have to be shared, potentially harming the concurrent workers' performance to the point of making them slower than the sequential variant.

It is possible to overcome those limitations by combining the "hashing trick" (*Feature hashing*) implemented by the **`sklearn.feature_extraction.FeatureHasher`** class and the text preprocessing and tokenization features of the **`CountVectorizer`**.

This combination is implementing in **`HashingVectorizer`**, a transformer class that is mostly API compatible with **`CountVectorizer`**. **`HashingVectorizer`** is stateless, meaning that you don't have to call `fit` on it:

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> hv = HashingVectorizer(n_features=10)
>>> hv.transform(corpus)
...
<4x10 sparse matrix of type '<... 'numpy.float64'>'
    with 16 stored elements in Compressed Sparse ... format>
```

You can see that 16 non-zero feature tokens were extracted in the vector output: this is less than the 19 non-zeros extracted previously by the **`CountVectorizer`** on the same toy corpus. The discrepancy comes from hash function collisions because of the low value of the `n_features` parameter.

In a real world setting, the `n_features` parameter can be left to its default value of `2 ** 20` (roughly one million possible features). If memory or downstream models size is an issue selecting a lower value such as `2 ** 18` might help without introducing too many additional collisions on typical text classification tasks.

Note that the dimensionality does not affect the CPU training time of algorithms which operate on CSR matrices (`LinearSVC(dual=True)`, `Perceptron`, `SGDClassifier`, `PassiveAggressive`) but it does for algorithms that work with CSC matrices (`LinearSVC(dual=False)`, `Lasso()`, etc).

Let's try again with the default setting:

```
>>> hv = HashingVectorizer()
>>> hv.transform(corpus)
...
<4x1048576 sparse matrix of type '<... 'numpy.float64'>'
    with 19 stored elements in Compressed Sparse ... format>
```

We no longer get the collisions, but this comes at the expense of a much larger dimensionality of the output

space. Of course, other terms than the 19 used here might still collide with each other.

The `HashingVectorizer` also comes with the following limitations:

- it is not possible to invert the model (no `inverse_transform` method), nor to access the original string representation of the features, because of the one-way nature of the hash function that performs the mapping.
- it does not provide IDF weighting as that would introduce statefulness in the model. A `TfidfTransformer` can be appended to it in a pipeline if required.

4.1.3.9. Performing out-of-core scaling with HashingVectorizer

An interesting development of using a `HashingVectorizer` is the ability to perform out-of-core scaling. This means that we can learn from data that does not fit into the computer's main memory.

A strategy to implement out-of-core scaling is to stream data to the estimator in mini-batches. Each mini-batch is vectorized using `HashingVectorizer` so as to guarantee that the input space of the estimator has always the same dimensionality. The amount of memory used at any time is thus bounded by the size of a mini-batch. Although there is no limit to the amount of data that can be ingested using such an approach, from a practical point of view the learning time is often limited by the CPU time one wants to spend on the task.

For a full-fledged example of out-of-core scaling in a text classification task see *Out-of-core classification of text documents*.

4.1.3.10. Customizing the vectorizer classes

It is possible to customize the behavior by passing a callable to the vectorizer constructor:

```
>>> def my_tokenizer(s):
...     return s.split()
...
>>> vectorizer = CountVectorizer(tokenizer=my_tokenizer)
>>> vectorizer.build_analyzer()(u"Some... punctuation!") == (
...     ['some...', 'punctuation!'])
True
```

In particular we name:

- `preprocessor`: a callable that takes an entire document as input (as a single string), and returns a possibly transformed version of the document, still as an entire string. This can be used to remove HTML tags, lowercase the entire document, etc.
- `tokenizer`: a callable that takes the output from the preprocessor and splits it into tokens, then returns a list of these.
- `analyzer`: a callable that replaces the preprocessor and tokenizer. The default analyzers all call the preprocessor and tokenizer, but custom analyzers will skip this. N-gram extraction and stop word filtering take place at the analyzer level, so a custom analyzer may have to reproduce these steps.

(Lucene users might recognize these names, but be aware that scikit-learn concepts may not map one-to-one onto Lucene concepts.)

To make the preprocessor, tokenizer and analyzers aware of the model parameters it is possible to derive

from the class and override the `build_preprocessor`, `build_tokenizer` and `build_analyzer` factory methods instead of passing custom functions.

Some tips and tricks:

- If documents are pre-tokenized by an external package, then store them in files (or strings) with the tokens separated by whitespace and pass `analyzer=str.split`
- Fancy token-level analysis such as stemming, lemmatizing, compound splitting, filtering based on part-of-speech, etc. are not included in the scikit-learn codebase, but can be added by customizing either the tokenizer or the analyzer. Here's a `CountVectorizer` with a tokenizer and lemmatizer using NLTK:

```
>>> from nltk import word_tokenize
>>> from nltk.stem import WordNetLemmatizer
>>> class LemmaTokenizer(object):
...     def __init__(self):
...         self.wnl = WordNetLemmatizer()
...     def __call__(self, doc):
...         return [self.wnl.lemmatize(t) for t in word_tokenize(doc)]
...
>>> vect = CountVectorizer(tokenizer=LemmaTokenizer())
```

(Note that this will not filter out punctuation.)

Customizing the vectorizer can also be useful when handling Asian languages that do not use an explicit word separator such as whitespace.

4.1.4. Image feature extraction

4.1.4.1. Patch extraction

The **`extract_patches_2d`** function extracts patches from an image stored as a two-dimensional array, or three-dimensional with color information along the third axis. For rebuilding an image from all its patches, use **`reconstruct_from_patches_2d`**. For example let use generate a 4x4 pixel picture with 3 color channels (e.g. in RGB format):

```
>>> import numpy as np
>>> from sklearn.feature_extraction import image

>>> one_image = np.arange(4 * 4 * 3).reshape((4, 4, 3))
>>> one_image[:, :, 0]  # R channel of a fake RGB picture
array([[ 0,  3,  6,  9],
       [12, 15, 18, 21],
       [24, 27, 30, 33],
       [36, 39, 42, 45]])

>>> patches = image.extract_patches_2d(one_image, (2, 2), max_patches=2,
...     random_state=0)
>>> patches.shape
(2, 2, 2, 3)
>>> patches[:, :, :, 0]
array([[[ 0,  3],
        [12, 15]],

       [[15, 18],
        [27, 30]]])
>>> patches = image.extract_patches_2d(one_image, (2, 2))
>>> patches.shape
(9, 2, 2, 3)
>>> patches[4, :, :, 0]
array([[15, 18],
```

```
        [27, 30]])
```

Let us now try to reconstruct the original image from the patches by averaging on overlapping areas:

```
>>> reconstructed = image.reconstruct_from_patches_2d(patches, (4, 4, 3))
>>> np.testing.assert_array_equal(one_image, reconstructed)
```

The **PatchExtractor** class works in the same way as **extract_patches_2d**, only it supports multiple images as input. It is implemented as an estimator, so it can be used in pipelines. See:

```
>>> five_images = np.arange(5 * 4 * 4 * 3).reshape(5, 4, 4, 3)
>>> patches = image.PatchExtractor((2, 2)).transform(five_images)
>>> patches.shape
(45, 2, 2, 3)
```

4.1.4.2. Connectivity graph of an image

Several estimators in the scikit-learn can use connectivity information between features or samples. For instance Ward clustering (*Hierarchical clustering*) can cluster together only neighboring pixels of an image, thus forming contiguous patches:

For this purpose, the estimators use a 'connectivity' matrix, giving which samples are connected.

The function **img_to_graph** returns such a matrix from a 2D or 3D image. Similarly, **grid_to_graph** build a connectivity matrix for images given the shape of these image.

These matrices can be used to impose connectivity in estimators that use connectivity information, such as Ward clustering (*Hierarchical clustering*), but also to build precomputed kernels, or similarity matrices.

Note: Examples

- *A demo of structured Ward hierarchical clustering on Lena image*
- *Spectral clustering for image segmentation*
- *Feature agglomeration vs. univariate selection*

4.3. Kernel Approximation

This submodule contains functions that approximate the feature mappings that correspond to certain kernels, as they are used for example in support vector machines (see *Support Vector Machines*). The following feature functions perform non-linear transformations of the input, which can serve as a basis for linear classification or other algorithms.

The advantage of using approximate explicit feature maps compared to the kernel trick, which makes use of feature maps implicitly, is that explicit mappings can be better suited for online learning and can significantly reduce the cost of learning with very large datasets. Standard kernelized SVMs do not scale well to large datasets, but using an approximate kernel map it is possible to use much more efficient linear SVMs. In particular, the combination of kernel map approximations with `SGDClassifier` can make non-linear learning on large datasets possible.

Since there has not been much empirical work using approximate embeddings, it is advisable to compare results against exact kernel methods when possible.

4.3.1. Nystroem Method for Kernel Approximation

The Nystroem method, as implemented in `Nystroem` is a general method for low-rank approximations of kernels. It achieves this by essentially subsampling the data on which the kernel is evaluated. By default `Nystroem` uses the `rbf` kernel, but it can use any kernel function or a precomputed kernel matrix. The number of samples used - which is also the dimensionality of the features computed - is given by the parameter `n_components`.

4.3.2. Radial Basis Function Kernel

The `RBFSampler` constructs an approximate mapping for the radial basis function kernel, also known as *Random Kitchen Sinks* [RR2007]. This transformation can be used to explicitly model a kernel map, prior to applying a linear algorithm, for example a linear SVM:

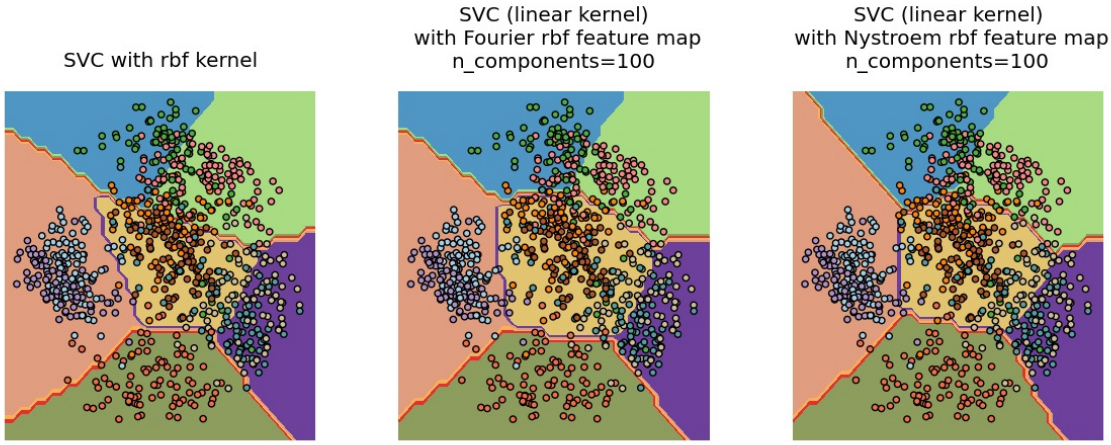
```
>>> from sklearn.kernel_approximation import RBFSampler
>>> from sklearn.linear_model import SGDClassifier
>>> X = [[0, 0], [1, 1], [1, 0], [0, 1]]
>>> y = [0, 0, 1, 1]
>>> rbf_feature = RBFSampler(gamma=1, random_state=1)
>>> X_features = rbf_feature.fit_transform(X)
>>> clf = SGDClassifier()
>>> clf.fit(X_features, y)
SGDClassifier(alpha=0.0001, class_weight=None, epsilon=0.1, eta0=0.0,
       fit_intercept=True, l1_ratio=0.15, learning_rate='optimal',
       loss='hinge', n_iter=5, n_jobs=1, penalty='l2', power_t=0.5,
       random_state=None, shuffle=False, verbose=0, warm_start=False)
>>> clf.score(X_features, y)
1.0
```

The mapping relies on a Monte Carlo approximation to the kernel values. The `fit` function performs the

Monte Carlo sampling, whereas the `transform` method performs the mapping of the data. Because of the inherent randomness of the process, results may vary between different calls to the `fit` function.

The `fit` function takes two arguments: `n_components`, which is the target dimensionality of the feature transform, and `gamma`, the parameter of the RBF-kernel. A higher `n_components` will result in a better approximation of the kernel and will yield results more similar to those produced by a kernel SVM. Note that "fitting" the feature function does not actually depend on the data given to the `fit` function. Only the dimensionality of the data is used. Details on the method can be found in [RR2007].

For a given value of `n_components` **`RBFSampler`** is often less accurate as **`Nystroem`**. **`RBFSampler`** is cheaper to compute, though, making use of larger feature spaces more efficient.



Comparing an exact RBF kernel (left) with the approximation (right)

Examples:

- *Explicit feature map approximation for RBF kernels*

4.3.3. Additive Chi Squared Kernel

The additive chi squared kernel is a kernel on histograms, often used in computer vision.

The additive chi squared kernel as used here is given by

$$k(x, y) = \sum_i \frac{2x_i y_i}{x_i + y_i}$$

This is not exactly the same as **`sklearn.metrics.additive_chi2_kernel`**. The authors of [VZ2010] prefer the version above as it is always positive definite. Since the kernel is additive, it is possible to treat all components x_i separately for embedding. This makes it possible to sample the Fourier transform in regular intervals, instead of approximating using Monte Carlo sampling.

The class **`AdditiveChi2Sampler`** implements this component wise deterministic sampling. Each component is sampled n times, yielding $2n + 1$ dimensions per input dimension (the multiple of two stems from the real and complex part of the Fourier transform). In the literature, n is usually chosen to be 1 or 2, transforming the dataset to size `n_samples * 5 * n_features` (in the case of $n = 2$).

The approximate feature map provided by `AdditiveChi2Sampler` can be combined with the approximate feature map provided by `RBFSampler` to yield an approximate feature map for the exponentiated chi squared kernel. See the [VZ2010] for details and [VVZ2010] for combination with the `RBFSampler`.

4.3.4. Skewed Chi Squared Kernel

The skewed chi squared kernel is given by:

$$k(x,y) = \prod_i \frac{2\sqrt{x_i + c}\sqrt{y_i + c}}{x_i + y_i + 2c}$$

It has properties that are similar to the exponentiated chi squared kernel often used in computer vision, but allows for a simple Monte Carlo approximation of the feature map.

The usage of the `SkewedChi2Sampler` is the same as the usage described above for the `RBFSampler`. The only difference is in the free parameter, that is called c. For a motivation for this mapping and the mathematical details see [LS2010].

4.3.5. Mathematical Details

Kernel methods like support vector machines or kernelized PCA rely on a property of reproducing kernel Hilbert spaces. For any positive definite kernel function k (a so called Mercer kernel), it is guaranteed that there exists a mapping ϕ into a Hilbert space $\mathcal{H}$, such that

$$k(x,y) = \langle \phi(x), \phi(y) \rangle$$

Where $\langle \cdot, \cdot \rangle$ denotes the inner product in the Hilbert space.

If an algorithm, such as a linear support vector machine or PCA, relies only on the scalar product of data points x_i, one may use the value of $k(x_i, x_j)$, which corresponds to applying the algorithm to the mapped data points $\phi(x_i)$. The advantage of using k is that the mapping ϕ never has to be calculated explicitly, allowing for arbitrary large features (even infinite).

One drawback of kernel methods is, that it might be necessary to store many kernel values $k(x_i, x_j)$ during optimization. If a kernelized classifier is applied to new data y_j, $k(x_i, y_j)$ needs to be computed to make predictions, possibly for many different x_i in the training set.

The classes in this submodule allow to approximate the embedding ϕ, thereby working explicitly with the representations $\phi(x_i)$, which obviates the need to apply the kernel or store training examples.

References:

[RR2007] *(1, 2)* "Random features for large-scale kernel machines" Rahimi, A. and Recht, B. - Advances in neural information processing 2007,

[LS2010] "Random Fourier approximations for skewed multiplicative histogram kernels" Random Fourier approximations for skewed multiplicative histogram kernels - Lecture Notes for Computer Sciencd (DAGM)

[VZ2010] *(1, 2)* "Efficient additive kernels via explicit feature maps" Vedaldi, A. and Zisserman, A. - Computer Vision and Pattern Recognition 2010

[VVZ2010] "Generalized RBF feature maps for Efficient Detection" Vempati, S. and Vedaldi, A. and Zisserman, A. and Jawahar, CV - 2010

4.2. Preprocessing data

The `sklearn.preprocessing` package provides several common utility functions and transformer classes to change raw feature vectors into a representation that is more suitable for the downstream estimators.

4.2.1. Standardization, or mean removal and variance scaling

Standardization of datasets is a **common requirement for many machine learning estimators** implemented in the scikit: they might behave badly if the individual feature do not more or less look like standard normally distributed data: Gaussian with **zero mean and unit variance**.

In practice we often ignore the shape of the distribution and just transform the data to center it by removing the mean value of each feature, then scale it by dividing non-constant features by their standard deviation.

For instance, many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the l1 and l2 regularizers of linear models) assume that all features are centered around zero and have variance in the same order. If a feature has a variance that is orders of magnitude larger that others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

The function **`scale`** provides a quick and easy way to perform this operation on a single array-like dataset:

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> X = np.array([[ 1., -1.,  2.],
...               [ 2.,  0.,  0.],
...               [ 0.,  1., -1.]])
>>> X_scaled = preprocessing.scale(X)

>>> X_scaled
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

Scaled data has zero mean and unit variance:

```
>>> X_scaled.mean(axis=0)
array([ 0.,  0.,  0.])

>>> X_scaled.std(axis=0)
array([ 1.,  1.,  1.])
```

The `preprocessing` module further provides a utility class **`StandardScaler`** that implements the `Transformer` API to compute the mean and standard deviation on a training set so as to be able to later reapply the same transformation on the testing set. This class is hence suitable for use in the early steps of a **`sklearn.pipeline.Pipeline`**:

```
>>> scaler = preprocessing.StandardScaler().fit(X)
>>> scaler
StandardScaler(copy=True, with_mean=True, with_std=True)

>>> scaler.mean_
```

```
array([ 1. ...,  0. ...,  0.33...])

>>> scaler.std_
array([ 0.81...,  0.81...,  1.24...])

>>> scaler.transform(X)
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

The scaler instance can then be used on new data to transform it the same way it did on the training set:

```
>>> scaler.transform([[-1.,  1., 0.]])
array([[-2.44...,  1.22..., -0.26...]])
```

It is possible to disable either centering or scaling by either passing `with_mean=False` or `with_std=False` to the constructor of **`StandardScaler`**.

4.2.1.1. Scaling features to a range

An alternative standardization is scaling features to lie between a given minimum and maximum value, often between zero and one. This can be achieved using **`MinMaxScaler`**.

The motivation to use this scaling include robustness to very small standard deviations of features and preserving zero entries in sparse data.

