

3.1 Supervised learning

3.1.1 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 features. In mathematical notation, if \hat{y} is the predicted value.

$$\hat{y}(w, x) = w_0 + w_1x_1 + \dots + w_px_p$$

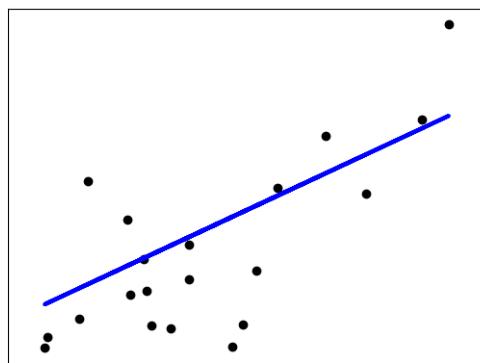
Across the module, we designate the vector $w = (w_1, \dots, w_p)$ as `coef_` and w_0 as `intercept_`.

To perform classification with generalized linear models, see [Logistic regression](#).

Ordinary Least Squares

[LinearRegression](#) fits a linear model with coefficients $w = (w_1, \dots, w_p)$ to minimize the residual sum of squares between the observed targets in the dataset, and the targets 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
>>> reg = linear_model.LinearRegression()
>>> reg.fit([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
...
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
normalize=False)
>>> reg.coef_
array([0.5, 0.5])
```

The coefficient estimates for Ordinary Least Squares rely on the independence of the features. When features 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 target, producing a large variance. This situation of *multicollinearity* can arise, for example, when data are collected without an experimental design.

Examples:

- *Linear Regression Example*

Ordinary Least Squares Complexity

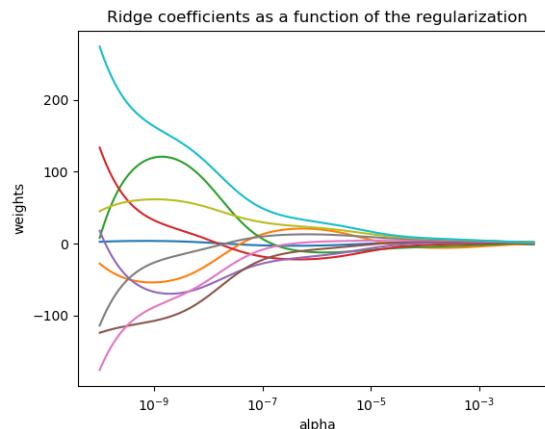
The least squares solution is computed using the singular value decomposition of X . If X is a matrix of shape $(n_{\text{samples}}, n_{\text{features}})$ this method has a cost of $O(n_{\text{samples}}n_{\text{features}}^2)$, assuming that $n_{\text{samples}} \geq n_{\text{features}}$.

Ridge Regression

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

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

The complexity parameter $\alpha \geq 0$ 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
>>> reg = linear_model.Ridge(alpha=.5)
>>> reg.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, random_state=None, solver='auto', tol=0.001)
>>> reg.coef_
array([0.34545455, 0.34545455])
>>> reg.intercept_
0.13636...
```

Examples:

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

Ridge Complexity

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

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:

```
>>> import numpy as np
>>> from sklearn import linear_model
>>> reg = linear_model.RidgeCV(alphas=np.logspace(-6, 6, 13))
>>> reg.fit([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
RidgeCV(alphas=array([1.e-06, 1.e-05, 1.e-04, 1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01,
       1.e+02, 1.e+03, 1.e+04, 1.e+05, 1.e+06]),
        cv=None, fit_intercept=True, gcv_mode=None, normalize=False,
        scoring=None, store_cv_values=False)
>>> reg.alpha_
0.01
```

Specifying the value of the *cv* attribute will trigger the use of cross-validation with *GridSearchCV*, for example *cv=10* for 10-fold cross-validation, rather than Generalized Cross-Validation.

References

- “Notes on Regularized Least Squares”, Rifkin & Lippert ([technical report](#), [course slides](#)).

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 non-zero coefficients, effectively reducing the number of features upon which the given solution is dependent. For this reason Lasso and its variants are fundamental to the field of compressed sensing.

Under certain conditions, it can recover the exact set of non-zero coefficients (see [Compressive sensing: tomography reconstruction with L1 prior \(Lasso\)](#)).

Mathematically, it consists of a linear model with an added regularization term. The objective function to minimize is:

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

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 coefficient vector.

The implementation in the class [`Lasso`](#) uses coordinate descent as the algorithm to fit the coefficients. See [Least Angle Regression](#) for another implementation:

```
>>> from sklearn import linear_model
>>> reg = linear_model.Lasso(alpha=0.1)
>>> reg.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=False, random_state=None,
      selection='cyclic', tol=0.0001, warm_start=False)
>>> reg.predict([[1, 1]])
array([0.8])
```

The function [`lasso_path`](#) is useful for lower-level tasks, as it 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](#).

The following two references explain the iterations used in the coordinate descent solver of scikit-learn, as well as the duality gap computation used for convergence control.

References

- “Regularization Path For Generalized linear Models by Coordinate Descent”, Friedman, Hastie & Tibshirani, J Stat Softw, 2010 ([Paper](#)).
- “An Interior-Point Method for Large-Scale L1-Regularized Least Squares,” S. J. Kim, K. Koh, M. Lustig, S. Boyd and D. Gorinevsky, in IEEE Journal of Selected Topics in Signal Processing, 2007 ([Paper](#))