Here is an example to scale a toy data matrix to the `[0, 1]` range:

```
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
...
>>> min_max_scaler = preprocessing.MinMaxScaler()
>>> X_train_minmax = min_max_scaler.fit_transform(X_train)
>>> X_train_minmax
array([[ 0.5       ,  0.        ,  1.        ],
       [ 1.        ,  0.5       ,  0.33333333],
       [ 0.        ,  1.        ,  0.        ]])
```

The same instance of the transformer can then be applied to some new test data unseen during the fit call: the same scaling and shifting operations will be applied to be consistent with the transformation performed on the train data:

```
>>> X_test = np.array([[ -3., -1.,  4.]])
>>> X_test_minmax = min_max_scaler.transform(X_test)
>>> X_test_minmax
array([[-1.5       ,  0.        ,  1.66666667]])
```

It is possible to introspect the scaler attributes to find about the exact nature of the transformation learned on the training data:

```
>>> min_max_scaler.scale_
array([ 0.5       ,  0.5       ,  0.33...])

>>> min_max_scaler.min_
array([ 0.        ,  0.5       ,  0.33...])
```

If **`MinMaxScaler`** is given an explicit `feature_range=(min, max)` the full formula is:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

X_scaled = X_std / (max - min) + min
```

References:

Further discussion on the importance of centering and scaling data is available on this FAQ: Should I normalize/standardize/rescale the data?

Scaling vs Whitening

It is sometimes not enough to center and scale the features independently, since a downstream model can further make some assumption on the linear independence of the features.

To address this issue you can use **`sklearn.decomposition.PCA`** or **`sklearn.decomposition.RandomizedPCA`** with `whiten=True` to further remove the linear correlation across features.

Sparse input

`scale` and **`StandardScaler`** accept `scipy.sparse` matrices as input **only when with_mean=False is explicitly passed to the constructor**. Otherwise a `ValueError` will be raised as silently centering would break the sparsity and would often crash the execution by allocating excessive amounts of memory unintentionally.

If the centered data is expected to be small enough, explicitly convert the input to an array using the `toarray` method of sparse matrices instead.

For sparse input the data is **converted to the Compressed Sparse Rows representation** (see `scipy.sparse.csr_matrix`). To avoid unnecessary memory copies, it is recommended to choose the CSR representation upstream.

Scaling target variables in regression

`scale` and **`StandardScaler`** work out-of-the-box with 1d arrays. This is very useful for scaling the target / response variables used for regression.

4.2.1.2. Centering kernel matrices

If you have a kernel matrix of a kernel K that computes a dot product in a feature space defined by function phi, a **`KernelCenterer`** can transform the kernel matrix so that it contains inner products in the feature space defined by phi followed by removal of the mean in that space.

4.2.2. Normalization

Normalization is the process of **scaling individual samples to have unit norm**. This process can be useful if you plan to use a quadratic form such as the dot-product or any other kernel to quantify the similarity of any pair of samples.

This assumption is the base of the Vector Space Model often used in text classification and clustering contexts.

The function **`normalize`** provides a quick and easy way to perform this operation on a single array-like

dataset, either using the `l1` or `l2` norms:

```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
>>> X_normalized = preprocessing.normalize(X, norm='l2')

>>> X_normalized
array([[ 0.40..., -0.40...,  0.81...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 0.  ...,  0.70..., -0.70...]])
```

The `preprocessing` module further provides a utility class **`Normalizer`** that implements the same operation using the `Transformer` API (even though the `fit` method is useless in this case: the class is stateless as this operation treats samples independently).

This class is hence suitable for use in the early steps of a **`sklearn.pipeline.Pipeline`**:

```
>>> normalizer = preprocessing.Normalizer().fit(X)  # fit does nothing
>>> normalizer
Normalizer(copy=True, norm='l2')
```

The normalizer instance can then be used on sample vectors as any transformer:

```
>>> normalizer.transform(X)
array([[ 0.40..., -0.40...,  0.81...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 0.  ...,  0.70..., -0.70...]])

>>> normalizer.transform([[-1.,  1., 0.]])
array([[-0.70...,  0.70...,  0.  ...]])
```

Sparse input

`normalize` and **`Normalizer`** accept **both dense array-like and sparse matrices from scipy.sparse as input**.

For sparse input the data is **converted to the Compressed Sparse Rows representation** (see `scipy.sparse.csr_matrix`) before being fed to efficient Cython routines. To avoid unnecessary memory copies, it is recommended to choose the CSR representation upstream.

4.2.3. Binarization

4.2.3.1. Feature binarization

Feature binarization is the process of **thresholding numerical features to get boolean values**. This can be useful for downstream probabilistic estimators that make assumption that the input data is distributed according to a multi-variate Bernoulli distribution. For instance, this is the case for the **`sklearn.neural_network.BernoulliRBM`**.

It is also common among the text processing community to use binary feature values (probably to simplify the probabilistic reasoning) even if normalized counts (a.k.a. term frequencies) or TF-IDF valued features often perform slightly better in practice.

As for the **`Normalizer`**, the utility class **`Binarizer`** is meant to be used in the early stages of **`sklearn.pipeline.Pipeline`**. The `fit` method does nothing as each sample is treated independently of

others:

```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]

>>> binarizer = preprocessing.Binarizer().fit(X)  # fit does nothing
>>> binarizer
Binarizer(copy=True, threshold=0.0)

>>> binarizer.transform(X)
array([[ 1.,  0.,  1.],
       [ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])
```

It is possible to adjust the threshold of the binarizer:

```
>>> binarizer = preprocessing.Binarizer(threshold=1.1)
>>> binarizer.transform(X)
array([[ 0.,  0.,  1.],
       [ 1.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

As for the **StandardScaler** and **Normalizer** classes, the preprocessing module provides a companion function **binarize** to be used when the transformer API is not necessary.

Sparse input

binarize and **Binarizer** accept **both dense array-like and sparse matrices from scipy.sparse as input**.

For sparse input the data is **converted to the Compressed Sparse Rows representation** (see `scipy.sparse.csr_matrix`). To avoid unnecessary memory copies, it is recommended to choose the CSR representation upstream.

4.2.4. Encoding categorical features

Often features are not given as continuous values but categorical. For example a person could have features `["male", "female"]`, `["from Europe", "from US", "from Asia"]`, `["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"]`. Such features can be efficiently coded as integers, for instance `["male", "from US", "uses Internet Explorer"]` could be expressed as `[0, 1, 3]` while `["female", "from Asia", "uses Chrome"]` would be `[1, 2, 1]`.

Such integer representation can not be used directly with scikit-learn estimators, as these expect continuous input, and would interpret the categories as being ordered, which is often not desired (i.e. the set of browsers was ordered arbitrarily).

One possibility to convert categorical features to features that can be used with scikit-learn estimators is to use a one-of-K or one-hot encoding, which is implemented in **OneHotEncoder**. This estimator transforms each categorical feature with `m` possible values into `m` binary features, with only one active.

Continuing the example above:

```
>>> enc = preprocessing.OneHotEncoder()
>>> enc.fit([[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]])
OneHotEncoder(categorical_features='all', dtype=<... 'float'>,
       n_values='auto', sparse=True)
>>> enc.transform([[0, 1, 3]]).toarray()
```

```
array([[ 1.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  1.]])
```

By default, how many values each feature can take is inferred automatically from the dataset. It is possible to specify this explicitly using the parameter `n_values`. There are two genders, three possible continents and four web browsers in our dataset. Then we fit the estimator, and transform a data point. In the result, the first two numbers encode the gender, the next set of three numbers the continent and the last four the web browser.

See *Loading features from dicts* for categorical features that are represented as a dict, not as integers.

4.2.5. Label preprocessing

4.2.5.1. Label binarization

`LabelBinarizer` is a utility class to help create a label indicator matrix from a list of multi-class labels:

```
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer(neg_label=0, pos_label=1, sparse_output=False)
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])
```

For multiple labels per instance, use **`MultiLabelBinarizer`**:

```
>>> lb = preprocessing.MultiLabelBinarizer()
>>> lb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
       [0, 0, 1]])
>>> lb.classes_
array([1, 2, 3])
```

4.2.5.2. Label encoding

`LabelEncoder` is a utility class to help normalize labels such that they contain only values between 0 and n_classes-1. This is sometimes useful for writing efficient Cython routines. **`LabelEncoder`** can be used as follows:

```
>>> from sklearn import preprocessing
>>> le = preprocessing.LabelEncoder()
>>> le.fit([1, 2, 2, 6])
LabelEncoder()
>>> le.classes_
array([1, 2, 6])
>>> le.transform([1, 1, 2, 6])
array([0, 0, 1, 2])
>>> le.inverse_transform([0, 0, 1, 2])
array([1, 1, 2, 6])
```

It can also be used to transform non-numerical labels (as long as they are hashable and comparable) to numerical labels:

```
>>> le = preprocessing.LabelEncoder()
>>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
LabelEncoder()
>>> list(le.classes_)
['amsterdam', 'paris', 'tokyo']
>>> le.transform(["tokyo", "tokyo", "paris"])
```

```
array([2, 2, 1])
>>> list(le.inverse_transform([2, 2, 1]))
['tokyo', 'tokyo', 'paris']
```

4.2.6. Imputation of missing values

For various reasons, many real world datasets contain missing values, often encoded as blanks, NaNs or other placeholders. Such datasets however are incompatible with scikit-learn estimators which assume that all values in an array are numerical, and that all have and hold meaning. A basic strategy to use incomplete datasets is to discard entire rows and/or columns containing missing values. However, this comes at the price of losing data which may be valuable (even though incomplete). A better strategy is to impute the missing values, i.e., to infer them from the known part of the data.

The **`Imputer`** class provides basic strategies for imputing missing values, either using the mean, the median or the most frequent value of the row or column in which the missing values are located. This class also allows for different missing values encodings.

The following snippet demonstrates how to replace missing values, encoded as `np.nan`, using the mean value of the columns (axis 0) that contain the missing values:

```
>>> import numpy as np
>>> from sklearn.preprocessing import Imputer
>>> imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imp.fit([[1, 2], [np.nan, 3], [7, 6]])
Imputer(axis=0, copy=True, missing_values='NaN', strategy='mean', verbose=0)
>>> X = [[np.nan, 2], [6, np.nan], [7, 6]]
>>> print(imp.transform(X))
[[ 4.          2.        ]
 [ 6.          3.666...]
 [ 7.          6.        ]]
```

The **`Imputer`** class also supports sparse matrices:

```
>>> import scipy.sparse as sp
>>> X = sp.csc_matrix([[1, 2], [0, 3], [7, 6]])
>>> imp = Imputer(missing_values=0, strategy='mean', axis=0)
>>> imp.fit(X)
Imputer(axis=0, copy=True, missing_values=0, strategy='mean', verbose=0)
>>> X_test = sp.csc_matrix([[0, 2], [6, 0], [7, 6]])
>>> print(imp.transform(X_test))
[[ 4.          2.        ]
 [ 6.          3.666...]
 [ 7.          6.        ]]
```

Note that, here, missing values are encoded by 0 and are thus implicitly stored in the matrix. This format is thus suitable when there are many more missing values than observed values.

`Imputer` can be used in a Pipeline as a way to build a composite estimator that supports imputation. See *Imputing missing values before building an estimator*

4.2.7. Unsupervised data reduction

If your number of features is high, it may be useful to reduce it with an unsupervised step prior to supervised steps. Many of the *Unsupervised learning* methods implement a `transform` method that can be used to reduce the dimensionality. Below we discuss two specific example of this pattern that are heavily used.

Pipelining

The unsupervised data reduction and the supervised estimator can be chained in one step. See *Pipeline: chaining estimators*.

4.2.7.1. PCA: principal component analysis

`decomposition.PCA` looks for a combination of features that capture well the variance of the original features.

Examples

- *Faces recognition example using eigenfaces and SVMs*

4.2.7.2. Random projections

The module: `random_projection` provides several tools for data reduction by random projections. See the relevant section of the documentation: *Random Projection*.

Examples

- *The Johnson-Lindenstrauss bound for embedding with random projections*

4.2.7.3. Feature agglometration

`cluster.FeatureAgglomeration` applies *Hierarchical clustering* to group together features that behave similarly.

Examples

- *Feature agglomeration vs. univariate selection*
- *Feature agglomeration*

Feature scaling

Note that if features have very different scaling or statistical properties, `cluster.FeatureAgglomeration` maye not be able to capture the links between related features. Using a `preprocessing.StandardScaler` can be useful in these settings.

4.4. Random Projection

The `sklearn.random_projection` module implements a simple and computationally efficient way to reduce the dimensionality of the data by trading a controlled amount of accuracy (as additional variance) for faster processing times and smaller model sizes. This module implements two types of unstructured random matrix: *Gaussian random matrix* and *sparse random matrix*.

The dimensions and distribution of random projections matrices are controlled so as to preserve the pairwise distances between any two samples of the dataset. Thus random projection is a suitable approximation technique for distance based method.

References:

- Sanjoy Dasgupta. 2000. Experiments with random projection. In Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence (UAI'00), Craig Boutilier and Moisés Goldszmidt (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 143-151.
- Ella Bingham and Heikki Mannila. 2001. Random projection in dimensionality reduction: applications to image and text data. In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '01). ACM, New York, NY, USA, 245-250.

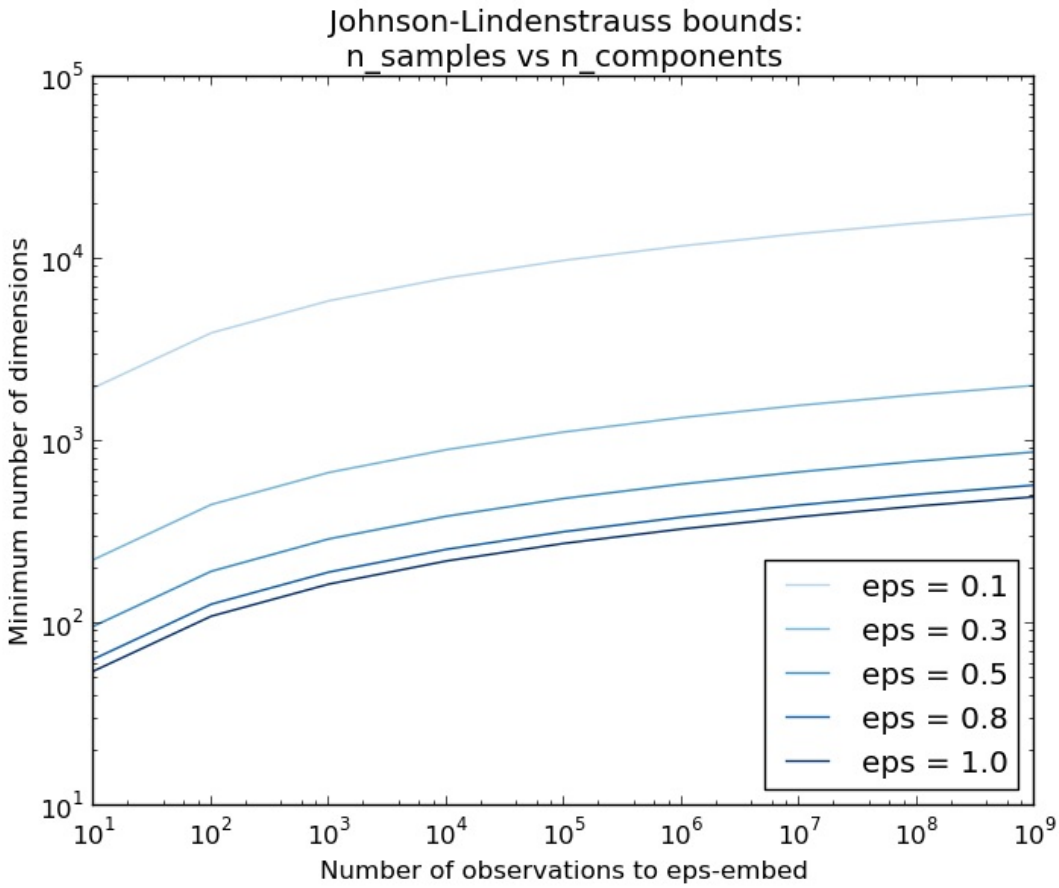
4.4.1. The Johnson-Lindenstrauss lemma

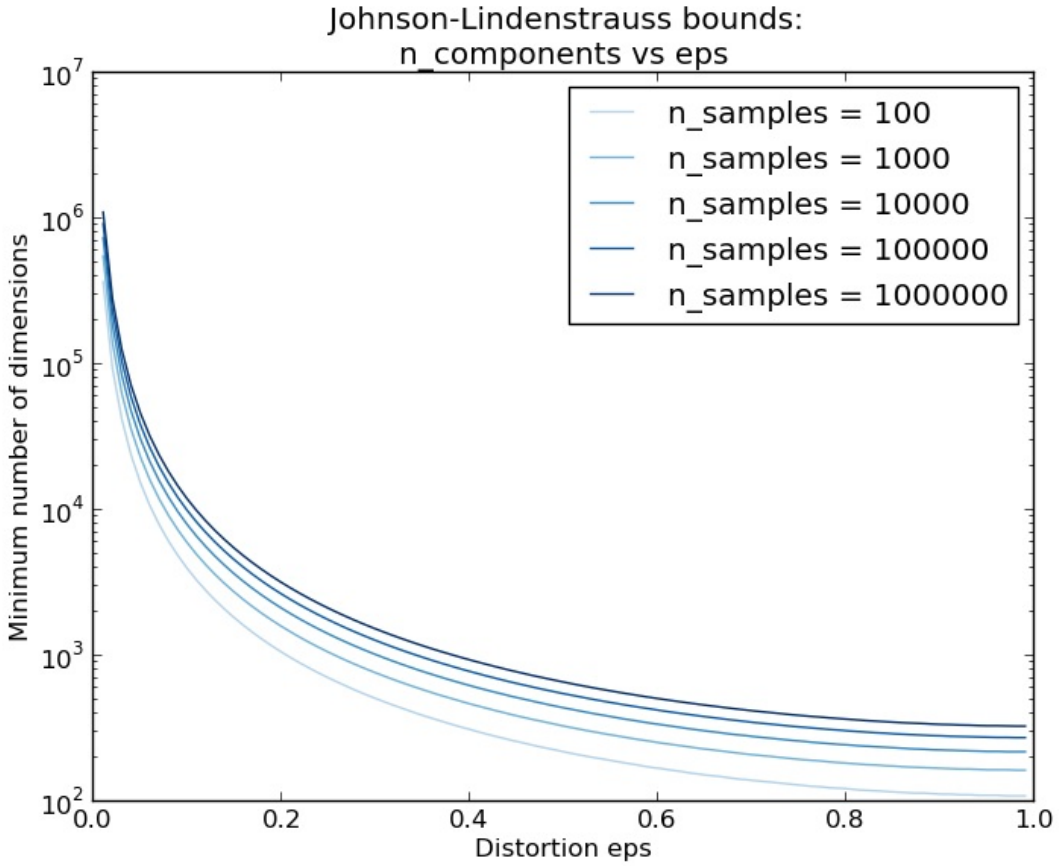
The main theoretical result behind the efficiency of random projection is the Johnson-Lindenstrauss lemma (quoting Wikipedia):

> In mathematics, the Johnson-Lindenstrauss lemma is a result concerning low-distortion embeddings of points from high-dimensional into low-dimensional Euclidean space. The lemma states that a small set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved. The map used for the embedding is at least Lipschitz, and can even be taken to be an orthogonal projection.

Knowing only the number of samples, the `sklearn.random_projection.johnson_lindenstrauss_min_dim` estimates conservatively the minimal size of the random subspace to guarantee a bounded distortion introduced by the random projection:

```
>>> from sklearn.random_projection import johnson_lindenstrauss_min_dim
>>> johnson_lindenstrauss_min_dim(n_samples=1e6, eps=0.5)
663
>>> johnson_lindenstrauss_min_dim(n_samples=1e6, eps=[0.5, 0.1, 0.01])
array([    663,   11841, 1112658])
>>> johnson_lindenstrauss_min_dim(n_samples=[1e4, 1e5, 1e6], eps=0.1)
array([ 7894,  9868, 11841])
```





Example:

- See *The Johnson-Lindenstrauss bound for embedding with random projections* for a theoretical explication on the Johnson-Lindenstrauss lemma and an empirical validation using sparse random matrices.

References:

- Sanjoy Dasgupta and Anupam Gupta, 1999. An elementary proof of the Johnson-Lindenstrauss Lemma.

4.4.2. Gaussian random projection

The `sklearn.random_projection.GaussianRandomProjection` reduces the dimensionality by projecting the original input space on a randomly generated matrix where components are drawn from the following distribution $N(0, \frac{1}{n_{components}})$.

Here a small excerpt which illustrates how to use the Gaussian random projection transformer:

```
>>> import numpy as np
>>> from sklearn import random_projection
>>> X = np.random.rand(100, 10000)
>>> transformer = random_projection.GaussianRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```

4.4.3. Sparse random projection

The `sklearn.random_projection.SparseRandomProjection` reduces the dimensionality by projecting the original input space using a sparse random matrix.

Sparse random matrices are an alternative to dense Gaussian random projection matrix that guarantees similar embedding quality while being much more memory efficient and allowing faster computation of the projected data.

If we define `s = 1 / density`, the elements of the random matrix are drawn from

$$\begin{cases} -\sqrt{\frac{s}{n_{\text{components}}}} & 1/2s \\ 0 & \text{with probability } 1 - 1/s \\ +\sqrt{\frac{s}{n_{\text{components}}}} & 1/2s \end{cases}$$

where $n_{\text{components}}$ is the size of the projected subspace. By default the density of non zero elements is set to the minimum density as recommended by Ping Li et al.: $1/\sqrt{n_{\text{features}}}$.

Here a small excerpt which illustrates how to use the sparse random projection transformer:

```
>>> import numpy as np
>>> from sklearn import random_projection
>>> X = np.random.rand(100,10000)
>>> transformer = random_projection.SparseRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```

References:

- D. Achlioptas. 2003. Database-friendly random projections: Johnson-Lindenstrauss with binary

coins. Journal of Computer and System Sciences 66 (2003) 671–687
- Ping Li, Trevor J. Hastie, and Kenneth W. Church. 2006. Very sparse random projections. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '06). ACM, New York, NY, USA, 287-296.

Previous Next

4.5. Pairwise metrics, Affinities and Kernels

The `sklearn.metrics.pairwise` submodule implements utilities to evaluate pairwise distances or affinity of sets of samples.

This module contains both distance metrics and kernels. A brief summary is given on the two here.

Distance metrics are a function d(a, b) such that d(a, b) < d(a, c) if objects a and b are considered “more similar” to objects a and c. Two objects exactly alike would have a distance of zero. One of the most popular examples is Euclidean distance. To be a ‘true’ metric, it must obey the following four conditions:

```
1. d(a, b) >= 0, for all a and b
2. d(a, b) == 0, if and only if a = b, positive definiteness
3. d(a, b) == d(b, a), symmetry
4. d(a, c) <= d(a, b) + d(b, c), the triangle inequality
```

Kernels are measures of similarity, i.e. `s(a, b) > s(a, c)` if objects `a` and `b` are considered “more similar” to objects `a` and `c`. A kernel must also be positive semi-definite.

There are a number of ways to convert between a distance metric and a similarity measure, such as a kernel. Let D be the distance, and S be the kernel:

1. `S = np.exp(-D * gamma)`, where one heuristic for choosing `gamma` is `1 / num_features`
2. `S = 1. / (D / np.max(D))`

4.5.1. Cosine similarity

`cosine_similarity` computes the L2-normalized dot product of vectors. That is, if x and y are row vectors, their cosine similarity k is defined as:

$$k(x, y) = \frac{x y^\top}{\|x\| \|y\|}$$

This is called cosine similarity, because Euclidean (L2) normalization projects the vectors onto the unit sphere, and their dot product is then the cosine of the angle between the points denoted by the vectors.

This kernel is a popular choice for computing the similarity of documents represented as tf-idf vectors. **`cosine_similarity`** accepts `scipy.sparse` matrices. (Note that the tf-idf functionality in `sklearn.feature_extraction.text` can produce normalized vectors, in which case **`cosine_similarity`** is equivalent to **`linear_kernel`**, only slower.)

References:

- C.D. Manning, P. Raghavan and H. Schütze (2008). Introduction to Information Retrieval. Cambridge University Press. http://nlp.stanford.edu/IR-book/html/htmledition/the-vector-space-

model-for-scoring-1.html

4.5.2. Chi Squared Kernel

The chi squared kernel is a very popular choice for training non-linear SVMs in Computer Vision applications. It can be computed using `chi2_kernel` and then passed to an `sklearn.svm.SVC` with `kernel="precomputed"`:

```
>>> from sklearn.svm import SVC
>>> from sklearn.metrics.pairwise import chi2_kernel
>>> X = [[0, 1], [1, 0], [.2, .8], [.7, .3]]
>>> y = [0, 1, 0, 1]
>>> K = chi2_kernel(X, gamma=.5)
>>> K
array([[ 1.        ,  0.36...,  0.89...,  0.58...],
       [ 0.36...,  1.        ,  0.51...,  0.83...],
       [ 0.89...,  0.51...,  1.        ,  0.77... ],
       [ 0.58...,  0.83...,  0.77... ,  1.        ]])

>>> svm = SVC(kernel='precomputed').fit(K, y)
>>> svm.predict(K)
array([0, 1, 0, 1])
```

It can also be directly used as the `kernel` argument:

```
>>> svm = SVC(kernel=chi2_kernel).fit(X, y)
>>> svm.predict(X)
array([0, 1, 0, 1])
```

The chi squared kernel is given by

$$k(x, y) = \exp\left(-\gamma \sum_i \frac{(x[i] - y[i])^2}{x[i] + y[i]}\right)$$

The data is assumed to be non-negative, and is often normalized to have an L1-norm of one. The normalization is rationalized with the connection to the chi squared distance, which is a distance between discrete probability distributions.

The chi squared kernel is most commonly used on histograms (bags) of visual words.

References:

- Zhang, J. and Marszalek, M. and Lazebnik, S. and Schmid, C. Local features and kernels for classification of texture and object categories: A comprehensive study International Journal of Computer Vision 2007 http://eprints.pascal-network.org/archive/00002309/01/Zhang06-IJCV.pdf

5. Dataset loading utilities

The `sklearn.datasets` package embeds some small toy datasets as introduced in the *Getting Started* section.

To evaluate the impact of the scale of the dataset (`n_samples` and `n_features`) while controlling the statistical properties of the data (typically the correlation and informativeness of the features), it is also possible to generate synthetic data.

This package also features helpers to fetch larger datasets commonly used by the machine learning community to benchmark algorithm on data that comes from the 'real world'.

5.1. General dataset API

There are three distinct kinds of dataset interfaces for different types of datasets. The simplest one is the interface for sample images, which is described below in the *Sample images* section.

The dataset generation functions and the svmlight loader share a simplistic interface, returning a tuple `(X, y)` consisting of a n_samples x n_features numpy array X and an array of length n_samples containing the targets y.

The toy datasets as well as the 'real world' datasets and the datasets fetched from mldata.org have more sophisticated structure. These functions return a dictionary-like object holding at least two items: an array of shape `n_samples` * `` n_features`` with key `data` (except for 20newsgroups) and a NumPy array of length `n_samples`, containing the target values, with key `target`.

The datasets also contain a description in `DESCR` and some contain `feature_names` and `target_names`. See the dataset descriptions below for details.

5.2. Toy datasets

scikit-learn comes with a few small standard datasets that do not require to download any file from some external website.

`load_boston`()	Load and return the boston house-prices dataset (regression).
`load_iris`()	Load and return the iris dataset (classification).
`load_diabetes`()	Load and return the diabetes dataset (regression).
`load_digits`([n_class])	Load and return the digits dataset (classification).
`load_linnerud`()	Load and return the linnerud dataset (multivariate regression).

These datasets are useful to quickly illustrate the behavior of the various algorithms implemented in the scikit. They are however often too small to be representative of real world machine learning tasks.

5.3. Sample images

The scikit also embed a couple of sample JPEG images published under Creative Commons license by their authors. Those image can be useful to test algorithms and pipeline on 2D data.

`load_sample_images`()	Load sample images for image manipulation.
`load_sample_image`(image_name)	Load the numpy array of a single sample image

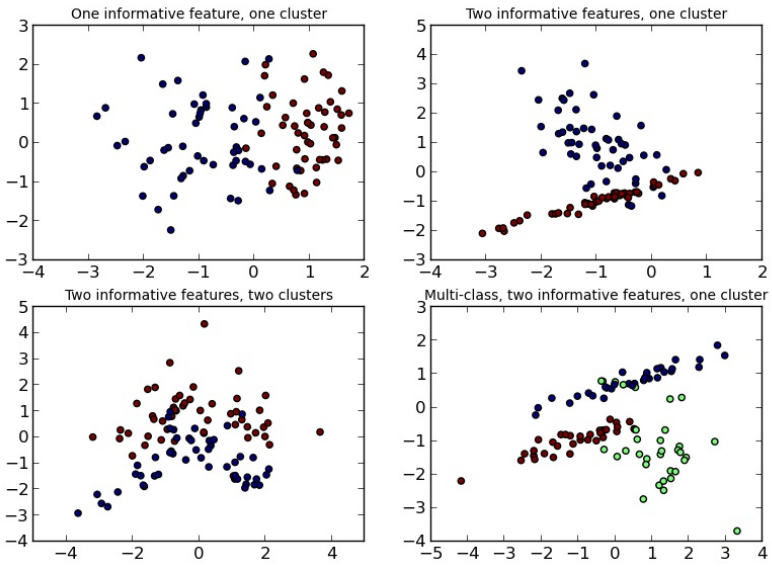
Warning: The default coding of images is based on the `uint8` dtype to spare memory. Often machine learning algorithms work best if the input is converted to a floating point representation first. Also, if you plan to use `pylab.imshow` don't forget to scale to the range 0 - 1 as done in the following example.

Examples:

- *Color Quantization using K-Means*

5.4. Sample generators

In addition, scikit-learn includes various random sample generators that can be used to build artificial datasets of controlled size and complexity.



`make_classification`([n_samples, n_features, ...])	Generate a random n-class classification problem.
`make_multilabel_classification`([n_samples, ...])	Generate a random multilabel classification problem.
`make_regression`([n_samples, n_features, ...])	Generate a random regression problem.
`make_blobs`([n_samples, n_features, centers, ...])	Generate isotropic Gaussian blobs for clustering.
`make_friedman1`([n_samples, n_features, ...])	Generate the "Friedman #1" regression problem
`make_friedman2`([n_samples, noise, random_state])	Generate the "Friedman #2" regression problem
`make_friedman3`([n_samples, noise, random_state])	Generate the "Friedman #3" regression problem
`make_hastie_10_2`([n_samples, random_state])	Generates data for binary classification used in Hastie et al.
`make_low_rank_matrix`([n_samples, ...])	Generate a mostly low rank matrix with bell-shaped singular values
`make_sparse_coded_signal`(n_samples, ...[, ...])	Generate a signal as a sparse combination of dictionary elements.

make_sparse_uncorrelated([n_samples, ...])	Generate a random regression problem with sparse uncorrelated design
make_spd_matrix(n_dim[, random_state])	Generate a random symmetric, positive-definite matrix.
make_swiss_roll([n_samples, noise, random_state])	Generate a swiss roll dataset.
make_s_curve([n_samples, noise, random_state])	Generate an S curve dataset.
make_sparse_spd_matrix([dim, alpha, ...])	Generate a sparse symmetric definite positive matrix.
make_biclusters(shape, n_clusters[, noise, ...])	Generate an array with constant block diagonal structure for biclustering.
make_checkerboard(shape, n_clusters[, ...])	Generate an array with block checkerboard structure for biclustering.

5.5. Datasets in svmlight / libsvm format

scikit-learn includes utility functions for loading datasets in the svmlight / libsvm format. In this format, each line takes the form `<label> <feature-id>:<feature-value> <feature-id>:<feature-value> ...`. This format is especially suitable for sparse datasets. In this module, scipy sparse CSR matrices are used for `X` and numpy arrays are used for `y`.

You may load a dataset like as follows:

```
>>> from sklearn.datasets import load_svmlight_file
>>> X_train, y_train = load_svmlight_file("/path/to/train_dataset.txt")
...
```

You may also load two (or more) datasets at once:

```
>>> X_train, y_train, X_test, y_test = load_svmlight_files(
...     ("/path/to/train_dataset.txt", "/path/to/test_dataset.txt"))
...
```

In this case, `X_train` and `X_test` are guaranteed to have the same number of features. Another way to achieve the same result is to fix the number of features:

```
>>> X_test, y_test = load_svmlight_file(
...     "/path/to/test_dataset.txt", n_features=X_train.shape[1])
...
```

Related links:

Public datasets in svmlight / libsvm format: http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

Faster API-compatible implementation: https://github.com/mblondel/svmlight-loader

5.6. The Olivetti faces dataset

This dataset contains a set of face images taken between April 1992 and April 1994 at AT&T Laboratories Cambridge. The website describing the original dataset is now defunct, but archived copies can be accessed through the Internet Archive's Wayback Machine. The **sklearn.datasets.fetch_olivetti_faces** function is the data fetching / caching function that downloads the data archive from AT&T.

As described on the original website:

> There are ten different images of each of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement).

The image is quantized to 256 grey levels and stored as unsigned 8-bit integers; the loader will convert these to floating point values on the interval [0, 1], which are easier to work with for many algorithms.

The “target” for this database is an integer from 0 to 39 indicating the identity of the person pictured; however, with only 10 examples per class, this relatively small dataset is more interesting from an unsupervised or semi-supervised perspective.

The original dataset consisted of 92 x 112, while the version available here consists of 64x64 images.

When using these images, please give credit to AT&T Laboratories Cambridge.

5.7. The 20 newsgroups text dataset

The 20 newsgroups dataset comprises around 18000 newsgroups posts on 20 topics split in two subsets: one for training (or development) and the other one for testing (or for performance evaluation). The split between the train and test set is based upon a messages posted before and after a specific date.

This module contains two loaders. The first one, `sklearn.datasets.fetch_20newsgroups`, returns a list of the raw texts that can be fed to text feature extractors such as **`sklearn.feature_extraction.text.Vectorizer`** with custom parameters so as to extract feature vectors. The second one, `sklearn.datasets.fetch_20newsgroups_vectorized`, returns ready-to-use features, i.e., it is not necessary to use a feature extractor.

5.7.1. Usage

The `sklearn.datasets.fetch_20newsgroups` function is a data fetching / caching functions that downloads the data archive from the original 20 newsgroups website, extracts the archive contents in the `~/scikit_learn_data/20news_home` folder and calls the `sklearn.datasets.load_file` on either the training or testing set folder, or both of them:

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> newsgroups_train = fetch_20newsgroups(subset='train')

>>> from pprint import pprint
>>> pprint(list(newsgroups_train.target_names))
['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
```

```
 'talk.politics.mideast',
 'talk.politics.misc',
 'talk.religion.misc']
```

The real data lies in the `filenames` and `target` attributes. The target attribute is the integer index of the category:

```
>>> newsgroups_train.filenames.shape
(11314,)
>>> newsgroups_train.target.shape
(11314,)
>>> newsgroups_train.target[:10]
array([12,  6,  9,  8,  6,  7,  9,  2, 13, 19])
```

It is possible to load only a sub-selection of the categories by passing the list of the categories to load to the `fetch_20newsgroups` function:

```
>>> cats = ['alt.atheism', 'sci.space']
>>> newsgroups_train = fetch_20newsgroups(subset='train', categories=cats)

>>> list(newsgroups_train.target_names)
['alt.atheism', 'sci.space']
>>> newsgroups_train.filenames.shape
(1073,)
>>> newsgroups_train.target.shape
(1073,)
>>> newsgroups_train.target[:10]
array([1, 1, 1, 0, 1, 0, 0, 1, 1, 1])
```

5.7.2. Converting text to vectors

In order to feed predictive or clustering models with the text data, one first need to turn the text into vectors of numerical values suitable for statistical analysis. This can be achieved with the utilities of the `sklearn.feature_extraction.text` as demonstrated in the following example that extract TF-IDF vectors of unigram tokens from a subset of 20news:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> categories = ['alt.atheism', 'talk.religion.misc',
...               'comp.graphics', 'sci.space']
>>> newsgroups_train = fetch_20newsgroups(subset='train',
...                                       categories=categories)
>>> vectorizer = TfidfVectorizer()
>>> vectors = vectorizer.fit_transform(newsgroups_train.data)
>>> vectors.shape
(2034, 34118)
```

The extracted TF-IDF vectors are very sparse, with an average of 159 non-zero components by sample in a more than 30000-dimensional space (less than .5% non-zero features):

```
>>> vectors.nnz / float(vectors.shape[0])
159.01327433628319
```

`sklearn.datasets.fetch_20newsgroups_vectorized` is a function which returns ready-to-use tfidf features instead of file names.

5.7.3. Filtering text for more realistic training

It is easy for a classifier to overfit on particular things that appear in the 20 Newsgroups data, such as newsgroup headers. Many classifiers achieve very high F-scores, but their results would not generalize to other documents that aren't from this window of time.

For example, let's look at the results of a multinomial Naive Bayes classifier, which is fast to train and achieves a decent F-score:

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> from sklearn import metrics
>>> newsgroups_test = fetch_20newsgroups(subset='test',
...                                      categories=categories)
>>> vectors_test = vectorizer.transform(newsgroups_test.data)
>>> clf = MultinomialNB(alpha=.01)
>>> clf.fit(vectors, newsgroups_train.target)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred)
0.88251152461278892
```

(The example *Classification of text documents using sparse features* shuffles the training and test data, instead of segmenting by time, and in that case multinomial Naive Bayes gets a much higher F-score of 0.88. Are you suspicious yet of what's going on inside this classifier?)

Let's take a look at what the most informative features are:

```
>>> import numpy as np
>>> def show_top10(classifier, vectorizer, categories):
...     feature_names = np.asarray(vectorizer.get_feature_names())
...     for i, category in enumerate(categories):
...         top10 = np.argsort(classifier.coef_[i])[-10:]
...         print("%s: %s" % (category, " ".join(feature_names[top10])))
...
>>> show_top10(clf, vectorizer, newsgroups_train.target_names)
alt.atheism: sgi livesey atheists writes people caltech com god keith edu
comp.graphics: organization thanks files subject com image lines university edu graphics
sci.space: toronto moon gov com alaska access henry nasa edu space
talk.religion.misc: article writes kent people christian jesus sandvik edu com god
```

You can now see many things that these features have overfit to:

- Almost every group is distinguished by whether headers such as `NNTP-Posting-Host:` and `Distribution:` appear more or less often.
- Another significant feature involves whether the sender is affiliated with a university, as indicated either by their headers or their signature.
- The word "article" is a significant feature, based on how often people quote previous posts like this: "In article [article ID], [name] <[e-mail address]> wrote:"
- Other features match the names and e-mail addresses of particular people who were posting at the time.

With such an abundance of clues that distinguish newsgroups, the classifiers barely have to identify topics from text at all, and they all perform at the same high level.

For this reason, the functions that load 20 Newsgroups data provide a parameter called **remove**, telling it what kinds of information to strip out of each file. **remove** should be a tuple containing any subset of `('headers', 'footers', 'quotes')`, telling it to remove headers, signature blocks, and quotation blocks respectively.