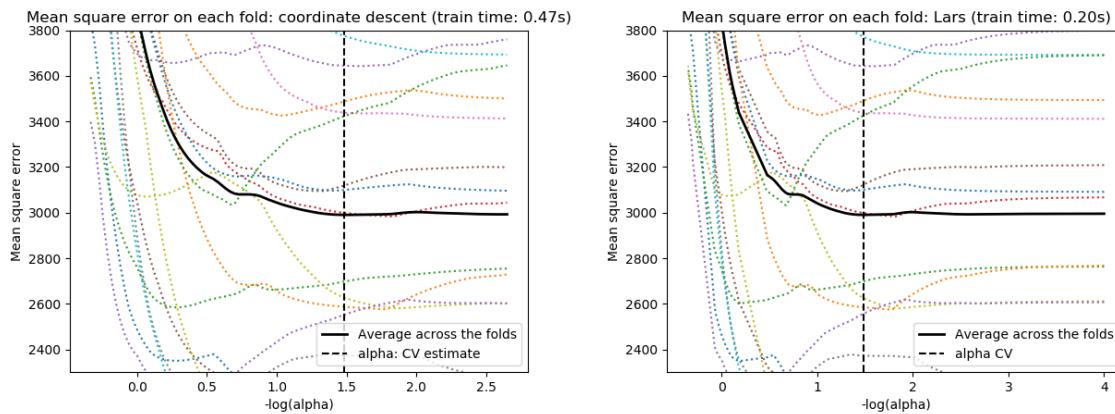
Setting regularization parameter

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

Using cross-validation

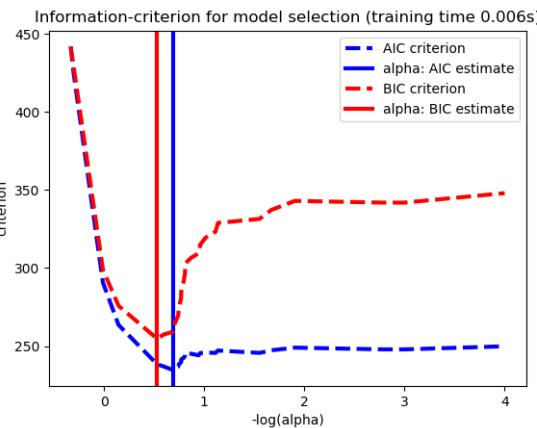
scikit-learn exposes objects that set the Lasso alpha parameter by cross-validation: `LassoCV` and `LassoLarsCV`. `LassoLarsCV` is based on the *Least Angle Regression* algorithm explained below.

For high-dimensional datasets with many collinear features, `LassoCV` is most often preferable. However, `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 features, it is often faster than `LassoCV`.



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*

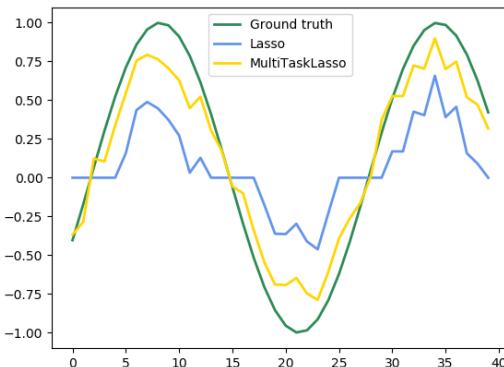
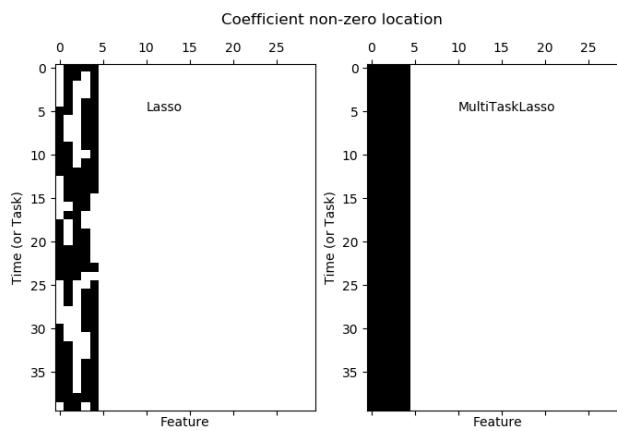
Comparison with the regularization parameter of SVM

The equivalence between alpha and the regularization parameter of SVM, C is given by $\text{alpha} = 1 / \text{C}$ or $\text{alpha} = 1 / (\text{n_samples} * \text{C})$, depending on the estimator and the exact objective function optimized by the model.

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 $(\text{n_samples}, \text{n_tasks})$. 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-zero entries in the coefficient matrix W obtained with a simple Lasso or a MultiTaskLasso. The Lasso estimates yield 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 $\ell_1 \ell_2$ -norm for regularization. The objective function

to minimize is:

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

where Fro indicates the Frobenius norm

$$\|A\|_{\text{Fro}} = \sqrt{\sum_{ij} a_{ij}^2}$$

and ℓ_1 ℓ_2 reads

$$\|A\|_{21} = \sum_i \sqrt{\sum_j a_{ij}^2}.$$

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

Elastic-Net

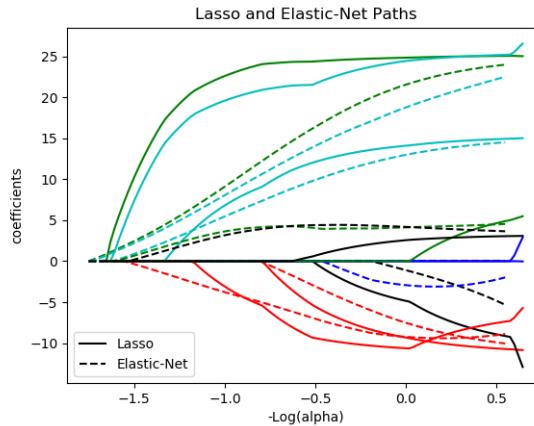
`ElasticNet` is a linear regression model trained with both ℓ_1 and ℓ_2 -norm regularization of the coefficients. This combination allows for learning a sparse model where few of the weights are non-zero like `Lasso`, while still maintaining the regularization properties of `Ridge`. We control the convex combination of ℓ_1 and ℓ_2 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 that it allows Elastic-Net to inherit some of Ridge's stability under rotation.

The objective function to minimize is in this case

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



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](#)

The following two references explain the iterations used in the coordinate descent solver of scikit-learn, as well as the duality gap computation used for convergence control.

References

- “Regularization Path For Generalized linear Models by Coordinate Descent”, Friedman, Hastie & Tibshirani, J Stat Softw, 2010 ([Paper](#)).
- “An Interior-Point Method for Large-Scale L1-Regularized Least Squares,” S. J. Kim, K. Koh, M. Lustig, S. Boyd and D. Gorinevsky, in IEEE Journal of Selected Topics in Signal Processing, 2007 ([Paper](#))

Multi-task Elastic-Net

The `MultiTaskElasticNet` is an elastic-net 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.