```
>>> newsgroups_test = fetch_20newsgroups(subset='test',
...                                      remove=('headers', 'footers', 'quotes'),
...                                      categories=categories)
>>> vectors_test = vectorizer.transform(newsgroups_test.data)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(pred, newsgroups_test.target)
0.78409163025839435
```

This classifier lost over a lot of its F-score, just because we removed metadata that has little to do with topic

classification. It loses even more if we also strip this metadata from the training data:

```
>>> newsgroups_train = fetch_20newsgroups(subset='train',
...                                       remove=('headers', 'footers', 'quotes'),
...                                       categories=categories)
>>> vectors = vectorizer.fit_transform(newsgroups_train.data)
>>> clf = BernoulliNB(alpha=.01)
>>> clf.fit(vectors, newsgroups_train.target)
>>> vectors_test = vectorizer.transform(newsgroups_test.data)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred)
0.73160869205141166
```

Some other classifiers cope better with this harder version of the task. Try running *Sample pipeline for text feature extraction and evaluation* with and without the `--filter` option to compare the results.

Recommendation

When evaluating text classifiers on the 20 Newsgroups data, you should strip newsgroup-related metadata. In scikit-learn, you can do this by setting `remove=('headers', 'footers', 'quotes')`. The F-score will be lower because it is more realistic.

Examples

- *Sample pipeline for text feature extraction and evaluation*
- *Classification of text documents using sparse features*

5.8. Downloading datasets from the mldata.org repository

mldata.org is a public repository for machine learning data, supported by the PASCAL network .

The `sklearn.datasets` package is able to directly download data sets from the repository using the function `fetch_mldata(dataname)`.

For example, to download the MNIST digit recognition database:

```
>>> from sklearn.datasets import fetch_mldata
>>> mnist = fetch_mldata('MNIST original', data_home=custom_data_home)
```

The MNIST database contains a total of 70000 examples of handwritten digits of size 28x28 pixels, labeled from 0 to 9:

```
>>> mnist.data.shape
(70000, 784)
>>> mnist.target.shape
(70000,)
>>> np.unique(mnist.target)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

After the first download, the dataset is cached locally in the path specified by the `data_home` keyword argument, which defaults to `~/scikit_learn_data/`:

```
>>> os.listdir(os.path.join(custom_data_home, 'mldata'))
['mnist-original.mat']
```

Data sets in mldata.org do not adhere to a strict naming or formatting convention. `fetch_mldata` is able to make sense of the most common cases, but allows to tailor the defaults to individual datasets:

- The data arrays in mldata.org are most often shaped as `(n_features, n_samples)`. This is the opposite of the `scikit-learn` convention, so `fetch_mldata` transposes the matrix by default. The `transpose_data` keyword controls this behavior:

```
>>> iris = fetch_mldata('iris', data_home=custom_data_home)
>>> iris.data.shape
(150, 4)
>>> iris = fetch_mldata('iris', transpose_data=False,
...                     data_home=custom_data_home)
>>> iris.data.shape
(4, 150)
```

- For datasets with multiple columns, `fetch_mldata` tries to identify the target and data columns and rename them to `target` and `data`. This is done by looking for arrays named `label` and `data` in the dataset, and failing that by choosing the first array to be `target` and the second to be `data`. This behavior can be changed with the `target_name` and `data_name` keywords, setting them to a specific name or index number (the name and order of the columns in the datasets can be found at its mldata.org under the tab “Data”:

```
>>> iris2 = fetch_mldata('datasets-UCI iris', target_name=1, data_name=0,
...                      data_home=custom_data_home)
>>> iris3 = fetch_mldata('datasets-UCI iris', target_name='class',
...                      data_name='double0', data_home=custom_data_home)
```

5.9. The Labeled Faces in the Wild face recognition dataset

This dataset is a collection of JPEG pictures of famous people collected over the internet, all details are available on the official website:

> http://vis-www.cs.umass.edu/lfw/

Each picture is centered on a single face. The typical task is called Face Verification: given a pair of two pictures, a binary classifier must predict whether the two images are from the same person.

An alternative task, Face Recognition or Face Identification is: given the picture of the face of an unknown person, identify the name of the person by referring to a gallery of previously seen pictures of identified persons.

Both Face Verification and Face Recognition are tasks that are typically performed on the output of a model trained to perform Face Detection. The most popular model for Face Detection is called Viola-Jones and is implemented in the OpenCV library. The LFW faces were extracted by this face detector from various online websites.

5.9.1. Usage

`scikit-learn` provides two loaders that will automatically download, cache, parse the metadata files, decode the jpeg and convert the interesting slices into memmaped numpy arrays. This dataset size is more than 200 MB. The first load typically takes more than a couple of minutes to fully decode the relevant part of the JPEG files into numpy arrays. If the dataset has been loaded once, the following times the loading times less than 200ms by using a memmaped version memoized on the disk in the `~/scikit_learn_data/lfw_home/` folder using `joblib`.

The first loader is used for the Face Identification task: a multi-class classification task (hence supervised learning):

```
>>> from sklearn.datasets import fetch_lfw_people
>>> lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

>>> for name in lfw_people.target_names:
...     print(name)
...
Ariel Sharon
Colin Powell
Donald Rumsfeld
George W Bush
Gerhard Schroeder
Hugo Chavez
Tony Blair
```

The default slice is a rectangular shape around the face, removing most of the background:

```
>>> lfw_people.data.dtype
dtype('float32')

>>> lfw_people.data.shape
(1288, 1850)

>>> lfw_people.images.shape
(1288, 50, 37)
```

Each of the `1140` faces is assigned to a single person id in the `target` array:

```
>>> lfw_people.target.shape
(1288,)

>>> list(lfw_people.target[:10])
[5, 6, 3, 1, 0, 1, 3, 4, 3, 0]
```

The second loader is typically used for the face verification task: each sample is a pair of two picture belonging or not to the same person:

```
>>> from sklearn.datasets import fetch_lfw_pairs
>>> lfw_pairs_train = fetch_lfw_pairs(subset='train')

>>> list(lfw_pairs_train.target_names)
['Different persons', 'Same person']

>>> lfw_pairs_train.pairs.shape
(2200, 2, 62, 47)

>>> lfw_pairs_train.data.shape
(2200, 5828)

>>> lfw_pairs_train.target.shape
(2200,)
```

Both for the `fetch_lfw_people` and `fetch_lfw_pairs` function it is possible to get an additional dimension with the RGB color channels by passing `color=True`, in that case the shape will be `(2200, 2, 62, 47, 3)`.

The `fetch_lfw_pairs` datasets is subdivided into 3 subsets: the development `train` set, the development `test` set and an evaluation `10_folds` set meant to compute performance metrics using a 10-folds cross validation scheme.

References:

- Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. University of Massachusetts, Amherst, Technical Report 07-49, October, 2007.

5.9.2. Examples

Faces recognition example using eigenfaces and SVMs

5.10. Forest covertypes

The samples in this dataset correspond to 30×30m patches of forest in the US, collected for the task of predicting each patch's cover type, i.e. the dominant species of tree. There are seven covertypes, making this a multiclass classification problem. Each sample has 54 features, described on the dataset's homepage. Some of the features are boolean indicators, while others are discrete or continuous measurements.

`sklearn.datasets.fetch_covtype` will load the covertype dataset; it returns a dictionary-like object with the feature matrix in the `data` member and the target values in `target`. The dataset will be downloaded from the web if necessary.

6. Strategies to scale computationally: bigger data

For some applications the amount of examples, features (or both) and/or the speed at which they need to be processed are challenging for traditional approaches. In these cases scikit-learn has a number of options you can consider to make your system scale.

6.1. Scaling with instances using out-of-core learning

Out-of-core (or “external memory”) learning is a technique used to learn from data that cannot fit in a computer’s main memory (RAM).

Here is sketch of a system designed to achieve this goal:

1. a way to stream instances
2. a way to extract features from instances
3. an incremental algorithm

6.1.1. Streaming instances

Basically, 1. may be a reader that yields instances from files on a hard drive, a database, from a network stream etc. However, details on how to achieve this are beyond the scope of this documentation.

6.1.2. Extracting features

2. could be any relevant way to extract features among the different *feature extraction* methods supported by scikit-learn. However, when working with data that needs vectorization and where the set of features or values is not known in advance one should take explicit care. A good example is text classification where unknown terms are likely to be found during training. It is possible to use a statefull vectorizer if making multiple passes over the data is reasonable from an application point of view. Otherwise, one can turn up the difficulty by using a stateless feature extractor. Currently the preferred way to do this is to use the so-called *hashing trick* as implemented by `sklearn.feature_extraction.FeatureHasher` for datasets with categorical variables represented as list of Python dicts or `sklearn.feature_extraction.text.HashingVectorizer` for text documents.

6.1.3. Incremental learning

Finally, for 3. we have a number of options inside scikit-learn. Although all algorithms cannot learn incrementally (i.e. without seeing all the instances at once), all estimators implementing the `partial_fit` API are candidates. Actually, the ability to learn incrementally from a mini-batch of instances (sometimes called “online learning”) is key to out-of-core learning as it guarantees that at any given time there will be only a small amount of instances in the main memory. Choosing a good size for the mini-batch that balances relevancy and memory footprint could involve some tuning [1].

Here is a list of incremental estimators for different tasks:

- Classification
  - `sklearn.naive_bayes.MultinomialNB`
  - `sklearn.naive_bayes.BernoulliNB`
  - `sklearn.linear_model.Perceptron`
  - `sklearn.linear_model.SGDClassifier`
  - `sklearn.linear_model.PassiveAggressiveClassifier`
- Regression
  - `sklearn.linear_model.SGDRegressor`
  - `sklearn.linear_model.PassiveAggressiveRegressor`
- Clustering
  - `sklearn.cluster.MiniBatchKMeans`
- Decomposition / feature Extraction
  - `sklearn.decomposition.MiniBatchDictionaryLearning`
  - `sklearn.cluster.MiniBatchKMeans`

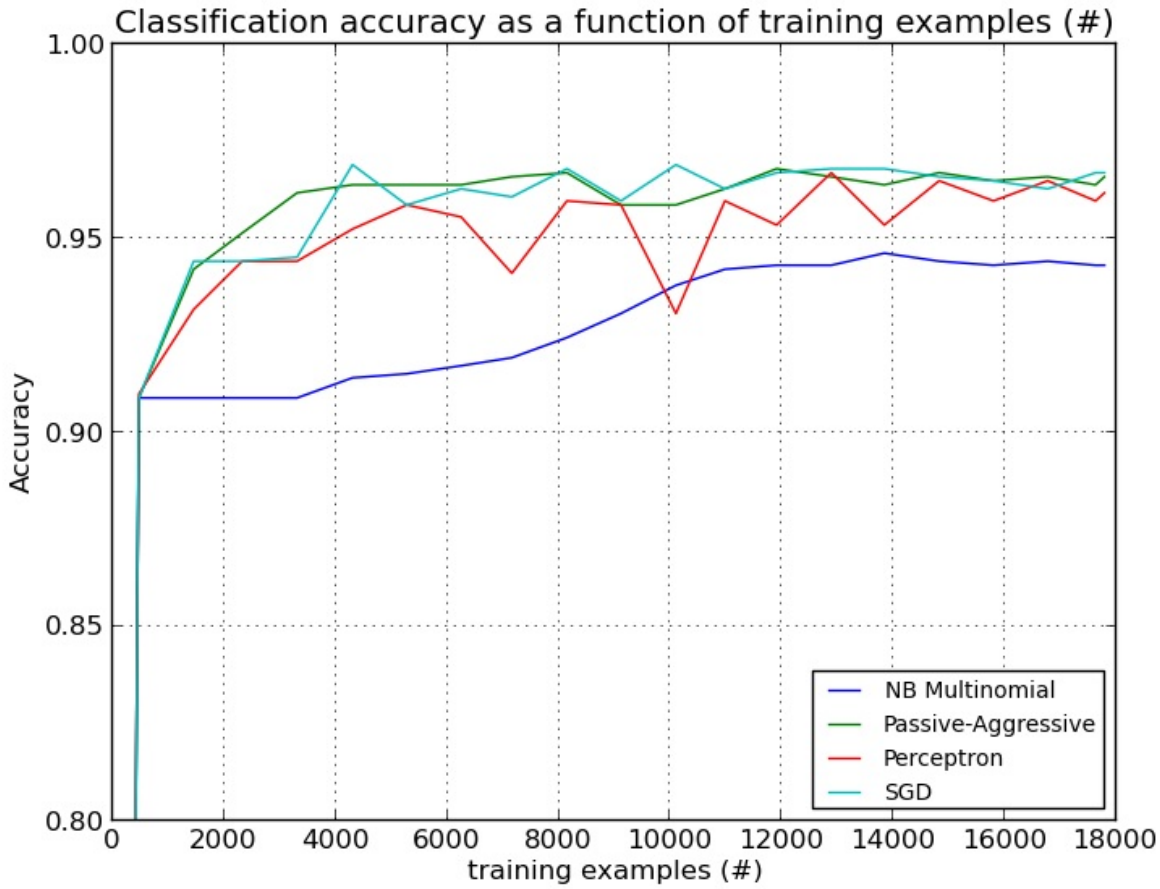
For classification, a somewhat important thing to note is that although a stateless feature extraction routine may be able to cope with new/unseen attributes, the incremental learner itself may be unable to cope with new/unseen targets classes. In this case you have to pass all the possible classes to the first `partial_fit` call using the `classes=` parameter.

Another aspect to consider when choosing a proper algorithm is that all of them don't put the same importance on each example over time. Namely, the `Perceptron` is still sensitive to badly labeled examples even after many examples whereas the `SGD*` and `PassiveAggressive*` families are more robust to this kind of artifacts. Conversely, the later also tend to give less importance to remarkably different, yet properly labeled examples when they come late in the stream as their learning rate decreases over time.

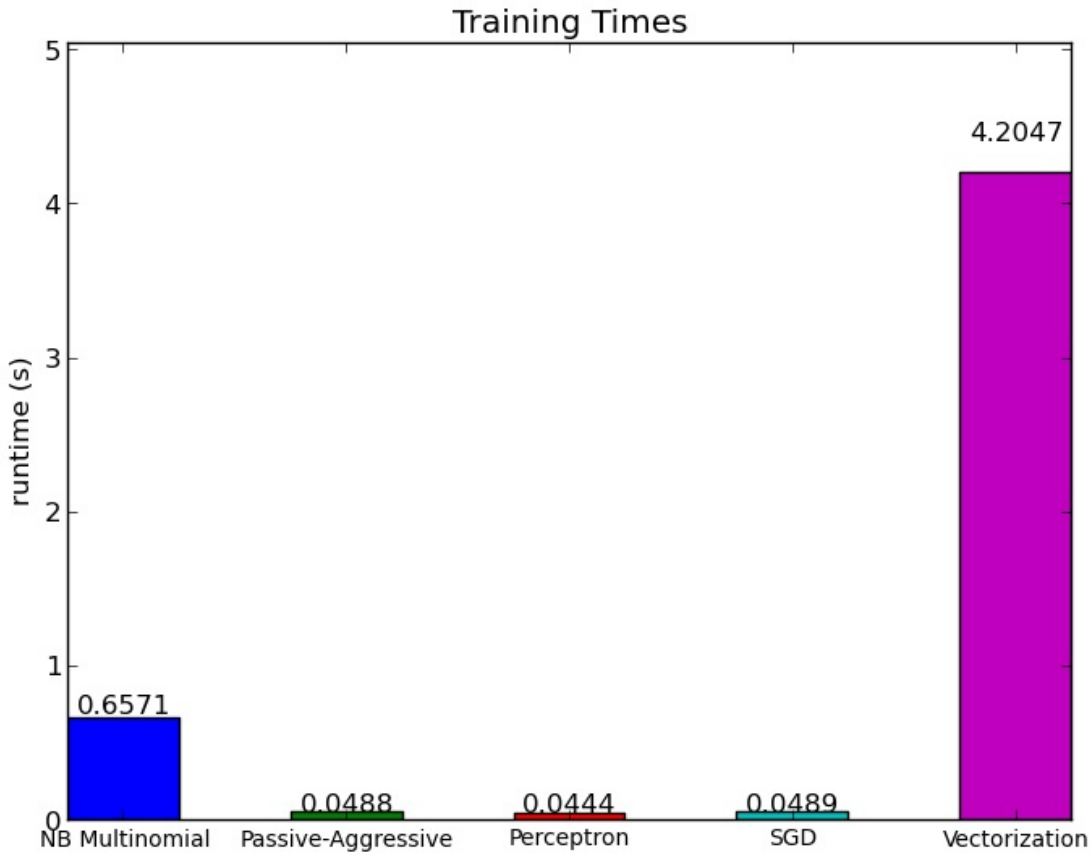
6.1.4. Examples

Finally, we have a full-fledged example of *Out-of-core classification of text documents*. It is aimed at providing a starting point for people wanting to build out-of-core learning systems and demonstrates most of the notions discussed above.

Furthermore, it also shows the evolution of the performance of different algorithms with the number of processed examples.



Now looking at the computation time of the different parts, we see that the vectorization is much more expensive than learning itself. From the different algorithms, `MultinomialNB` is the most expensive, but its overhead can be mitigated by increasing the size of the mini-batches (exercise: change `minibatch_size` to 100 and 10000 in the program and compare).



6.1.5. Notes

[1] Depending on the algorithm the mini-batch size can influence results or not. SGD*, PassiveAggressive*, and discrete NaiveBayes are truly online and are not affected by batch size. Conversely, MiniBatchKMeans convergence rate is affected by the batch size. Also, its memory footprint can vary dramatically with batch size.

Previous Next

7. Computational Performance

For some applications the performance (mainly latency and throughput at prediction time) of estimators is crucial. It may also be of interest to consider the training throughput but this is often less important in a production setup (where it often takes place offline).

We will review here the orders of magnitude you can expect from a number of scikit-learn estimators in different contexts and provide some tips and tricks for overcoming performance bottlenecks.

Prediction latency is measured as the elapsed time necessary to make a prediction (e.g. in micro-seconds). Latency is often viewed as a distribution and operations engineers often focus on the latency at a given percentile of this distribution (e.g. the 90 percentile).

Prediction throughput is defined as the number of predictions the software can deliver in a given amount of time (e.g. in predictions per second).

An important aspect of performance optimization is also that it can hurt prediction accuracy. Indeed, simpler models (e.g. linear instead of non-linear, or with fewer parameters) often run faster but are not always able to take into account the same exact properties of the data as more complex ones.

7.1. Prediction Latency

One of the most straight-forward concerns one may have when using/choosing a machine learning toolkit is the latency at which predictions can be made in a production environment.