Mathematically, it consists of a linear model trained with a mixed ℓ_1 ℓ_2 -norm and ℓ_2 -norm for regularization. The objective function to minimize is:

$$\min_W \frac{1}{2n_{\text{samples}}} \|XW - Y\|_{\text{Fro}}^2 + \alpha\rho\|W\|_{21} + \frac{\alpha(1-\rho)}{2}\|W\|_{\text{Fro}}^2$$

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

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

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. LARS is similar to forward stepwise regression. At each step, it finds the feature most correlated with the target. When there are multiple features having equal correlation, instead of continuing along the same feature, it proceeds in a direction equiangular between the features.

The advantages of LARS are:

- It is numerically efficient in contexts where the number of features is significantly greater than the number of samples.
- It is computationally just as fast as forward selection and has the same order of complexity as 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 features are almost equally correlated with the target, 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.

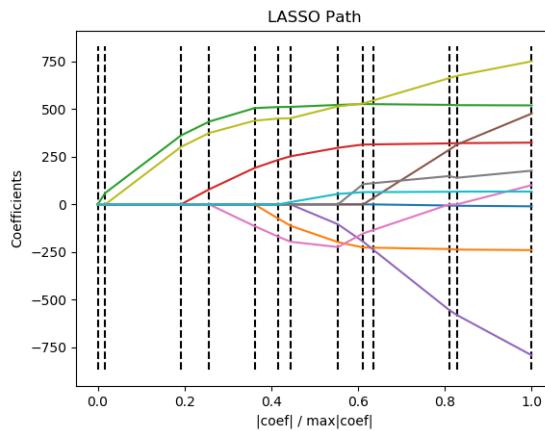
The disadvantages of the LARS method include:

- 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` or `lars_path_gram`.

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
>>> reg = linear_model.LassoLars(alpha=.1)
>>> reg.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, positive=False,
    precompute='auto', verbose=False)
>>> reg.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 is to retrieve the path with one of the functions `lars_path` or `lars_path_gram`.

Mathematical formulation

The algorithm is similar to forward stepwise regression, but instead of including features at each step, the estimated coefficients 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 ℓ_1 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.

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:

$$\arg \min_{\gamma} \|y - X\gamma\|_2^2 \text{ subject to } \|\gamma\|_0 \leq n_{\text{nonzero_coefs}}$$

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

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

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:

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

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 posteriori estimation under a Gaussian prior over the coefficients w with precision λ^{-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 :

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

where α 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

Bayesian Ridge Regression

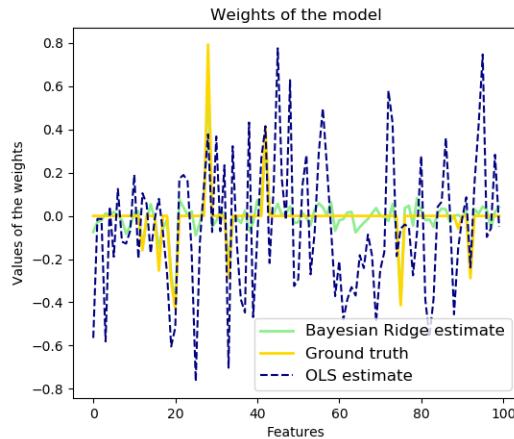
`BayesianRidge` estimates a probabilistic model of the regression problem as described above. The prior for the coefficient w is given by a spherical Gaussian:

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

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 regularization parameters α and λ being estimated by maximizing the *log marginal likelihood*. The scikit-learn implementation is based on the algorithm described in Appendix A of (Tipping, 2001) where the update of the parameters α and λ is done as suggested in (MacKay, 1992).

The remaining hyperparameters are the parameters α_1 , α_2 , λ_1 and λ_2 of the gamma priors over α and λ . These are usually chosen to be *non-informative*. By default $\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 10^{-6}$.



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.]
>>> reg = linear_model.BayesianRidge()
>>> reg.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:

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

The coefficients w of the model can be accessed:

```
>>> reg.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 problems.

Examples:

- *Bayesian Ridge Regression*

References:

- Section 3.3 in Christopher M. Bishop: Pattern Recognition and Machine Learning, 2006
- David J. C. MacKay, *Bayesian Interpolation*, 1992.
- Michael E. Tipping, *Sparse Bayesian Learning and the Relevance Vector Machine*, 2001.

Automatic Relevance Determination - ARD

ARDRegression is very similar to *Bayesian Ridge Regression*, but can lead to sparser coefficients w ¹². *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 coefficient 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 $\text{diag}(A) = \lambda = \{\lambda_1, \dots, \lambda_p\}$.

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 .

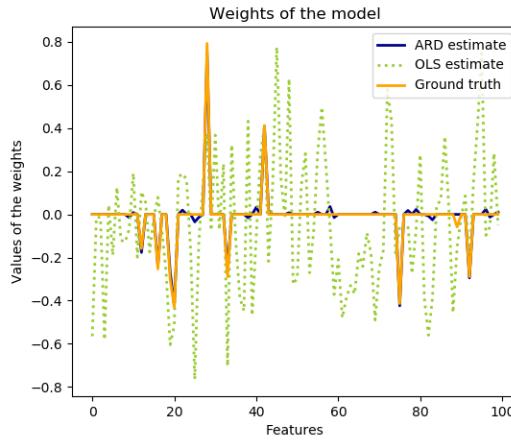
ARD is also known in the literature as *Sparse Bayesian Learning* and *Relevance Vector Machine*³⁴.

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

³ Michael E. Tipping: Sparse Bayesian Learning and the Relevance Vector Machine

⁴ Tristan Fletcher: Relevance Vector Machines explained



Examples:

- *Automatic Relevance Determination Regression (ARD)*

References:

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](#).

Logistic regression is implemented in [`LogisticRegression`](#). This implementation can fit binary, One-vs-Rest, or multinomial logistic regression with optional ℓ_1 , ℓ_2 or Elastic-Net regularization.

Note: Regularization is applied by default, which is common in machine learning but not in statistics. Another advantage of regularization is that it improves numerical stability. No regularization amounts to setting C to a very high value.

As an optimization problem, binary class ℓ_2 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, ℓ_1 regularized logistic regression solves the following optimization problem:

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

Elastic-Net regularization is a combination of ℓ_1 and ℓ_2 , and minimizes the following cost function:

$$\min_{w,c} \frac{1-\rho}{2} w^T w + \rho \|w\|_1 + C \sum_{i=1}^n \log(\exp(-y_i(X_i^T w + c)) + 1),$$

where ρ controls the strength of ℓ_1 regularization vs. ℓ_2 regularization (it corresponds to the `l1_ratio` parameter).

Note that, in this notation, it's assumed that the target y_i takes values in the set $-1, 1$ at trial i . We can also see that Elastic-Net is equivalent to ℓ_1 when $\rho = 1$ and equivalent to ℓ_2 when $\rho = 0$.

The solvers implemented in the class `LogisticRegression` are “liblinear”, “newton-cg”, “lbfgs”, “sag” and “saga”:

The solver “liblinear” uses a coordinate descent (CD) algorithm, and relies on the excellent C++ `LIBLINEAR` library, which is shipped with scikit-learn. However, the CD algorithm implemented in liblinear cannot learn a true multinomial (multiclass) model; instead, the optimization problem is decomposed in a “one-vs-rest” fashion so separate binary classifiers are trained for all classes. This happens under the hood, so `LogisticRegression` instances using this solver behave as multiclass classifiers. For ℓ_1 regularization `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 “lbfgs”, “sag” and “newton-cg” solvers only support ℓ_2 regularization or no regularization, and are found to converge faster for some high-dimensional data. Setting `multi_class` to “multinomial” with these solvers learns a true multinomial logistic regression model⁵, which means that its probability estimates should be better calibrated than the default “one-vs-rest” setting.

The “sag” solver uses Stochastic Average Gradient descent⁶. It is faster than other solvers for large datasets, when both the number of samples and the number of features are large.

The “saga” solver⁷ is a variant of “sag” that also supports the non-smooth `penalty="l1"`. This is therefore the solver of choice for sparse multinomial logistic regression. It is also the only solver that supports `penalty="elasticnet"`.

The “lbfgs” is an optimization algorithm that approximates the Broyden–Fletcher–Goldfarb–Shanno algorithm⁸, which belongs to quasi-Newton methods. The “lbfgs” solver is recommended for use for small data-sets but for larger datasets its performance suffers.⁹

The following table summarizes the penalties supported by each solver:

	Solvers				
Penalties	‘liblinear’	‘lbfgs’	‘newton-cg’	‘sag’	‘saga’
Multinomial + L2 penalty	no	yes	yes	yes	yes
OVR + L2 penalty	yes	yes	yes	yes	yes
Multinomial + L1 penalty	no	no	no	no	yes
OVR + L1 penalty	yes	no	no	no	yes
Elastic-Net	no	no	no	no	yes
No penalty (‘none’)	no	yes	yes	yes	yes
Behaviors					
Penalize the intercept (bad)	yes	no	no	no	no
Faster for large datasets	no	no	no	yes	yes
Robust to unscaled datasets	yes	yes	yes	no	no

The “lbfgs” solver is used by default for its robustness. For large datasets the “saga” solver is usually faster. For large dataset, you may also consider using `SGDClassifier` with ‘log’ loss, which might be even faster but requires more tuning.

⁵ Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 4.3.4

⁶ Mark Schmidt, Nicolas Le Roux, and Francis Bach: Minimizing Finite Sums with the Stochastic Average Gradient.

⁷ Aaron Defazio, Francis Bach, Simon Lacoste-Julien: SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives.

⁸ https://en.wikipedia.org/wiki/Broyden%20%20%93Fletcher%20%20%93Goldfarb%20%20%93Shanno_algorithm

⁹ “Performance Evaluation of Lbfgs vs other solvers”

Examples:

- [L1 Penalty and Sparsity in Logistic Regression](#)
- [Regularization path of L1- Logistic Regression](#)
- [Plot multinomial and One-vs-Rest Logistic Regression](#)
- [Multiclass sparse logistic regression on newsgroups20](#)
- [MNIST classification using multinomial logistic + L1](#)

Differences from liblinear:

There might be a difference in the scores obtained between `LogisticRegression` with `solver=liblinear` or `LinearSVC` and the external liblinear library directly, when `fit_intercept=False` and the fit `coef_` (or) the data to be predicted are zeroes. This is because for the sample(s) with `decision_function zero`, `LogisticRegression` and `LinearSVC` predict the negative class, while liblinear predicts the positive class. Note that a model with `fit_intercept=False` and having many samples with `decision_function zero`, is likely to be a underfit, bad model and you are advised to set `fit_intercept=True` and increase the `intercept_scaling`.

Note: Feature selection with sparse logistic regression

A logistic regression with ℓ_1 penalty yields sparse models, and can thus be used to perform feature selection, as detailed in [L1-based feature selection](#).

`LogisticRegressionCV` implements Logistic Regression with built-in cross-validation support, to find the optimal `C` and `l1_ratio` parameters according to the `scoring` attribute. The “newton-cg”, “sag”, “saga” and “lbfgs” solvers are found to be faster for high-dimensional dense data, due to warm-starting (see [Glossary](#)).

References:**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 online/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](#)

Perceptron

The *Perceptron* is another simple classification 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.

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 .

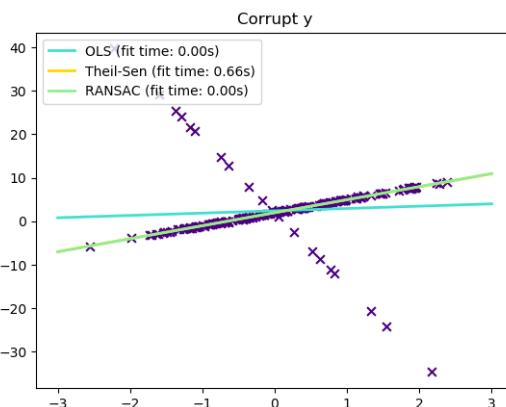