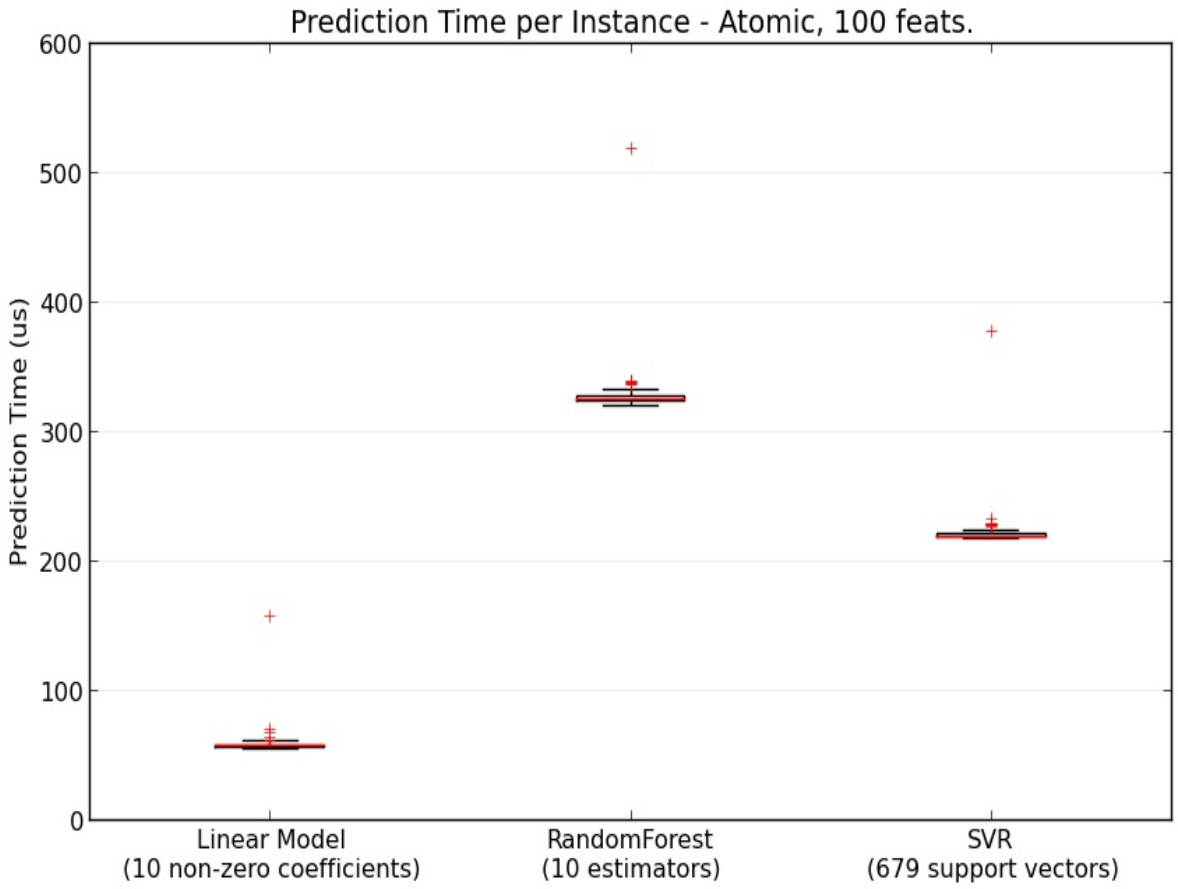
The main factors that influence the prediction latency are

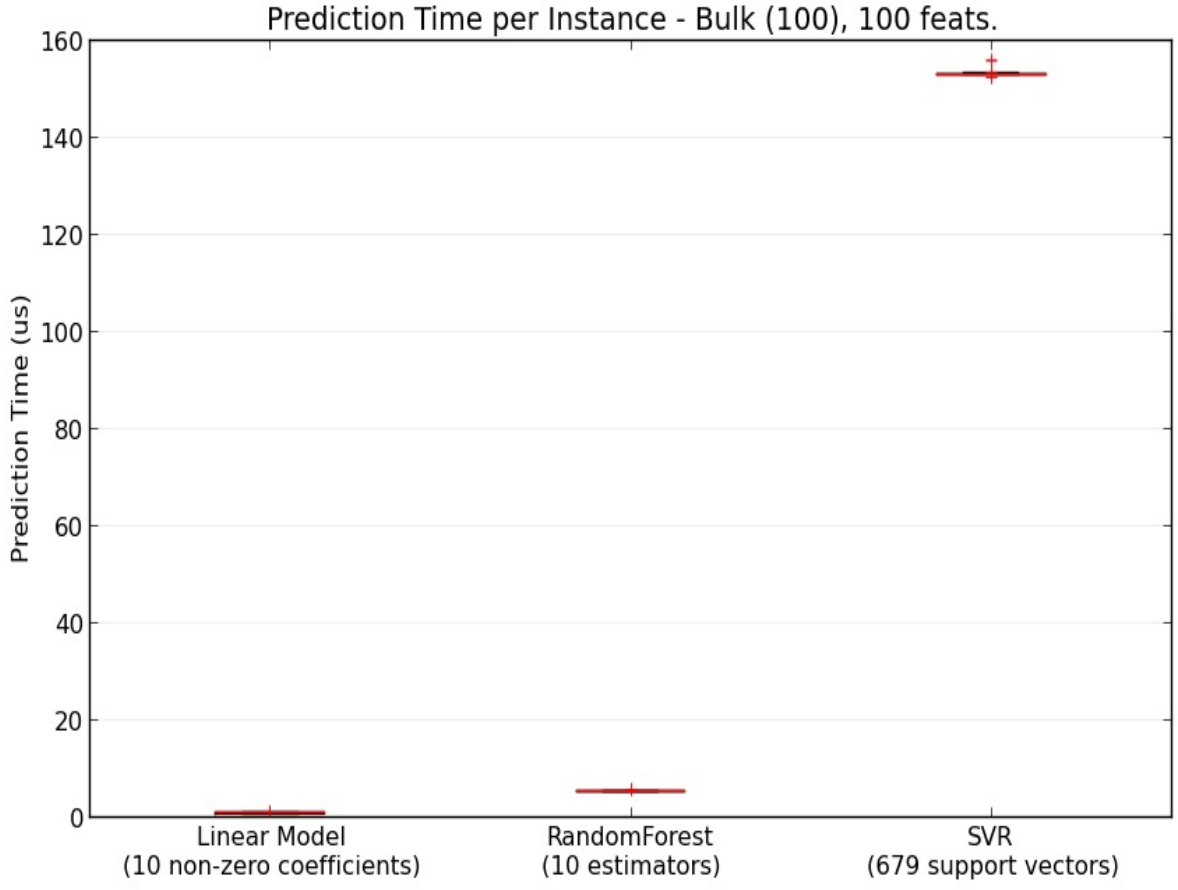
1. Number of features
2. Input data representation and sparsity
3. Model complexity
4. Feature extraction

A last major parameter is also the possibility to do predictions in bulk or one-at-a-time mode.

7.1.1. Bulk versus Atomic mode

In general doing predictions in bulk (many instances at the same time) is more efficient for a number of reasons (branching predictability, CPU cache, linear algebra libraries optimizations etc.). Here we see on a setting with few features that independently of estimator choice the bulk mode is always faster, and for some of them by 1 to 2 orders of magnitude:



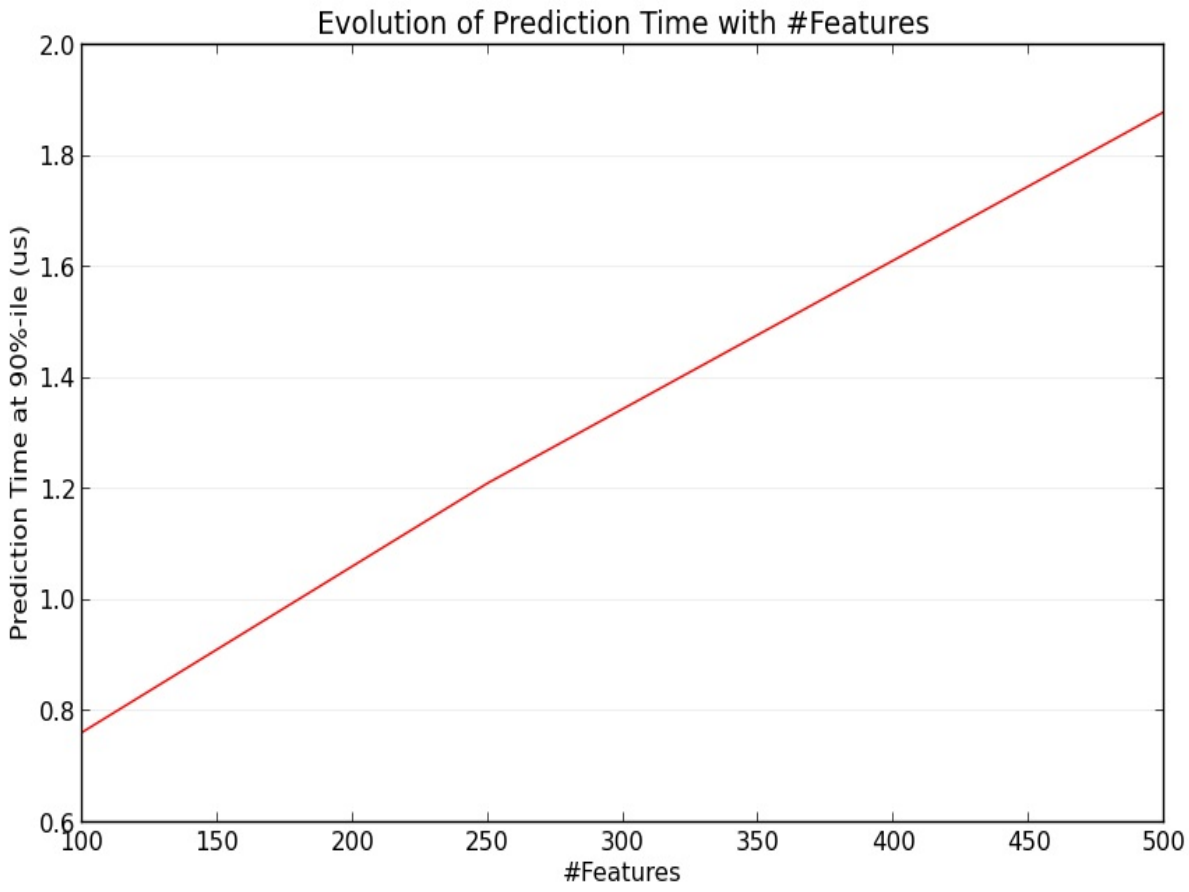


To benchmark different estimators for your case you can simply change the `n_features` parameter in this

example: *Prediction Latency*. This should give you an estimate of the order of magnitude of the prediction latency.

7.1.2. Influence of the Number of Features

Obviously when the number of features increases so does the memory consumption of each example. Indeed, for a matrix of M instances with N features, the space complexity is in $O(NM)$. From a computing perspective it also means that the number of basic operations (e.g., multiplications for vector-matrix products in linear models) increases too. Here is a graph of the evolution of the prediction latency with the number of features:



Overall you can expect the prediction time to increase at least linearly with the number of features (non-linear cases can happen depending on the global memory footprint and estimator).

7.1.3. Influence of the Input Data Representation

Scipy provides sparse matrix datastructures which are optimized for storing sparse data. The main feature of sparse formats is that you don't store zeros so if your data is sparse then you use much less memory. A non-zero value in a sparse (CSR or CSC) representation will only take on average one 32bit integer position + the 64 bit floating point value + an additional 32bit per row or column in the matrix. Using sparse input on a dense (or sparse) linear model can speedup prediction by quite a bit as only the non zero valued features impact the dot product and thus the model predictions. Hence if you have 100 non zeros in 1e6 dimensional space, you only need 100 multiply and add operation instead of 1e6.

Calculation over a dense representation, however, may leverage highly optimised vector operations and

multithreading in BLAS, and tends to result in fewer CPU cache misses. So the sparsity should typically be quite high (10% non-zeros max, to be checked depending on the hardware) for the sparse input representation to be faster than the dense input representation on a machine with many CPUs and an optimized BLAS implementation.

Here is sample code to test the sparsity of your input:

```
def sparsity_ratio(X):
    return 1.0 - np.count_nonzero(X) / float(X.shape[0] * X.shape[1])
print("input sparsity ratio:", sparsity_ratio(X))
```

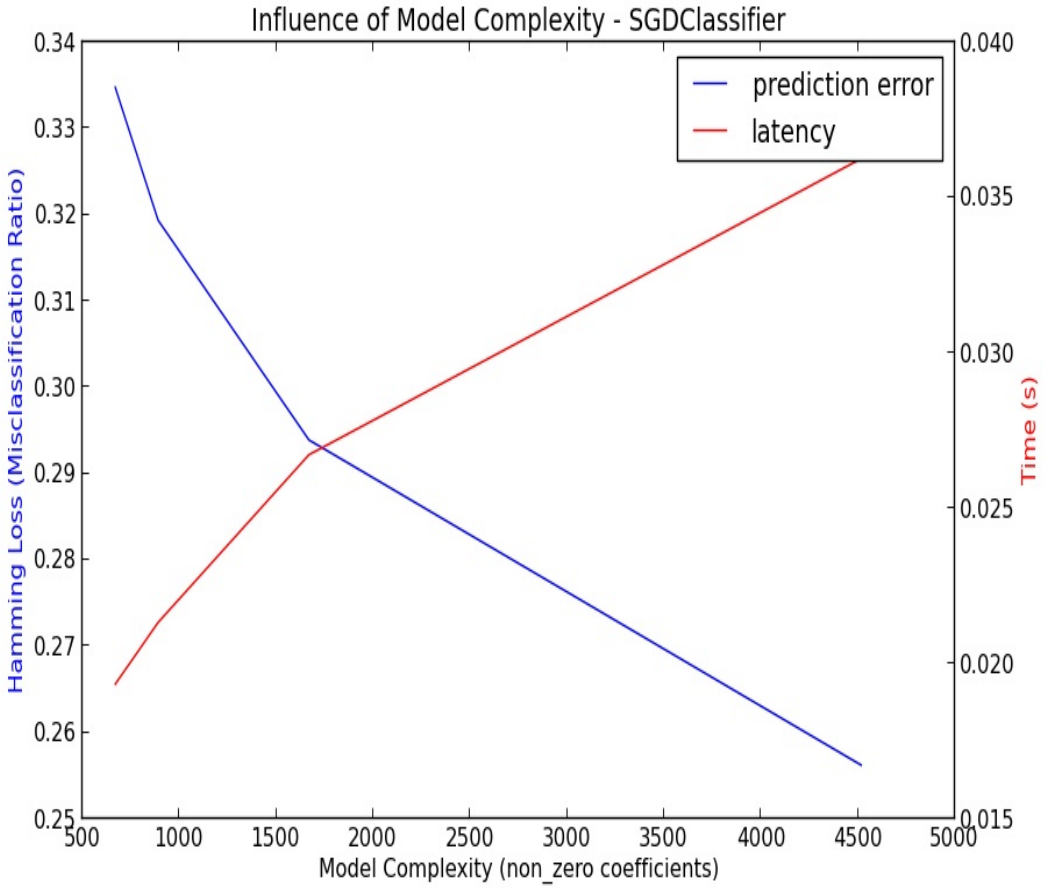
As a rule of thumb you can consider that if the sparsity ratio is greater than 90% you can probably benefit from sparse formats. Check Scipy's sparse matrix formats documentation for more information on how to build (or convert your data to) sparse matrix formats. Most of the time the `CSR` and `CSC` formats work best.

7.1.4. Influence of the Model Complexity

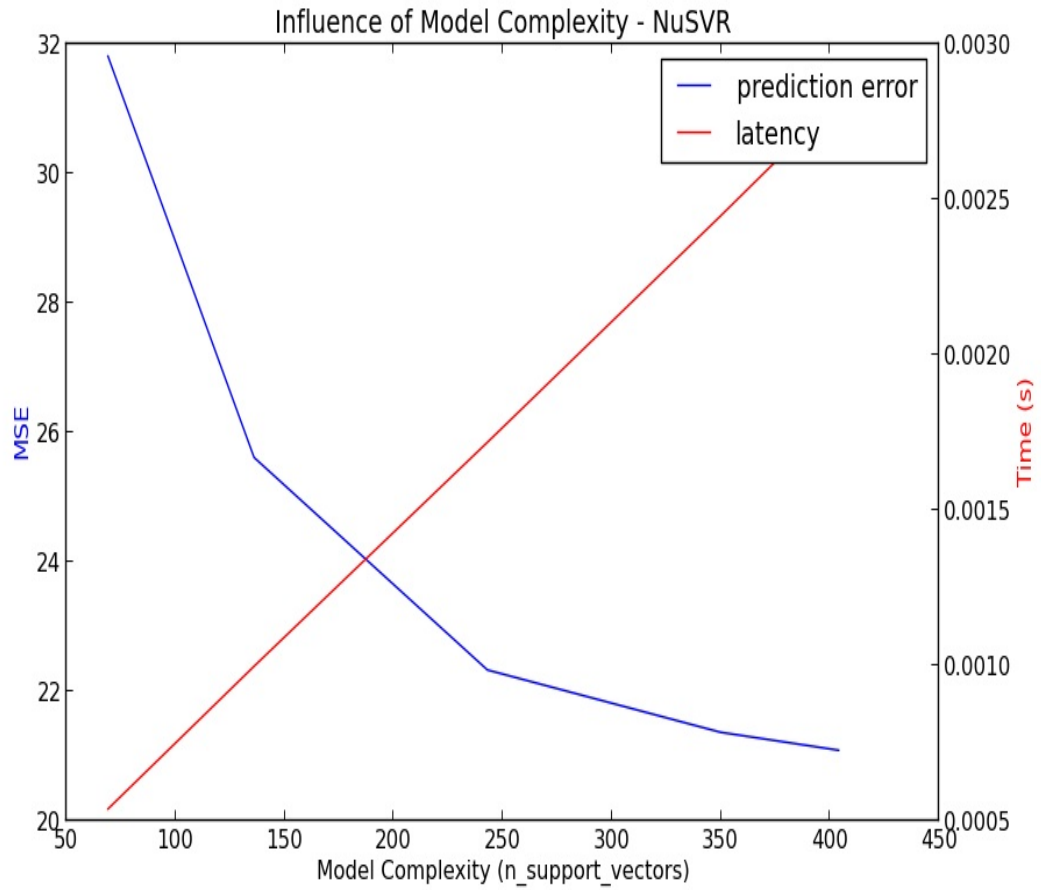
Generally speaking, when model complexity increases, predictive power and latency are supposed to increase. Increasing predictive power is usually interesting, but for many applications we would better not increase prediction latency too much. We will now review this idea for different families of supervised models.

For **`sklearn.linear_model`** (e.g. Lasso, ElasticNet, SGDClassifier/Regressor, Ridge & RidgeClassifier, PassiveAgressiveClassifier/Regressor, LinearSVC, LogisticRegression...) the decision function that is applied at prediction time is the same (a dot product) , so latency should be equivalent.

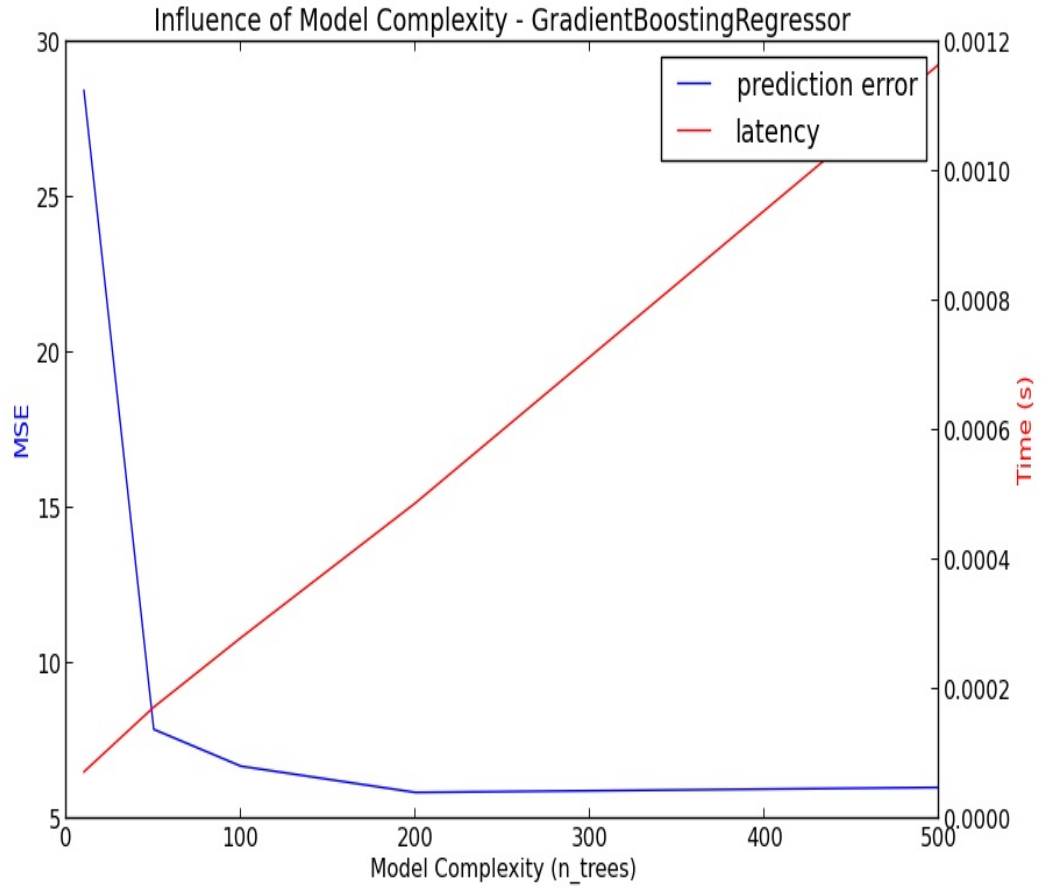
Here is an example using **`sklearn.linear_model.stochastic_gradient.SGDClassifier`** with the `elasticnet` penalty. The regularization strength is globally controlled by the `alpha` parameter. With a sufficiently high `alpha`, one can then increase the `l1_ratio` parameter of `elasticnet` to enforce various levels of sparsity in the model coefficients. Higher sparsity here is interpreted as less model complexity as we need fewer coefficients to describe it fully. Of course sparsity influences in turn the prediction time as the sparse dot-product takes time roughly proportional to the number of non-zero coefficients.



For the **`sklearn.svm`** family of algorithms with a non-linear kernel, the latency is tied to the number of support vectors (the fewer the faster). Latency and throughput should (asymptotically) grow linearly with the number of support vectors in a SVC or SVR model. The kernel will also influence the latency as it is used to compute the projection of the input vector once per support vector. In the following graph the `nu` parameter of **`sklearn.svm.classes.NuSVR`** was used to influence the number of support vectors.



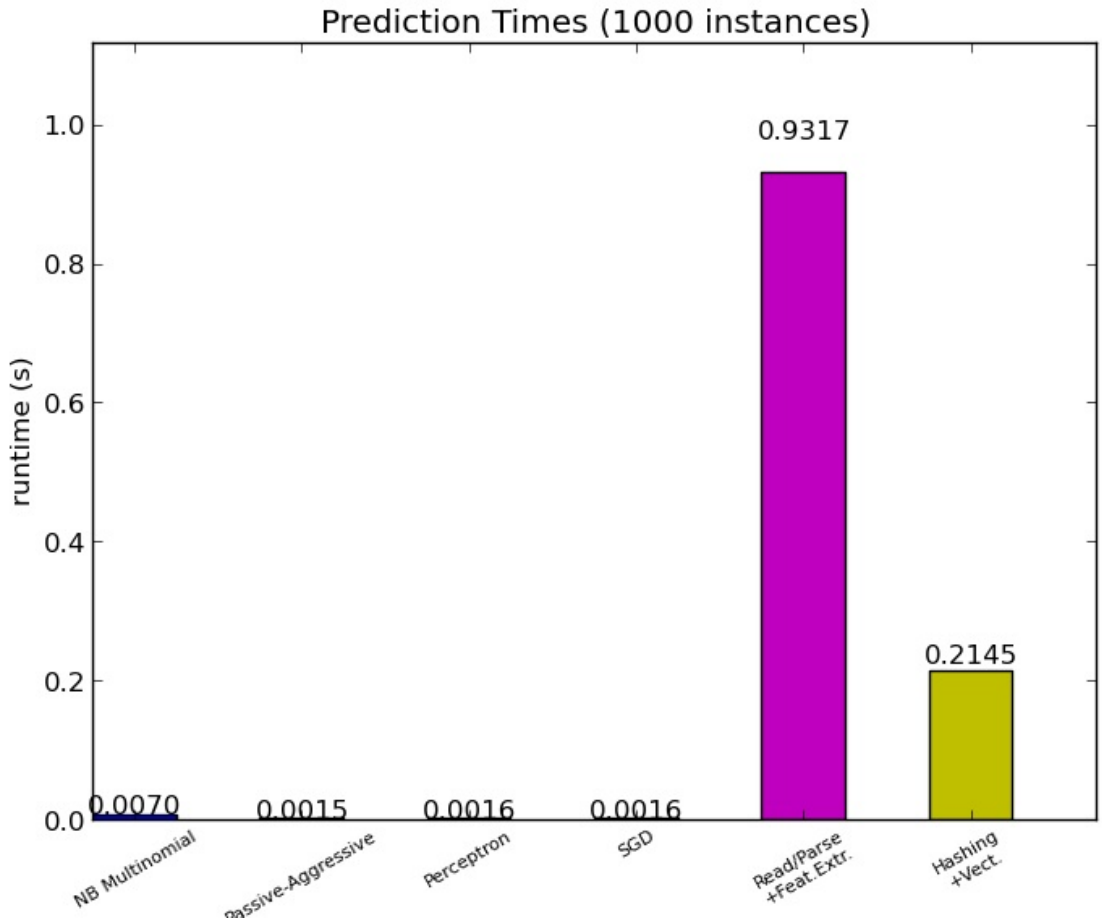
For **`sklearn.ensemble`** of trees (e.g. RandomForest, GBT, ExtraTrees etc) the number of trees and their depth play the most important role. Latency and throughput should scale linearly with the number of trees. In this case we used directly the `n_estimators` parameter of **`sklearn.ensemble.gradient_boosting.GradientBoostingRegressor`**.



In any case be warned that decreasing model complexity can hurt accuracy as mentioned above. For instance a non-linearly separable problem can be handled with a speedy linear model but prediction power will very likely suffer in the process.

7.1.5. Feature Extraction Latency

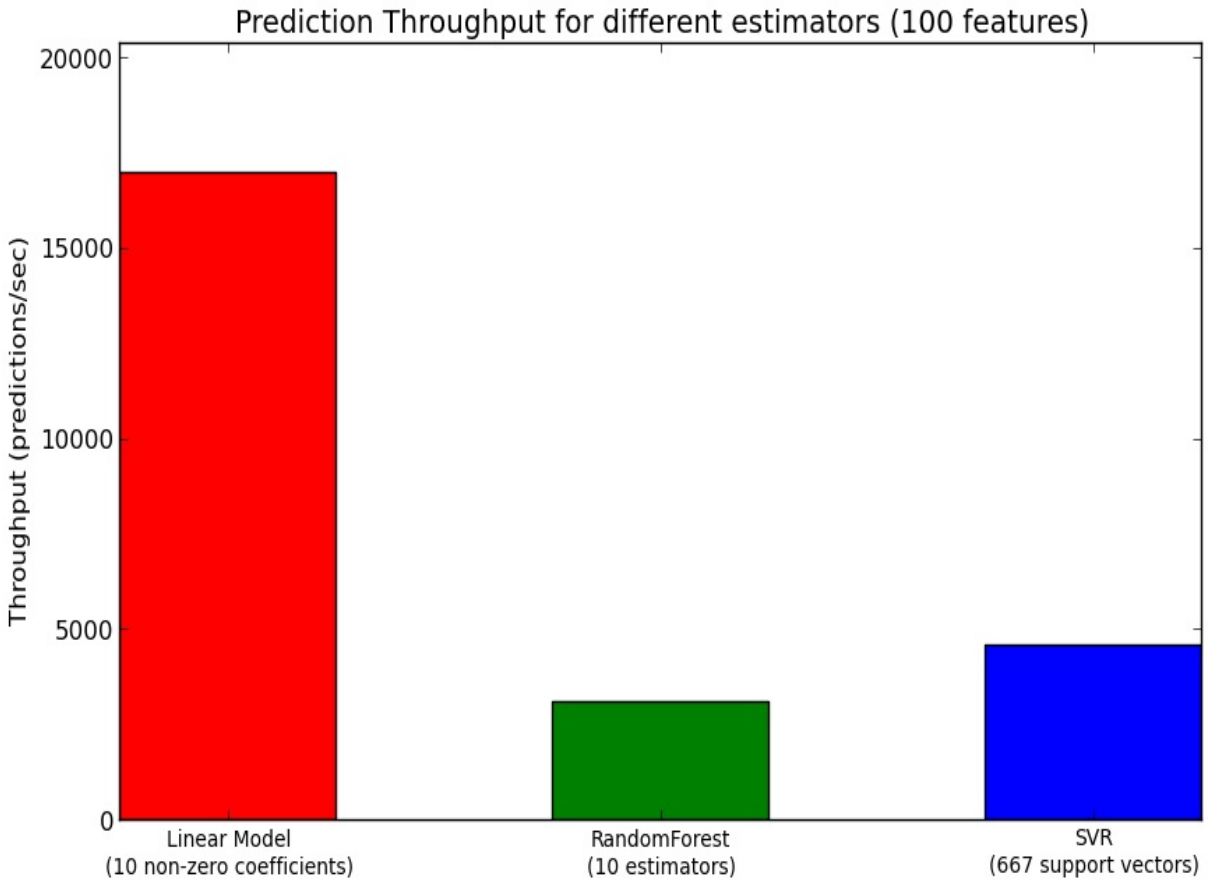
Most scikit-learn models are usually pretty fast as they are implemented either with compiled Cython extensions or optimized computing libraries. On the other hand, in many real world applications the feature extraction process (i.e. turning raw data like database rows or network packets into numpy arrays) governs the overall prediction time. For example on the Reuters text classification task the whole preparation (reading and parsing SGML files, tokenizing the text and hashing it into a common vector space) is taking 100 to 500 times more time than the actual prediction code, depending on the chosen model.



In many cases it is thus recommended to carefully time and profile your feature extraction code as it may be a good place to start optimizing when your overall latency is too slow for your application.

7.2. Prediction Throughput

Another important metric to care about when sizing production systems is the throughput i.e. the number of predictions you can make in a given amount of time. Here is a benchmark from the *Prediction Latency* example that measures this quantity for a number of estimators on synthetic data:



These throughputs are achieved on a single process. An obvious way to increase the throughput of your application is to spawn additional instances (usually processes in Python because of the GIL) that share the same model. One might also add machines to spread the load. A detailed explanation on how to achieve this is beyond the scope of this documentation though.

7.3. Tips and Tricks

7.3.1. Linear algebra libraries

As scikit-learn relies heavily on Numpy/Scipy and linear algebra in general it makes sense to take explicit care of the versions of these libraries. Basically, you ought to make sure that Numpy is built using an optimized BLAS / LAPACK library.

Not all models benefit from optimized BLAS and Lapack implementations. For instance models based on (randomized) decision trees typically do not rely on BLAS calls in their inner loops, nor do kernel SVMs (`SVC`, `SVR`, `NuSVC`, `NuSVR`). On the other hand a linear model implemented with a BLAS DGEMM call (via `numpy.dot`) will typically benefit hugely from a tuned BLAS implementation and lead to orders of magnitude speedup over a non-optimized BLAS.

You can display the BLAS / LAPACK implementation used by your NumPy / SciPy / scikit-learn install with the following commands:

```
from numpy.distutils.system_info import get_info
print(get_info('blas_opt'))
print(get_info('lapack_opt'))
```

Optimized BLAS / LAPACK implementations include:

- Atlas (need hardware specific tuning by rebuilding on the target machine)
- OpenBLAS
- MKL
- Apple Accelerate and vecLib frameworks (OSX only)

More information can be found on the Scipy install page and in this blog post from Daniel Nouri which has some nice step by step install instructions for Debian / Ubuntu.

Warning: Multithreaded BLAS libraries sometimes conflict with Python's `multiprocessing` module, which is used by e.g. `GridSearchCV` and most other estimators that take an `n_jobs` argument (with the exception of `SGDClassifier`, `SGDRegressor`, `Perceptron`, `PassiveAggressiveClassifier` and tree-based methods such as random forests). This is true of Apple's Accelerate and OpenBLAS when built with OpenMP support.

Besides scikit-learn, NumPy and SciPy also use BLAS internally, as explained earlier.

If you experience hanging subprocesses with `n_jobs>1` or `n_jobs=-1`, make sure you have a single-threaded BLAS library, or set `n_jobs=1`, or upgrade to Python 3.4 which has a new version of `multiprocessing` that should be immune to this problem.

7.3.2. Model Compression

Model compression in scikit-learn only concerns linear models for the moment. In this context it means that we want to control the model sparsity (i.e. the number of non-zero coordinates in the model vectors). It is generally a good idea to combine model sparsity with sparse input data representation.

Here is sample code that illustrates the use of the `sparsify()` method:

```
clf = SGDRegressor(penalty='elasticnet', l1_ratio=0.25)
clf.fit(X_train, y_train).sparsify()
clf.predict(X_test)
```

In this example we prefer the `elasticnet` penalty as it is often a good compromise between model compactness and prediction power. One can also further tune the `l1_ratio` parameter (in combination with the regularization strength `alpha`) to control this tradeoff.

A typical benchmark on synthetic data yields a >30% decrease in latency when both the model and input are sparse (with 0.000024 and 0.027400 non-zero coefficients ratio respectively). Your mileage may vary depending on the sparsity and size of your data and model. Furthermore, sparsifying can be very useful to reduce the memory usage of predictive models deployed on production servers.

7.3.3. Model Reshaping

Model reshaping consists in selecting only a portion of the available features to fit a model. In other words, if a model discards features during the learning phase we can then strip those from the input. This has several benefits. Firstly it reduces memory (and therefore time) overhead of the model itself. It also allows to discard explicit feature selection components in a pipeline once we know which features to keep from a previous run. Finally, it can help reduce processing time and I/O usage upstream in the data access and feature extraction layers by not collecting and building features that are discarded by the model. For instance if the raw data come from a database, it can make it possible to write simpler and faster queries or reduce I/O usage by

making the queries return lighter records. At the moment, reshaping needs to be performed manually in scikit-learn. In the case of sparse input (particularly in `CSR` format), it is generally sufficient to not generate the relevant features, leaving their columns empty.

7.3.4. Links

- scikit-learn developer performance documentation
- Scipy sparse matrix formats documentation

Reference

This is the class and function reference of scikit-learn. Please refer to the *full user guide* for further details, as the class and function raw specifications may not be enough to give full guidelines on their uses.

`sklearn.base`: Base classes and utility functions

Base classes for all estimators.

Base classes

`base.BaseEstimator`	Base class for all estimators in scikit-learn
`base.ClassifierMixin`	Mixin class for all classifiers in scikit-learn.
`base.ClusterMixin`	Mixin class for all cluster estimators in scikit-learn.
`base.RegressorMixin`	Mixin class for all regression estimators in scikit-learn.
`base.TransformerMixin`	Mixin class for all transformers in scikit-learn.

Functions

`base.clone`(estimator[, safe])	Constructs a new estimator with the same parameters.

`sklearn.cluster`: Clustering

The `sklearn.cluster` module gathers popular unsupervised clustering algorithms.

User guide: See the *Clustering* section for further details.

Classes

`cluster.AffinityPropagation`([damping, ...])	Perform Affinity Propagation Clustering of data.
`cluster.AgglomerativeClustering`([...])	Agglomerative Clustering
`cluster.DBSCAN`([eps, min_samples, metric, ...])	Perform DBSCAN clustering from vector array or distance matrix.
`cluster.FeatureAgglomeration`([n_clusters, ...])	
`cluster.KMeans`([n_clusters, init, n_init, ...])	K-Means clustering
`cluster.MiniBatchKMeans`([n_clusters, init, ...])	Mini-Batch K-Means clustering
`cluster.MeanShift`([bandwidth, seeds, ...])	Mean shift clustering using a flat kernel.
`cluster.SpectralClustering`([n_clusters, ...])	Apply clustering to a projection to the normalized laplacian.
`cluster.Ward`([n_clusters, memory, ...])	Ward hierarchical clustering: constructs a tree and cuts it.

Functions

cluster.estimate_bandwidth(X[, quantile, ...])	Estimate the bandwidth to use with the mean-shift algorithm.
cluster.k_means(X, n_clusters[, init, ...])	K-means clustering algorithm.
cluster.ward_tree(X[, connectivity, ...])	Ward clustering based on a Feature matrix.
cluster.affinity_propagation(S[, ...])	Perform Affinity Propagation Clustering of data
cluster.dbscan(X[, eps, min_samples, ...])	Perform DBSCAN clustering from vector array or distance matrix.
cluster.mean_shift(X[, bandwidth, seeds, ...])	Perform mean shift clustering of data using a flat kernel.
cluster.spectral_clustering(affinity[, ...])	Apply clustering to a projection to the normalized laplacian.

sklearn.cluster.bicluster: Biclustering

User guide: See the *Biclustering* section for further details.

Classes

SpectralBiclustering([n_clusters, method, ...])	Spectral biclustering (Kluger, 2003).
SpectralCoclustering([n_clusters, ...])	Spectral Co-Clustering algorithm (Dhillon, 2001).

sklearn.covariance: Covariance Estimators

The sklearn.covariance module includes methods and algorithms to robustly estimate the covariance of features given a set of points. The precision matrix defined as the inverse of the covariance is also estimated. Covariance estimation is closely related to the theory of Gaussian Graphical Models.

User guide: See the *Covariance estimation* section for further details.

covariance.EmpiricalCovariance([...])	Maximum likelihood covariance estimator
covariance.EllipticEnvelope([...])	An object for detecting outliers in a Gaussian distributed dataset.
covariance.GraphLasso([alpha, mode, tol, ...])	Sparse inverse covariance estimation with an l1-penalized estimator.
covariance.GraphLassoCV([alphas, ...])	Sparse inverse covariance w/ cross-validated choice of the l1 penalty
covariance.LedoitWolf([store_precision, ...])	LedoitWolf Estimator
covariance.MinCovDet([store_precision, ...])	Minimum Covariance Determinant (MCD): robust estimator of covariance.
covariance.OAS([store_precision, ...])	Oracle Approximating Shrinkage Estimator
covariance.ShrunkCovariance([...])	Covariance estimator with shrinkage
covariance.empirical_covariance(X[, ...])	Computes the Maximum likelihood covariance estimator
covariance.ledoit_wolf(X[, assume_centered, ...])	Estimates the shrunk Ledoit-Wolf covariance matrix.
covariance.shrunk_covariance(emp_cov[, ...])	Calculates a covariance matrix shrunk on the diagonal
covariance.oas(X[, assume_centered])	Estimate covariance with the Oracle Approximating Shrinkage algorithm.
covariance.graph_lasso(emp_cov, alpha[, ...])	l1-penalized covariance estimator

sklearn.cross_validation: Cross Validation

The sklearn.cross_validation module includes utilities for cross- validation and performance evaluation.

User guide: See the *Cross-validation: evaluating estimator performance* section for further details.

cross_validation.KFold(n[, n_folds, ...])	K-Folds cross validation iterator.
cross_validation.LeaveOneLabelOut(labels[, ...])	Leave-One-Label_Out cross-validation iterator
cross_validation.LeaveOneOut(n[, indices])	Leave-One-Out cross validation iterator.
cross_validation.LeavePLabelOut(labels, p[, ...])	Leave-P-Label_Out cross-validation iterator
cross_validation.LeavePOut(n, p[, indices])	Leave-P-Out cross validation iterator
cross_validation.StratifiedKFold(y[, ...])	Stratified K-Folds cross validation iterator
cross_validation.ShuffleSplit(n[, n_iter, ...])	Random permutation cross-validation iterator.
cross_validation.StratifiedShuffleSplit(y[, ...])	Stratified ShuffleSplit cross validation iterator
cross_validation.train_test_split(*arrays, ...)	Split arrays or matrices into random train and test subsets
cross_validation.cross_val_score(estimator, X)	Evaluate a score by cross-validation
cross_validation.permutation_test_score(...)	Evaluate the significance of a cross-validated score with permutations
cross_validation.check_cv(cv[, X, y, classifier])	Input checker utility for building a CV in a user friendly way.

sklearn.datasets: Datasets

The sklearn.datasets module includes utilities to load datasets, including methods to load and fetch popular reference datasets. It also features some artificial data generators.

User guide: See the *Dataset loading utilities* section for further details.

Loaders

datasets.fetch_20newsgroups([data_home, ...])	Load the filenames and data from the 20 newsgroups dataset.
datasets.fetch_20newsgroups_vectorized([...])	Load the 20 newsgroups dataset and transform it into tf-idf vectors.
datasets.load_boston()	Load and return the boston house-prices dataset (regression).
datasets.load_diabetes()	Load and return the diabetes dataset (regression).
datasets.load_digits([n_class])	Load and return the digits dataset (classification).
datasets.load_files(container_path[, ...])	Load text files with categories as subfolder names.
datasets.load_iris()	Load and return the iris dataset (classification).
datasets.load_lfw_pairs([download_if_missing])	Alias for fetch_lfw_pairs(download_if_missing=False)
datasets.fetch_lfw_pairs([subset, ...])	Loader for the Labeled Faces in the Wild (LFW) pairs dataset
datasets.load_lfw_people([download_if_missing])	Alias for fetch_lfw_people(download_if_missing=False)
datasets.fetch_lfw_people([data_home, ...])	Loader for the Labeled Faces in the Wild (LFW) people dataset
datasets.load_linnerud()	Load and return the linnerud dataset (multivariate regression).
datasets.fetch_mldata(dataname[, ...])	Fetch an mldata.org data set

datasets.fetch_olivetti_faces([data_home, ...])	Loader for the Olivetti faces data-set from AT&T.
datasets.fetch_california_housing([...])	Loader for the California housing dataset from StatLib.
datasets.fetch_covtype([data_home, ...])	Load the covertype dataset, downloading it if necessary.
datasets.load_mlcomp(name_or_id[, set_, ...])	Load a datasets as downloaded from http://mlcomp.org
datasets.load_sample_image(image_name)	Load the numpy array of a single sample image
datasets.load_sample_images()	Load sample images for image manipulation.
datasets.load_svmlight_file(f[, n_features, ...])	Load datasets in the svmlight / libsvm format into sparse CSR matrix
datasets.dump_svmlight_file(X, y, f[, ...])	Dump the dataset in svmlight / libsvm file format.