For classification, `PassiveAggressiveClassifier` can be used with `loss='hinge'` (PA-I) or `loss='squared_hinge'` (PA-II). For regression, `PassiveAggressiveRegressor` can be used with `loss='epsilon_insensitive'` (PA-I) or `loss='squared_epsilon_insensitive'` (PA-II).

References:

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

Robustness regression: outliers and modeling errors

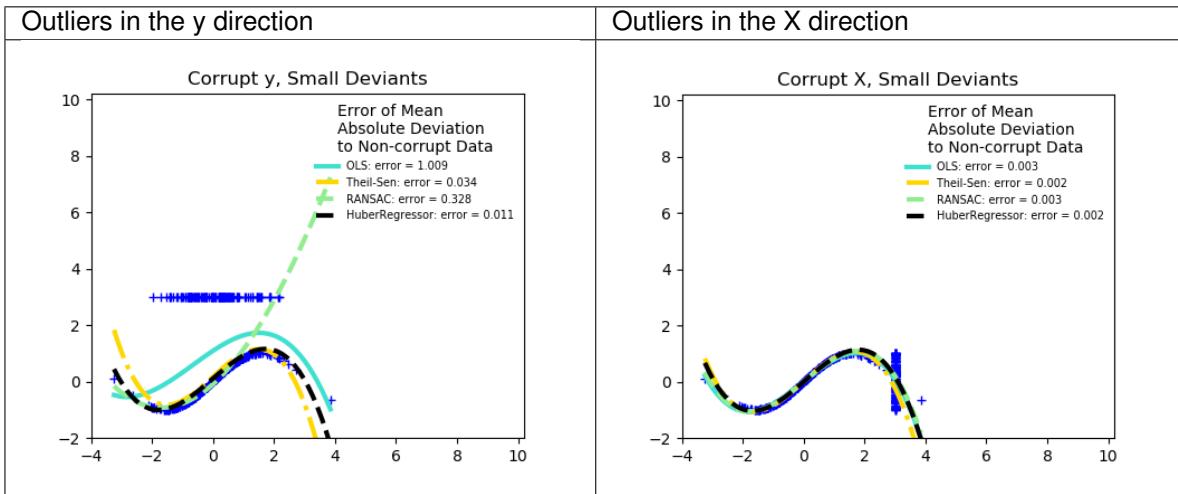
Robust regression aims to fit a regression model in the presence of corrupt data: either outliers, or error in the model.



Different scenario and useful concepts

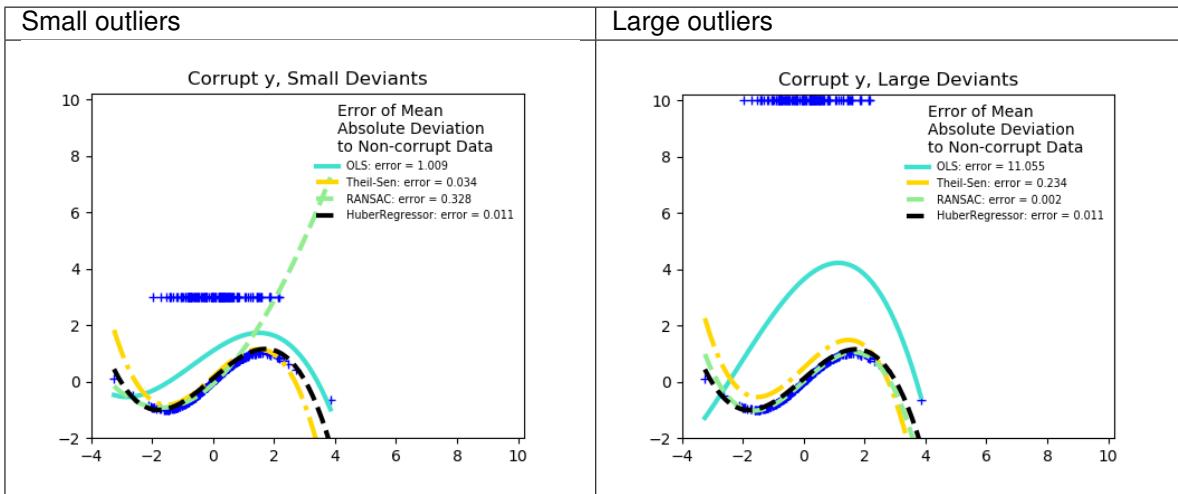
There are different things to keep in mind when dealing with data corrupted by outliers:

- Outliers in X or in y?



- Fraction of outliers versus amplitude of error

The number of outlying points matters, but also how much they are outliers.



An important notion of robust fitting is that of breakdown point: the fraction of data that can be outlying for the fit to start missing the inlying data.

Note that in general, robust fitting in high-dimensional setting (large `n_features`) is very hard. The robust models here will probably not work in these settings.

Trade-offs: which estimator?

Scikit-learn provides 3 robust regression estimators: `RANSAC`, `Theil Sen` and `HuberRegressor`.

- `HuberRegressor` should be faster than `RANSAC` and `Theil Sen` unless the number of samples are very large, i.e `n_samples >> n_features`. This is because `RANSAC` and `Theil Sen` fit on smaller subsets of the data. However, both `Theil Sen` and `RANSAC` are unlikely to be as robust as `HuberRegressor` for the default parameters.

- *RANSAC* is faster than *Theil Sen* and scales much better with the number of samples.
- *RANSAC* will deal better with large outliers in the y direction (most common situation).
- *Theil Sen* will cope better with medium-size outliers in the X direction, but this property will disappear in high-dimensional settings.

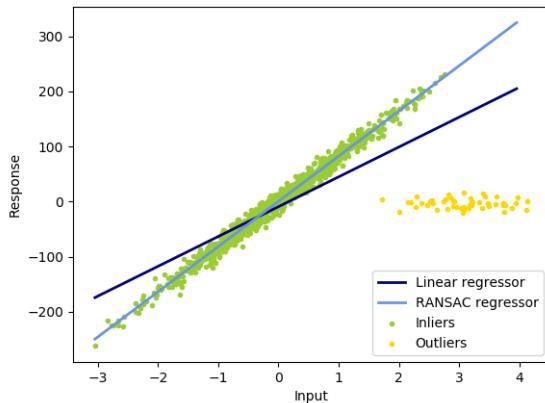
When in doubt, use *RANSAC*.

RANSAC: RANDom SAmple Consensus

RANSAC (RANdom SAmple Consensus) fits a model from random subsets of inliers from the complete data set.

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 field 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.



Details of the algorithm

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*
- *Robust linear estimator fitting*

References:

- <https://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)

Theil-Sen estimator: generalized-median-based estimator

The `TheilSenRegressor` estimator uses a generalization of the median in multiple dimensions. It is thus robust to multivariate outliers. Note however that the robustness of the estimator decreases quickly with the dimensionality of the problem. It loses its robustness properties and becomes no better than an ordinary least squares in high dimension.

Examples:

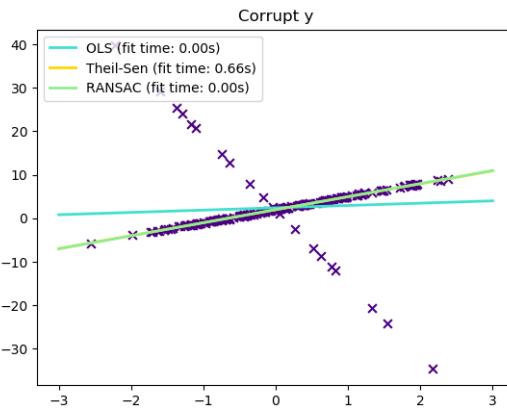
- *Theil-Sen Regression*
- *Robust linear estimator fitting*

References:

- https://en.wikipedia.org/wiki/Theil%20%93Sen_estimator

Theoretical considerations

`TheilSenRegressor` is comparable to the *Ordinary Least Squares (OLS)* in terms of asymptotic efficiency and as an unbiased estimator. In contrast to OLS, Theil-Sen is a non-parametric method which means it makes no assumption about the underlying distribution of the data. Since Theil-Sen is a median-based estimator, it is more robust against corrupted data aka outliers. In univariate setting, Theil-Sen has a breakdown point of about 29.3% in case of a simple linear regression which means that it can tolerate arbitrary corrupted data of up to 29.3%.



The implementation of `TheilSenRegressor` in scikit-learn follows a generalization to a multivariate linear regression model¹⁰ using the spatial median which is a generalization of the median to multiple dimensions¹¹.

In terms of time and space complexity, Theil-Sen scales according to

$$\binom{n_{\text{samples}}}{n_{\text{subsamples}}}$$

which makes it infeasible to be applied exhaustively to problems with a large number of samples and features. Therefore, the magnitude of a subpopulation can be chosen to limit the time and space complexity by considering only a random subset of all possible combinations.

Examples:

- `Theil-Sen Regression`

References:

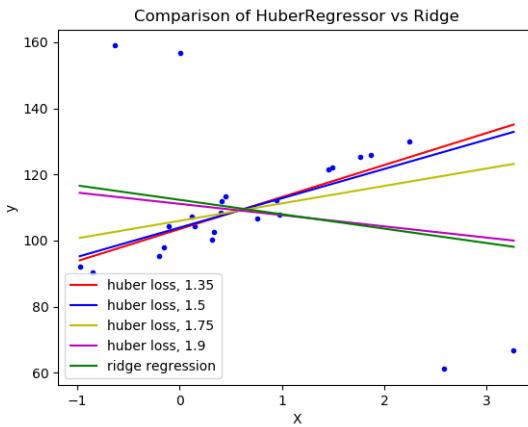
Huber Regression

The `HuberRegressor` is different to `Ridge` because it applies a linear loss to samples that are classified as outliers. A sample is classified as an inlier if the absolute error of that sample is lesser than a certain threshold. It differs from `TheilSenRegressor` and `RANSACRegressor` because it does not ignore the effect of the outliers but gives a lesser weight to them.

The loss function that `HuberRegressor` minimizes is given by

$$\min_{w,\sigma} \sum_{i=1}^n \left(\sigma + H_\epsilon \left(\frac{X_i w - y_i}{\sigma} \right) \sigma \right) + \alpha \|w\|_2^2$$

¹⁰ Xin Dang, Hanxiang Peng, Xueqin Wang and Heping Zhang: Theil-Sen Estimators in a Multiple Linear Regression Model.
¹¹ 20. Kärkkäinen and S. Äyrämö: On Computation of Spatial Median for Robust Data Mining.



where

$$H_\epsilon(z) = \begin{cases} z^2, & \text{if } |z| < \epsilon, \\ 2\epsilon|z| - \epsilon^2, & \text{otherwise} \end{cases}$$

It is advised to set the parameter `epsilon` to 1.35 to achieve 95% statistical efficiency.

Notes

The `HuberRegressor` differs from using `SGDRegressor` with loss set to `huber` in the following ways.

- `HuberRegressor` is scaling invariant. Once `epsilon` is set, scaling X and y down or up by different values would produce the same robustness to outliers as before. as compared to `SGDRegressor` where `epsilon` has to be set again when X and y are scaled.
- `HuberRegressor` should be more efficient to use on data with small number of samples while `SGDRegressor` needs a number of passes on the training data to produce the same robustness.

Examples:

- `HuberRegressor vs Ridge on dataset with strong outliers`

References:

- Peter J. Huber, Elvezio M. Ronchetti: Robust Statistics, Concomitant scale estimates, pg 172

Note that this estimator is different from the R implementation of Robust Regression (<http://www.ats.ucla.edu/stat/r/dae/rreg.htm>) because the R implementation does a weighted least squares implementation with weights given to each sample on the basis of how much the residual is greater than a certain threshold.

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 set of features

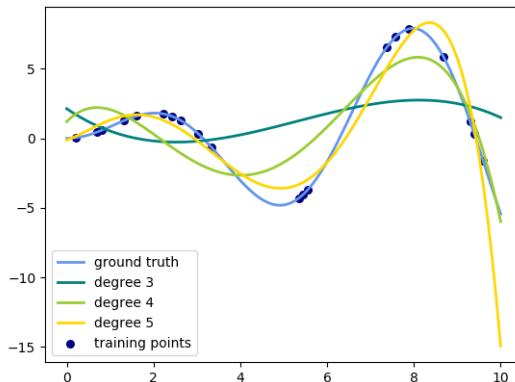
$$z = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]$$

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

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

We see that the resulting *polynomial regression* is in the same class of linear models we 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` transformer, which 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 $[1, x_1, x_2, x_1^2, x_1x_2, x_2^2]$, 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
>>> import numpy as np
>>> 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
>>> import numpy as np
>>> X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
>>> y = X[:, 0] ^ X[:, 1]
>>> y
array([0, 1, 1, 0])
>>> X = PolynomialFeatures(interaction_only=True).fit_transform(X).astype(int)
>>> X
array([[1, 0, 0, 0],
       [1, 0, 1, 0],
       [1, 1, 0, 0],
       [1, 1, 1, 1]])
>>> clf = Perceptron(fit_intercept=False, max_iter=10, tol=None,
...                    shuffle=False).fit(X, y)
```

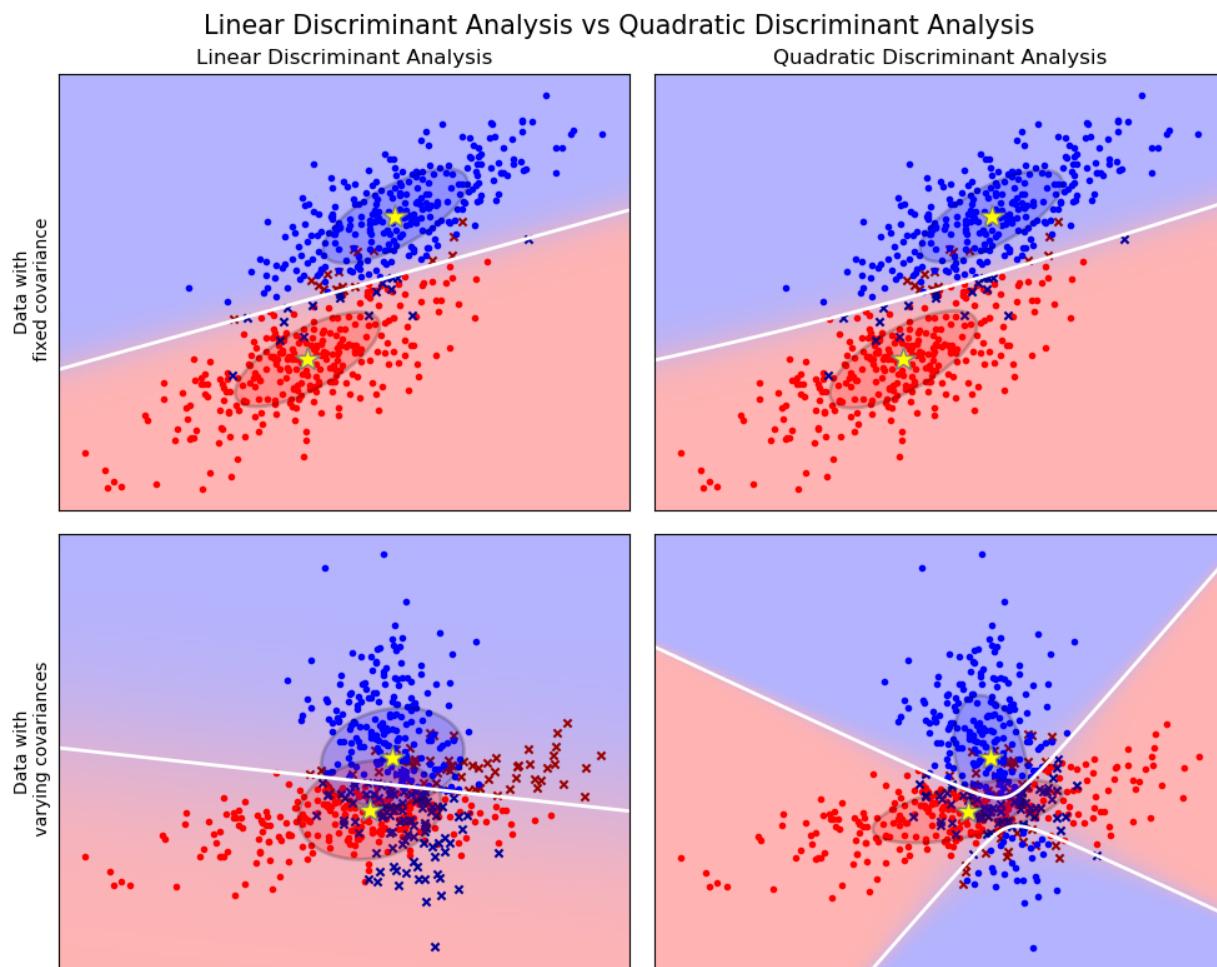
And the classifier "predictions" are perfect:

```
>>> clf.predict(X)
array([0, 1, 1, 0])
>>> clf.score(X, y)
1.0
```