Samples generator

datasets.make_blobs([n_samples, n_features, ...])	Generate isotropic Gaussian blobs for clustering.
datasets.make_classification([n_samples, ...])	Generate a random n-class classification problem.
datasets.make_circles([n_samples, shuffle, ...])	Make a large circle containing a smaller circle in 2d.
datasets.make_friedman1([n_samples, ...])	Generate the “Friedman #1” regression problem
datasets.make_friedman2([n_samples, noise, ...])	Generate the “Friedman #2” regression problem
datasets.make_friedman3([n_samples, noise, ...])	Generate the “Friedman #3” regression problem
datasets.make_gaussian_quantiles([mean, ...])	Generate isotropic Gaussian and label samples by quantile
datasets.make_hastie_10_2([n_samples, ...])	Generates data for binary classification used in Hastie et al.
datasets.make_low_rank_matrix([n_samples, ...])	Generate a mostly low rank matrix with bell-shaped singular values
datasets.make_moons([n_samples, shuffle, ...])	Make two interleaving half circles
datasets.make_multilabel_classification([...])	Generate a random multilabel classification problem.
datasets.make_regression([n_samples, ...])	Generate a random regression problem.
datasets.make_s_curve([n_samples, noise, ...])	Generate an S curve dataset.
datasets.make_sparse_coded_signal(n_samples, ...)	Generate a signal as a sparse combination of dictionary elements.
datasets.make_sparse_spd_matrix([dim, ...])	Generate a sparse symmetric definite positive matrix.
datasets.make_sparse_uncorrelated([...])	Generate a random regression problem with sparse uncorrelated design
datasets.make_spd_matrix(n_dim[, random_state])	Generate a random symmetric, positive-definite matrix.
datasets.make_swiss_roll([n_samples, noise, ...])	Generate a swiss roll dataset.
datasets.make_biclusters(shape, n_clusters)	Generate an array with constant block diagonal structure for biclustering.
datasets.make_checkerboard(shape, n_clusters)	Generate an array with block checkerboard structure for biclustering.

sklearn.decomposition: Matrix Decomposition

The sklearn.decomposition module includes matrix decomposition algorithms, including among others

PCA, NMF or ICA. Most of the algorithms of this module can be regarded as dimensionality reduction techniques.

User guide: See the *Decomposing signals in components (matrix factorization problems)* section for further details.

`decomposition.PCA`([n_components, copy, whiten])	Principal component analysis (PCA)
`decomposition.ProjectedGradientNMF`([...])	Non-Negative matrix factorization by Projected Gradient (NMF)
`decomposition.RandomizedPCA`([n_components, ...])	Principal component analysis (PCA) using randomized SVD
`decomposition.KernelPCA`([n_components, ...])	Kernel Principal component analysis (KPCA)
`decomposition.FactorAnalysis`([n_components, ...])	Factor Analysis (FA)
`decomposition.FastICA`([n_components, ...])	FastICA: a fast algorithm for Independent Component Analysis.
`decomposition.TruncatedSVD`([n_components, ...])	Dimensionality reduction using truncated SVD (aka LSA).
`decomposition.NMF`([n_components, init, ...])	Non-Negative matrix factorization by Projected Gradient (NMF)
`decomposition.SparsePCA`([n_components, ...])	Sparse Principal Components Analysis (SparsePCA)
`decomposition.MiniBatchSparsePCA`([...])	Mini-batch Sparse Principal Components Analysis
`decomposition.SparseCoder`(dictionary[, ...])	Sparse coding
`decomposition.DictionaryLearning`([...])	Dictionary learning
`decomposition.MiniBatchDictionaryLearning`([...])	Mini-batch dictionary learning
`decomposition.fastica`(X[, n_components, ...])	Perform Fast Independent Component Analysis.
`decomposition.dict_learning`(X, n_components, ...)	Solves a dictionary learning matrix factorization problem.
`decomposition.dict_learning_online`(X[, ...])	Solves a dictionary learning matrix factorization problem online.
`decomposition.sparse_encode`(X, dictionary[, ...])	Sparse coding

`sklearn.dummy`: Dummy estimators

User guide: See the *Model evaluation: quantifying the quality of predictions* section for further details.

`dummy.DummyClassifier`([strategy, ...])	DummyClassifier is a classifier that makes predictions using simple rules.
`dummy.DummyRegressor`([strategy, constant])	DummyRegressor is a regressor that makes predictions using simple rules.

`sklearn.ensemble`: Ensemble Methods

The `sklearn.ensemble` module includes ensemble-based methods for classification and regression.

User guide: See the *Ensemble methods* section for further details.

`ensemble.AdaBoostClassifier`([...])	An AdaBoost classifier.
`ensemble.AdaBoostRegressor`([base_estimator, ...])	An AdaBoost regressor.
`ensemble.BaggingClassifier`([base_estimator, ...])	A Bagging classifier.
`ensemble.BaggingRegressor`([base_estimator, ...])	A Bagging regressor.
`ensemble.ExtraTreesClassifier`([...])	An extra-trees classifier.
`ensemble.ExtraTreesRegressor`([n_estimators, ...])	An extra-trees regressor.
`ensemble.GradientBoostingClassifier`([loss, ...])	Gradient Boosting for classification.

ensemble.GradientBoostingRegressor([loss, ...])	Gradient Boosting for regression.
ensemble.RandomForestClassifier([...])	A random forest classifier.
ensemble.RandomTreesEmbedding([...])	An ensemble of totally random trees.
ensemble.RandomForestRegressor([...])	A random forest regressor.

partial dependence

Partial dependence plots for tree ensembles.

ensemble.partial_dependence.partial_dependence(...)	Partial dependence of `target_variables`.
ensemble.partial_dependence.plot_partial_dependence(...)	Partial dependence plots for `features`.

sklearn.feature_extraction: Feature Extraction

The sklearn.feature_extraction module deals with feature extraction from raw data. It currently includes methods to extract features from text and images.

User guide: See the *Feature extraction* section for further details.

feature_extraction.DictVectorizer([dtype, ...])	Transforms lists of feature-value mappings to vectors.
feature_extraction.FeatureHasher([...])	Implements feature hashing, aka the hashing trick.

From images

The sklearn.feature_extraction.image submodule gathers utilities to extract features from images.

feature_extraction.image.img_to_graph(img[, ...])	Graph of the pixel-to-pixel gradient connections
feature_extraction.image.grid_to_graph(n_x, n_y)	Graph of the pixel-to-pixel connections
feature_extraction.image.extract_patches_2d(...)	Reshape a 2D image into a collection of patches
feature_extraction.image.reconstruct_from_patches_2d(...)	Reconstruct the image from all of its patches.
feature_extraction.image.PatchExtractor([...])	Extracts patches from a collection of images

From text

The sklearn.feature_extraction.text submodule gathers utilities to build feature vectors from text documents.

feature_extraction.text.CountVectorizer([...])	Convert a collection of text documents to a matrix of token counts
feature_extraction.text.HashingVectorizer([...])	Convert a collection of text documents to a matrix of token occurrences
feature_extraction.text.TfidfTransformer([...])	Transform a count matrix to a normalized tf or tf-idf representation
feature_extraction.text.TfidfVectorizer([...])	Convert a collection of raw documents to a matrix of TF-IDF features.

sklearn.feature_selection: Feature Selection

The `sklearn.feature_selection` module implements feature selection algorithms. It currently includes univariate filter selection methods and the recursive feature elimination algorithm.

User guide: See the *Feature selection* section for further details.

`feature_selection.GenericUnivariateSelect`([...])	Univariate feature selector with configurable strategy.
`feature_selection.SelectPercentile`([...])	Select features according to a percentile of the highest scores.
`feature_selection.SelectKBest`([score_func, k])	Select features according to the k highest scores.
`feature_selection.SelectFpr`([score_func, alpha])	Filter: Select the pvalues below alpha based on a FPR test.
`feature_selection.SelectFdr`([score_func, alpha])	Filter: Select the p-values for an estimated false discovery rate
`feature_selection.SelectFwe`([score_func, alpha])	Filter: Select the p-values corresponding to Family-wise error rate
`feature_selection.RFE`(estimator[, ...])	Feature ranking with recursive feature elimination.
`feature_selection.RFECV`(estimator[, step, ...])	Feature ranking with recursive feature elimination and cross-validated selection of the best number of features.
`feature_selection.VarianceThreshold`([threshold])	Feature selector that removes all low-variance features.
`feature_selection.chi2`(X, y)	Compute chi-squared statistic for each class/feature combination.
`feature_selection.f_classif`(X, y)	Compute the Anova F-value for the provided sample
`feature_selection.f_regression`(X, y[, center])	Univariate linear regression tests

`sklearn.gaussian_process`: Gaussian Processes

The `sklearn.gaussian_process` module implements scalar Gaussian Process based predictions.

User guide: See the *Gaussian Processes* section for further details.

`gaussian_process.GaussianProcess`([regr, ...])	The Gaussian Process model class.
`gaussian_process.correlation_models.absolute_exponential`(...)	Absolute exponential autocorrelation model.
`gaussian_process.correlation_models.squared_exponential`(...)	Squared exponential correlation model (Radial Basis Function).
`gaussian_process.correlation_models.generalized_exponential`(...)	Generalized exponential correlation model.
`gaussian_process.correlation_models.pure_nugget`(...)	Spatial independence correlation model (pure nugget).
`gaussian_process.correlation_models.cubic`(...)	Cubic correlation model:
`gaussian_process.correlation_models.linear`(...)	Linear correlation model:
`gaussian_process.regression_models.constant`(x)	Zero order polynomial (constant, p = 1) regression model.
`gaussian_process.regression_models.linear`(x)	First order polynomial (linear, p = n+1) regression model.
`gaussian_process.regression_models.quadratic`(x)	Second order polynomial (quadratic, p = n*(n-

	1)/2+n+1) regression model.

sklearn.grid_search: Grid Search

The sklearn.grid_search includes utilities to fine-tune the parameters of an estimator.

User guide: See the *Grid Search: Searching for estimator parameters* section for further details.

grid_search.GridSearchCV(estimator, param_grid)	Exhaustive search over specified parameter values for an estimator.
grid_search.ParameterGrid(param_grid)	Grid of parameters with a discrete number of values for each.
grid_search.ParameterSampler(... [, random_state])	Generator on parameters sampled from given distributions.
grid_search.RandomizedSearchCV(estimator, ...)	Randomized search on hyper parameters.

sklearn.isotonic: Isotonic regression

User guide: See the *Isotonic regression* section for further details.

isotonic.IsotonicRegression([y_min, y_max, ...])	Isotonic regression model.
isotonic.isotonic_regression(y[, ...])	Solve the isotonic regression model:
isotonic.check_increasing(x, y)	Determine whether y is monotonically correlated with x.

sklearn.kernel_approximation Kernel Approximation

The sklearn.kernel_approximation module implements several approximate kernel feature maps base on Fourier transforms.

User guide: See the *Kernel Approximation* section for further details.

kernel_approximation.AdditiveChi2Sampler([...])	Approximate feature map for additive chi2 kernel.
kernel_approximation.Nystroem([kernel, ...])	Approximate a kernel map using a subset of the training data.
kernel_approximation.RBFSampler([gamma, ...])	Approximates feature map of an RBF kernel by Monte Carlo approximation of its Fourier transform.
kernel_approximation.SkewedChi2Sampler([...])	Approximates feature map of the “skewed chi-squared” kernel by Monte Carlo approximation of its Fourier transform.

sklearn.lda: Linear Discriminant Analysis

The sklearn.lda module implements Linear Discriminant Analysis (LDA).

User guide: See the *Linear and quadratic discriminant analysis* section for further details.

lda.LDA([n_components, priors])	Linear Discriminant Analysis (LDA)

sklearn.learning_curve Learning curve evaluation

Utilities to evaluate models with respect to a variable

learning_curve.learning_curve(estimator, X, y)	Learning curve.
learning_curve.validation_curve(estimator, ...)	Validation curve.

sklearn.linear_model: Generalized Linear Models

The sklearn.linear_model module implements generalized linear models. It includes Ridge regression, Bayesian Regression, Lasso and Elastic Net estimators computed with Least Angle Regression and coordinate descent. It also implements Stochastic Gradient Descent related algorithms.

User guide: See the *Generalized Linear Models* section for further details.

linear_model.ARDRegression([n_iter, tol, ...])	Bayesian ARD regression.
linear_model.BayesianRidge([n_iter, tol, ...])	Bayesian ridge regression
linear_model.ElasticNet([alpha, l1_ratio, ...])	Linear regression with combined L1 and L2 priors as regularizer.
linear_model.ElasticNetCV([l1_ratio, eps, ...])	Elastic Net model with iterative fitting along a regularization path
linear_model.Lars([fit_intercept, verbose, ...])	Least Angle Regression model a.k.a.
linear_model.LarsCV([fit_intercept, ...])	Cross-validated Least Angle Regression model
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.LassoLars([alpha, ...])	Lasso model fit with Least Angle Regression a.k.a.
linear_model.LassoLarsCV([fit_intercept, ...])	Cross-validated Lasso, using the LARS algorithm
linear_model.LassoLarsIC([criterion, ...])	Lasso model fit with Lars using BIC or AIC for model selection
linear_model.LinearRegression([...])	Ordinary least squares Linear Regression.
linear_model.LogisticRegression([penalty, ...])	Logistic Regression (aka logit, MaxEnt) classifier.
linear_model.MultiTaskLasso([alpha, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer
linear_model.MultiTaskElasticNet([alpha, ...])	Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer
linear_model.MultiTaskLassoCV([eps, ...])	Multi-task L1/L2 Lasso with built-in cross-validation.
linear_model.MultiTaskElasticNetCV([...])	Multi-task L1/L2 ElasticNet with built-in cross-validation.
linear_model.OrthogonalMatchingPursuit([...])	Orthogonal Mathching Pursuit model (OMP)
linear_model.OrthogonalMatchingPursuitCV([...])	Cross-validated Orthogonal Mathching Pursuit model (OMP)
linear_model.PassiveAggressiveClassifier([...])	Passive Aggressive Classifier
linear_model.PassiveAggressiveRegressor([C, ...])	Passive Aggressive Regressor
linear_model.Perceptron([penalty, alpha, ...])	Perceptron
linear_model.RandomizedLasso([alpha, ...])	Randomized Lasso.
linear_model.RandomizedLogisticRegression([...])	Randomized Logistic Regression
linear_model.RANSACRegressor([...])	RANSAC (RANdom SAmple Consensus) algorithm.
linear_model.Ridge([alpha, fit_intercept, ...])	Linear least squares with l2 regularization.
linear_model.RidgeClassifier([alpha, ...])	Classifier using Ridge regression.
linear_model.RidgeClassifierCV([alphas, ...])	Ridge classifier with built-in cross-validation.
linear_model.RidgeCV([alphas, ...])	Ridge regression with built-in cross-validation.

linear_model.SGDClassifier([loss, penalty, ...])	Linear classifiers (SVM, logistic regression, a.o.) with SGD training.
linear_model.SGDRegressor([loss, penalty, ...])	Linear model fitted by minimizing a regularized empirical loss with SGD
linear_model.lars_path(X, y[, Xy, Gram, ...])	Compute Least Angle Regression or Lasso path using LARS algorithm [1]
linear_model.lasso_path(X, y[, eps, ...])	Compute Lasso path with coordinate descent
linear_model.lasso_stability_path(X, y[, ...])	Stabiliy path based on randomized Lasso estimates
linear_model.orthogonal_mp(X, y[, ...])	Orthogonal Matching Pursuit (OMP)
linear_model.orthogonal_mp_gram(Gram, Xy[, ...])	Gram Orthogonal Matching Pursuit (OMP)

sklearn.manifold: Manifold Learning

The sklearn.manifold module implements data embedding techniques.

User guide: See the *Manifold learning* section for further details.

manifold.LocallyLinearEmbedding([...])	Locally Linear Embedding
manifold.Isomap([n_neighbors, n_components, ...])	Isomap Embedding
manifold.MDS([n_components, metric, n_init, ...])	Multidimensional scaling
manifold.SpectralEmbedding([n_components, ...])	Spectral embedding for non-linear dimensionality reduction.
manifold.TSNE([n_components, perplexity, ...])	t-distributed Stochastic Neighbor Embedding.
manifold.locally_linear_embedding(X, ...[, ...])	Perform a Locally Linear Embedding analysis on the data.
manifold.spectral_embedding(adjacency[, ...])	Project the sample on the first eigen vectors of the graph Laplacian.

sklearn.metrics: Metrics

See the *Model evaluation: quantifying the quality of predictions* section and the *Pairwise metrics, Affinities and Kernels* section of the user guide for further details.

The sklearn.metrics module includes score functions, performance metrics and pairwise metrics and distance computations.

Model Selection Interface

See the *The scoring parameter: defining model evaluation rules* section of the user guide for further details.

metrics.make_scorer(score_func[, ...])	Make a scorer from a performance metric or loss function.

Classification metrics

See the *Classification metrics* section of the user guide for further details.

metrics.accuracy_score(y_true, y_pred[, ...])	Accuracy classification score.
metrics.auc(x, y[, reorder])	Compute Area Under the Curve (AUC) using the trapezoidal rule
metrics.average_precision_score(y_true, y_score)	Compute average precision (AP) from prediction scores
metrics.classification_report(y_true, y_pred)	Build a text report showing the main

	classification metrics
metrics.confusion_matrix(y_true, y_pred[, ...])	Compute confusion matrix to evaluate the accuracy of a classification
metrics.f1_score(y_true, y_pred[, labels, ...])	Compute the F1 score, also known as balanced F-score or F-measure
metrics.fbeta_score(y_true, y_pred, beta[, ...])	Compute the F-beta score
metrics.hamming_loss(y_true, y_pred[, classes])	Compute the average Hamming loss.
metrics.hinge_loss(y_true, pred_decision[, ...])	Average hinge loss (non-regularized)
metrics.jaccard_similarity_score(y_true, y_pred)	Jaccard similarity coefficient score
metrics.log_loss(y_true, y_pred[, eps, ...])	Log loss, aka logistic loss or cross-entropy loss.
metrics.matthews_corrcoef(y_true, y_pred)	Compute the Matthews correlation coefficient (MCC) for binary classes
metrics.precision_recall_curve(y_true, ...)	Compute precision-recall pairs for different probability thresholds
metrics.precision_recall_fscore_support(...)	Compute precision, recall, F-measure and support for each class
metrics.precision_score(y_true, y_pred[, ...])	Compute the precision
metrics.recall_score(y_true, y_pred[, ...])	Compute the recall
metrics.roc_auc_score(y_true, y_score[, ...])	Compute Area Under the Curve (AUC) from prediction scores
metrics.roc_curve(y_true, y_score[, ...])	Compute Receiver operating characteristic (ROC)
metrics.zero_one_loss(y_true, y_pred[, ...])	Zero-one classification loss.

Regression metrics

See the *Regression metrics* section of the user guide for further details.

metrics.explained_variance_score(y_true, y_pred)	Explained variance regression score function
metrics.mean_absolute_error(y_true, y_pred)	Mean absolute error regression loss
metrics.mean_squared_error(y_true, y_pred[, ...])	Mean squared error regression loss
metrics.r2_score(y_true, y_pred[, sample_weight])	R^2 (coefficient of determination) regression score function.

Clustering metrics

See the *Clustering performance evaluation* section of the user guide for further details.