3.1.2 Linear and Quadratic Discriminant Analysis

Linear Discriminant Analysis ([discriminant_analysis.LinearDiscriminantAnalysis](#)) and Quadratic Discriminant Analysis ([discriminant_analysis.QuadraticDiscriminantAnalysis](#)) 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, have proven to work well in practice, and have no hyperparameters to tune.



The plot shows decision boundaries for Linear Discriminant Analysis and Quadratic Discriminant Analysis. The bottom row demonstrates that Linear Discriminant Analysis can only learn linear boundaries, while Quadratic Discriminant Analysis can learn quadratic boundaries and is therefore more flexible.

Examples:

[*Linear and Quadratic Discriminant Analysis with covariance ellipsoid*](#): Comparison of LDA and QDA on synthetic data.

Dimensionality reduction using Linear Discriminant Analysis

[*discriminant_analysis.LinearDiscriminantAnalysis*](#) can be used to perform supervised dimensionality reduction, by projecting the input data to a linear subspace consisting of the directions which maximize the separation between classes (in a precise sense discussed in the mathematics section below). The dimension of the output is necessarily less than the number of classes, so this is, in general, a rather strong dimensionality reduction, and only makes sense in a multiclass setting.

This is implemented in `discriminant_analysis.LinearDiscriminantAnalysis.transform`. The desired dimensionality can be set using the `n_components` constructor parameter. This parameter has no influence on `discriminant_analysis.LinearDiscriminantAnalysis.fit` or `discriminant_analysis.LinearDiscriminantAnalysis.predict`.

Examples:

Comparison of LDA and PCA 2D projection of Iris dataset: Comparison of LDA and PCA for dimensionality reduction of the Iris dataset

Mathematical formulation of the LDA and QDA classifiers

Both LDA and QDA can be derived from simple probabilistic models which model the class conditional distribution of the data $P(X|y = k)$ for each class k . Predictions can then be obtained by using Bayes' rule:

$$P(y = k|X) = \frac{P(X|y = k)P(y = k)}{P(X)} = \frac{P(X|y = k)P(y = k)}{\sum_l P(X|y = l) \cdot P(y = l)}$$

and we select the class k which maximizes this conditional probability.