The sklearn.metrics.cluster submodule contains evaluation metrics for cluster analysis results. There are two forms of evaluation:

- supervised, which uses a ground truth class values for each sample.
- unsupervised, which does not and measures the 'quality' of the model itself.

metrics.adjusted_mutual_info_score(...)	Adjusted Mutual Information between two clusterings
metrics.adjusted_rand_score(labels_true, ...)	Rand index adjusted for chance
metrics.completeness_score(labels_true, ...)	Completeness metric of a cluster labeling given a ground truth
metrics.homogeneity_completeness_v_measure(...)	Compute the homogeneity and completeness and V-Measure scores at once
metrics.homogeneity_score(labels_true, ...)	Homogeneity metric of a cluster labeling given a ground truth
metrics.mutual_info_score(labels_true, ...)	Mutual Information between two clusterings

metrics.normalized_mutual_info_score(...)	Normalized Mutual Information between two clusterings
metrics.silhouette_score(X, labels[, ...])	Compute the mean Silhouette Coefficient of all samples.
metrics.silhouette_samples(X, labels[, metric])	Compute the Silhouette Coefficient for each sample.
metrics.v_measure_score(labels_true, labels_pred)	V-measure cluster labeling given a ground truth.

Biclustering metrics

See the *Biclustering evaluation* section of the user guide for further details.

metrics.consensus_score(a, b[, similarity])	The similarity of two sets of biclusters.

Pairwise metrics

The sklearn.metrics.pairwise submodule implements utilities to evaluate pairwise distances, paired distances or affinity of sets of samples.

This module contains both distance metrics and kernels. A brief summary is given on the two here.

Distance metrics are a function d(a, b) such that d(a, b) < d(a, c) if objects a and b are considered “more similar” to objects a and c. Two objects exactly alike would have a distance of zero. One of the most popular examples is Euclidean distance. To be a ‘true’ metric, it must obey the following four conditions:

```
1. d(a, b) >= 0, for all a and b
2. d(a, b) == 0, if and only if a = b, positive definiteness
3. d(a, b) == d(b, a), symmetry
4. d(a, c) <= d(a, b) + d(b, c), the triangle inequality
```

Kernels are measures of similarity, i.e. `s(a, b) > s(a, c)` if objects `a` and `b` are considered “more similar” to objects `a` and `c`. A kernel must also be positive semi-definite.

There are a number of ways to convert between a distance metric and a similarity measure, such as a kernel. Let D be the distance, and S be the kernel:

1. `S = np.exp(-D * gamma)`, where one heuristic for choosing `gamma` is `1 / num_features`
2. `S = 1. / (D / np.max(D))`

metrics.pairwise.additive_chi2_kernel(X[, Y])	Computes the additive chi-squared kernel between observations in X and Y
metrics.pairwise.chi2_kernel(X[, Y, gamma])	Computes the exponential chi-squared kernel X and Y.
metrics.pairwise.distance_metrics()	Valid metrics for pairwise_distances.
metrics.pairwise.euclidean_distances(X[, Y, ...])	Considering the rows of X (and Y=X) as vectors, compute the distance matrix between each pair of vectors.
metrics.pairwise.kernel_metrics()	Valid metrics for pairwise_kernels
metrics.pairwise.linear_kernel(X[, Y])	Compute the linear kernel between X and Y.
metrics.pairwise.manhattan_distances(X[, Y, ...])	Compute the L1 distances between the vectors in X and Y.
metrics.pairwise.pairwise_distances(X[, Y, ...])	Compute the distance matrix from a vector array X and optional Y.
metrics.pairwise.pairwise_kernels(X[, Y, ...])	Compute the kernel between arrays X and optional array Y.

metrics.pairwise.polynomial_kernel(X[, Y, ...])	Compute the polynomial kernel between X and Y:
metrics.pairwise.rbf_kernel(X[, Y, gamma])	Compute the rbf (gaussian) kernel between X and Y:
metrics.pairwise_distances(X[, Y, metric, ...])	Compute the distance matrix from a vector array X and optional Y.
metrics.pairwise_distances_argmin(X, Y[, ...])	Compute minimum distances between one point and a set of points.
metrics.pairwise_distances_argmin_min(X, Y)	Compute minimum distances between one point and a set of points.

sklearn.mixture: Gaussian Mixture Models

The sklearn.mixture module implements mixture modeling algorithms.

User guide: See the *Gaussian mixture models* section for further details.

mixture.GMM([n_components, covariance_type, ...])	Gaussian Mixture Model
mixture.DPGMM([n_components, ...])	Variational Inference for the Infinite Gaussian Mixture Model.
mixture.VBGMM([n_components, ...])	Variational Inference for the Gaussian Mixture Model

sklearn.multiclass: Multiclass and multilabel classification

Multiclass and multilabel classification strategies

This module implements multiclass learning algorithms:

- one-vs-the-rest / one-vs-all
- one-vs-one
- error correcting output codes

The estimators provided in this module are meta-estimators: they require a base estimator to be provided in their constructor. For example, it is possible to use these estimators to turn a binary classifier or a regressor into a multiclass classifier. It is also possible to use these estimators with multiclass estimators in the hope that their accuracy or runtime performance improves.

All classifiers in scikit-learn implement multiclass classification; you only need to use this module if you want to experiment with custom multiclass strategies.

The one-vs-the-rest meta-classifier also implements a *predict_proba* method, so long as such a method is implemented by the base classifier. This method returns probabilities of class membership in both the single label and multilabel case. Note that in the multilabel case, probabilities are the marginal probability that a given sample falls in the given class. As such, in the multilabel case the sum of these probabilities over all possible labels for a given sample *will not* sum to unity, as they do in the single label case.

User guide: See the *Multiclass and multilabel algorithms* section for further details.

multiclass.OneVsRestClassifier(estimator[, ...])	One-vs-the-rest (OvR) multiclass/multilabel strategy
multiclass.OneVsOneClassifier(estimator[, ...])	One-vs-one multiclass strategy
multiclass.OutputCodeClassifier(estimator[, ...])	(Error-Correcting) Output-Code multiclass strategy

multiclass.fit_ovr(estimator, X, y[, n_jobs])	Fit a one-vs-the-rest strategy.
multiclass.predict_ovr(estimators, ...)	Make predictions using the one-vs-the-rest strategy.
multiclass.fit_ovo(estimator, X, y[, n_jobs])	Fit a one-vs-one strategy.
multiclass.predict_ovo(estimators, classes, X)	Make predictions using the one-vs-one strategy.
multiclass.fit_ecoc(estimator, X, y[, ...])	Fit an error-correcting output-code strategy.
multiclass.predict_ecoc(estimators, classes, ...)	Make predictions using the error-correcting output-code strategy.

sklearn.naive_bayes: Naive Bayes

The sklearn.naive_bayes module implements Naive Bayes algorithms. These are supervised learning methods based on applying Bayes' theorem with strong (naive) feature independence assumptions.

User guide: See the *Naive Bayes* section for further details.

naive_bayes.GaussianNB	Gaussian Naive Bayes (GaussianNB)
naive_bayes.MultinomialNB([alpha, ...])	Naive Bayes classifier for multinomial models
naive_bayes.BernoulliNB([alpha, binarize, ...])	Naive Bayes classifier for multivariate Bernoulli models.

sklearn.neighbors: Nearest Neighbors

The sklearn.neighbors module implements the k-nearest neighbors algorithm.

User guide: See the *Nearest Neighbors* section for further details.

neighbors.NearestNeighbors([n_neighbors, ...])	Unsupervised learner for implementing neighbor searches.
neighbors.KNeighborsClassifier([...])	Classifier implementing the k-nearest neighbors vote.
neighbors.RadiusNeighborsClassifier([...])	Classifier implementing a vote among neighbors within a given radius
neighbors.KNeighborsRegressor([n_neighbors, ...])	Regression based on k-nearest neighbors.
neighbors.RadiusNeighborsRegressor([radius, ...])	Regression based on neighbors within a fixed radius.
neighbors.NearestCentroid([metric, ...])	Nearest centroid classifier.
neighbors.BallTree	BallTree for fast generalized N-point problems
neighbors.KDTree	KDTree for fast generalized N-point problems
neighbors.DistanceMetric	DistanceMetric class
neighbors.KernelDensity([bandwidth, ...])	Kernel Density Estimation
neighbors.kneighbors_graph(X, n_neighbors[, ...])	Computes the (weighted) graph of k-Neighbors for points in X
neighbors.radius_neighbors_graph(X, radius)	Computes the (weighted) graph of Neighbors for points in X

sklearn.neural_network: Neural network models

The sklearn.neural_network module includes models based on neural networks.

User guide: See the *Neural network models (unsupervised)* section for further details.

neural_network.BernoulliRBM([n_components, ...])	Bernoulli Restricted Boltzmann Machine (RBM).

sklearn.cross_decomposition: Cross decomposition

User guide: See the *Cross decomposition* section for further details.

cross_decomposition.PLSRegression([...])	PLS regression
cross_decomposition.PLSCanonical([...])	PLSCanonical implements the 2 blocks canonical PLS of the original Wold algorithm [Tenenhaus 1998] p.204, referred as PLS-C2A in [Wegelin 2000].
cross_decomposition.CCA([n_components, ...])	CCA Canonical Correlation Analysis.
cross_decomposition.PLSSVD([n_components, ...])	Partial Least Square SVD

sklearn.pipeline: Pipeline

The sklearn.pipeline module implements utilities to build a composite estimator, as a chain of transforms and estimators.

pipeline.Pipeline(steps)	Pipeline of transforms with a final estimator.
pipeline.FeatureUnion(transformer_list[, ...])	Concatenates results of multiple transformer objects.
pipeline.make_pipeline(*steps)	Construct a Pipeline from the given estimators.
pipeline.make_union(*transformers)	Construct a FeatureUnion from the given transformers.

sklearn.preprocessing: Preprocessing and Normalization

The sklearn.preprocessing module includes scaling, centering, normalization, binarization and imputation methods.

User guide: See the *Preprocessing data* section for further details.

preprocessing.Binarizer([threshold, copy])	Binarize data (set feature values to 0 or 1) according to a threshold
preprocessing.Imputer([missing_values, ...])	Imputation transformer for completing missing values.
preprocessing.KernelCenterer	Center a kernel matrix
preprocessing.LabelBinarizer([neg_label, ...])	Binarize labels in a one-vs-all fashion
preprocessing.LabelEncoder	Encode labels with value between 0 and n_classes-1.
preprocessing.MultiLabelBinarizer([classes, ...])	Transform between iterable of iterables and a multilabel format
preprocessing.MinMaxScaler([feature_range, copy])	Standardizes features by scaling each feature to a given range.
preprocessing.Normalizer([norm, copy])	Normalize samples individually to unit norm
preprocessing.OneHotEncoder([n_values, ...])	Encode categorical integer features using a one-hot aka one-of-K scheme.
preprocessing.StandardScaler([copy, ...])	Standardize features by removing the mean and scaling to unit variance
preprocessing.PolynomialFeatures([degree, ...])	Generate polynomial and interaction features.
preprocessing.add_dummy_feature(X[, value])	Augment dataset with an additional dummy feature.
preprocessing.binarize(X[, threshold, copy])	Boolean thresholding of array-like or scipy.sparse matrix
preprocessing.label_binarize(y, classes[, ...])	Binarize labels in a one-vs-all fashion
preprocessing.normalize(X[, norm, axis, copy])	Normalize a dataset along any axis
preprocessing.scale(X[, axis, with_mean, ...])	Standardize a dataset along any axis

sklearn.qda: Quadratic Discriminant Analysis

Quadratic Discriminant Analysis

User guide: See the *Linear and quadratic discriminant analysis* section for further details.

qda.QDA([priors, reg_param])	Quadratic Discriminant Analysis (QDA)

sklearn.random_projection: Random projection

Random Projection transformers

Random Projections are a simple and computationally efficient way to reduce the dimensionality of the data by trading a controlled amount of accuracy (as additional variance) for faster processing times and smaller model sizes.

The dimensions and distribution of Random Projections matrices are controlled so as to preserve the pairwise distances between any two samples of the dataset.

The main theoretical result behind the efficiency of random projection is the Johnson-Lindenstrauss lemma (quoting Wikipedia):

> In mathematics, the Johnson-Lindenstrauss lemma is a result concerning low-distortion embeddings of points from high-dimensional into low-dimensional Euclidean space. The lemma states that a small set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved. The map used for the embedding is at least Lipschitz, and can even be taken to be an orthogonal projection.

User guide: See the *Random Projection* section for further details.

random_projection.GaussianRandomProjection([...])	Reduce dimensionality through Gaussian random projection
random_projection.SparseRandomProjection([...])	Reduce dimensionality through sparse random projection
random_projection.johnson_lindenstrauss_min_dim(...)	Find a ‘safe’ number of components to randomly project to

sklearn.semi_supervised Semi-Supervised Learning

The sklearn.semi_supervised module implements semi-supervised learning algorithms. These algorithms utilized small amounts of labeled data and large amounts of unlabeled data for classification tasks. This module includes Label Propagation.

User guide: See the *Semi-Supervised* section for further details.

semi_supervised.LabelPropagation([kernel, ...])	Label Propagation classifier
semi_supervised.LabelSpreading([kernel, ...])	LabelSpreading model for semi-supervised learning

sklearn.svm: Support Vector Machines

The sklearn.svm module includes Support Vector Machine algorithms.

User guide: See the *Support Vector Machines* section for further details.

Estimators

`svm.SVC`([C, kernel, degree, gamma, coef0, ...])	C-Support Vector Classification.
`svm.LinearSVC`([penalty, loss, dual, tol, C, ...])	Linear Support Vector Classification.
`svm.NuSVC`([nu, kernel, degree, gamma, ...])	Nu-Support Vector Classification.
`svm.SVR`([kernel, degree, gamma, coef0, tol, ...])	epsilon-Support Vector Regression.
`svm.NuSVR`([nu, C, kernel, degree, gamma, ...])	Nu Support Vector Regression.
`svm.OneClassSVM`([kernel, degree, gamma, ...])	Unsupervised Outliers Detection.
`svm.l1_min_c`(X, y[, loss, fit_intercept, ...])	Return the lowest bound for C such that for C in (l1_min_C, infinity) the model is guaranteed not to be empty.

Low-level methods

`svm.libsvm.fit`	Train the model using libsvm (low-level method)
`svm.libsvm.decision_function`	Predict margin (libsvm name for this is predict_values)
`svm.libsvm.predict`	Predict target values of X given a model (low-level method)
`svm.libsvm.predict_proba`	Predict probabilities
`svm.libsvm.cross_validation`	Binding of the cross-validation routine (low-level routine)

`sklearn.tree`: Decision Trees

The `sklearn.tree` module includes decision tree-based models for classification and regression.

User guide: See the *Decision Trees* section for further details.

`tree.DecisionTreeClassifier`([criterion, ...])	A decision tree classifier.
`tree.DecisionTreeRegressor`([criterion, ...])	A decision tree regressor.
`tree.ExtraTreeClassifier`([criterion, ...])	An extremely randomized tree classifier.
`tree.ExtraTreeRegressor`([criterion, ...])	An extremely randomized tree regressor.
`tree.export_graphviz`(decision_tree[, ...])	Export a decision tree in DOT format.

`sklearn.utils`: Utilities

The `sklearn.utils` module includes various utilities.

Developer guide: See the *Utilities for Developers* page for further details.

`utils.check_random_state`(seed)	Turn seed into a np.random.RandomState instance
`utils.resample`(*arrays, **options)	Resample arrays or sparse matrices in a consistent way
`utils.shuffle`(*arrays, **options)	Shuffle arrays or sparse matrices in a consistent way

scikit-learn Tutorials

An introduction to machine learning with scikit-learn

- Machine learning: the problem setting
- Loading an example dataset
- Learning and predicting
- Model persistence

A tutorial on statistical-learning for scientific data processing

- Statistical learning: the setting and the estimator object in scikit-learn
- Supervised learning: predicting an output variable from high-dimensional observations
- Model selection: choosing estimators and their parameters
- Unsupervised learning: seeking representations of the data
- Putting it all together
- Finding help

Working With Text Data

- Tutorial setup
- Loading the 20 newgroups dataset
- Extracting features from text files
- Training a classifier
- Building a pipeline
- Evaluation of the performance on the test set
- Parameter tuning using grid search
- Exercise 1: Language identification
- Exercise 2: Sentiment Analysis on movie reviews
- Exercise 3: CLI text classification utility
- Where to from here

Note: Doctest Mode

The code-examples in the above tutorials are written in a *python-console* format. If you wish to easily execute these examples in **IPython**, use:

```
%doctest_mode
```

in the IPython-console. You can then simply copy and paste the examples directly into IPython without having to worry about removing the **>>>** manually.

An introduction to machine learning with scikit-learn

Section contents

In this section, we introduce the machine learning vocabulary that we use throughout scikit-learn and give a simple learning example.

Machine learning: the problem setting

In general, a learning problem considers a set of n samples of data and then tries 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 (*Click here* to go to the scikit-learn supervised learning page).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 handwritten digit recognition example, in which the aim is to assign each input vector to one of a finite number of discrete categories. Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories and for each of the n samples provided, one is to try to label them with the correct category or class.
    - 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 three dimensions for the purpose of *visualization* (*Click here* to go to the Scikit-Learn unsupervised learning page).

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 into two sets, one that we call the **training set** on which we learn data properties and one that we call the **testing set** on which we test these properties.

Loading an example dataset

scikit-learn comes with a few standard datasets, for instance the iris and digits datasets for classification and the boston house prices dataset for regression.

In the following, we start a Python interpreter from our shell and then load the `iris` and `digits` datasets. Our notational convention is that `$` denotes the shell prompt while `>>>` denotes the Python interpreter prompt:

```
$ python
>>> from sklearn 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, one or more response variables are stored in the `.target` member. More details on the different datasets can be found in the *dedicated section*.

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, shape `(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 scikit-learn.

Learning and predicting

In the case of the digits dataset, the task is to predict, given an image, which digit it represents. We are given samples of each of the 10 possible classes (the digits zero through nine) on which we *fit* an estimator to be able to *predict* the classes to which unseen samples belong.

In scikit-learn, an estimator for classification is a Python object that implements the methods `fit(X, y)` and `predict(T)`.

An example of an estimator is the class `sklearn.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:

```
>>> from sklearn import svm
>>> clf = svm.SVC(gamma=0.001, C=100.)
```

Choosing the parameters of the model

In this example we set the value of `gamma` manually. It is possible to automatically find good values for the parameters by using tools such as *grid search* and *cross validation*.

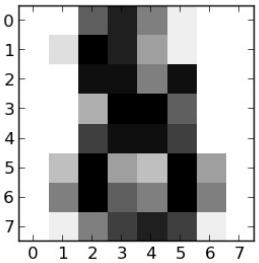
We call our estimator instance `clf`, as it is a classifier. It now must be fitted to the model, that is, it must *learn* from the model. This is done by passing our training set to the `fit` method. As a training set, let us use all the images of our dataset apart from the last one. We select this training set with the `[:-1]` Python syntax, which produces a new array that contains all but the last entry of `digits.data`:

```
>>> clf.fit(digits.data[:-1], digits.target[:-1])
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
  random_state=None, shrinking=True, tol=0.001, verbose=False)
```

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*.

Model persistence

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

```
>>> from sklearn import svm
>>> from sklearn import datasets
>>> clf = svm.SVC()
>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)

>>> 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 (`joblib.dump` & `joblib.load`), which is more efficient on big data, but can only pickle to the disk and not to a string:

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

Later you can load back the pickled model (possibly in another Python process) with:

```
>>> clf = joblib.load('filename.pkl')
```

Note: joblib.dump returns a list of filenames. Each individual numpy array contained in the `clf` object is serialized as a separate file on the filesystem. All files are required in the same folder when reloading the model with joblib.load.

Note that pickle has some security and maintainability issues. Please refer to section *Model persistence* for more detailed information about model persistence with scikit-learn.