More specifically, for linear and quadratic discriminant analysis, $P(X|y)$ is modeled as a multivariate Gaussian distribution with density:

$$P(X|y = k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(X - \mu_k)^t \Sigma_k^{-1} (X - \mu_k)\right)$$

where d is the number of features.

To use this model as a classifier, we just need to estimate from the training data the class priors $P(y = k)$ (by the proportion of instances of class k), the class means μ_k (by the empirical sample class means) and the covariance matrices (either by the empirical sample class covariance matrices, or by a regularized estimator: see the section on shrinkage below).

In the case of LDA, the Gaussians for each class are assumed to share the same covariance matrix: $\Sigma_k = \Sigma$ for all k . This leads to linear decision surfaces, which can be seen by comparing the log-probability ratios $\log[P(y = k|X)/P(y = l|X)]$:

$$\begin{aligned} \log\left(\frac{P(y = k|X)}{P(y = l|X)}\right) &= \log\left(\frac{P(X|y = k)P(y = k)}{P(X|y = l)P(y = l)}\right) = 0 \Leftrightarrow \\ (\mu_k - \mu_l)^t \Sigma^{-1} X &= \frac{1}{2}(\mu_k^t \Sigma^{-1} \mu_k - \mu_l^t \Sigma^{-1} \mu_l) - \log \frac{P(y = k)}{P(y = l)} \end{aligned}$$

In the case of QDA, there are no assumptions on the covariance matrices Σ_k of the Gaussians, leading to quadratic decision surfaces. See³ for more details.

Note: Relation with Gaussian Naive Bayes

If in the QDA model one assumes that the covariance matrices are diagonal, then the inputs are assumed to be conditionally independent in each class, and the resulting classifier is equivalent to the Gaussian Naive Bayes classifier `naive_bayes.GaussianNB`.

³ “The Elements of Statistical Learning”, Hastie T., Tibshirani R., Friedman J., Section 4.3, p.106-119, 2008.

Mathematical formulation of LDA dimensionality reduction

To understand the use of LDA in dimensionality reduction, it is useful to start with a geometric reformulation of the LDA classification rule explained above. We write K for the total number of target classes. Since in LDA we assume that all classes have the same estimated covariance Σ , we can rescale the data so that this covariance is the identity:

$$X^* = D^{-1/2}U^t X \text{ with } \Sigma = UDU^t$$

Then one can show that to classify a data point after scaling is equivalent to finding the estimated class mean μ_k^* which is closest to the data point in the Euclidean distance. But this can be done just as well after projecting on the $K - 1$ affine subspace H_K generated by all the μ_k^* for all classes. This shows that, implicit in the LDA classifier, there is a dimensionality reduction by linear projection onto a $K - 1$ dimensional space.

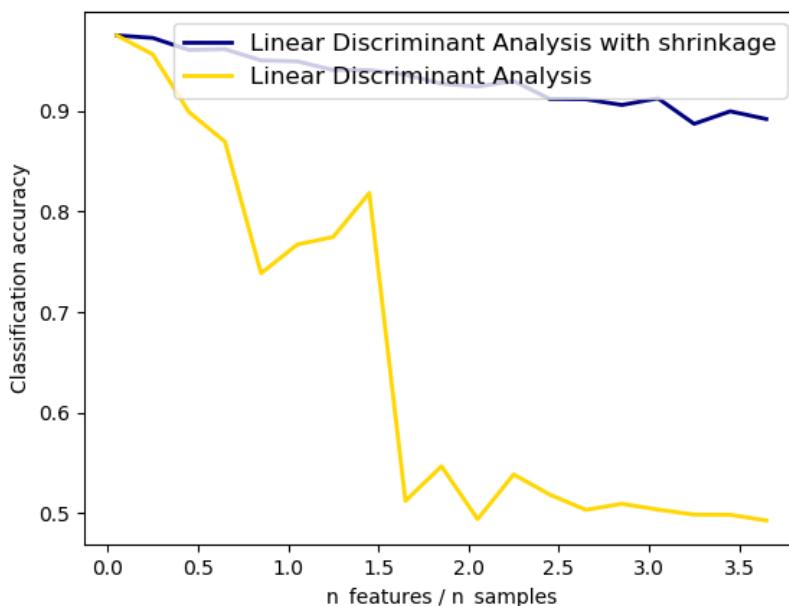
We can reduce the dimension even more, to a chosen L , by projecting onto the linear subspace H_L which maximizes the variance of the μ_k^* after projection (in effect, we are doing a form of PCA for the transformed class means μ_k^*). This L corresponds to the `n_components` parameter used in the [`discriminant_analysis`](#). [`LinearDiscriminantAnalysis.transform`](#) method. See³ for more details.

Shrinkage

Shrinkage is a tool to improve estimation of covariance matrices in situations where the number of training samples is small compared to the number of features. In this scenario, the empirical sample covariance is a poor estimator. Shrinkage LDA can be used by setting the `shrinkage` parameter of the [`discriminant_analysis`](#). [`LinearDiscriminantAnalysis`](#) class to ‘auto’. This automatically determines the optimal shrinkage parameter in an analytic way following the lemma introduced by Ledoit and Wolf⁴. Note that currently shrinkage only works when setting the `solver` parameter to ‘lsqr’ or ‘eigen’.

The `shrinkage` parameter can also be manually set between 0 and 1. In particular, a value of 0 corresponds to no shrinkage (which means the empirical covariance matrix will be used) and a value of 1 corresponds to complete shrinkage (which means that the diagonal matrix of variances will be used as an estimate for the covariance matrix). Setting this parameter to a value between these two extrema will estimate a shrunk version of the covariance matrix.

`discriminant Analysis vs. shrinkage Linear Discriminant Analysis (1 discriminant)`



⁴ Ledoit O, Wolf M. Honey, I Shrunk the Sample Covariance Matrix. The Journal of Portfolio Management 30(4), 110-119, 2004.

Estimation algorithms

The default solver is ‘svd’. It can perform both classification and transform, and it does not rely on the calculation of the covariance matrix. This can be an advantage in situations where the number of features is large. However, the ‘svd’ solver cannot be used with shrinkage.

The ‘lsqr’ solver is an efficient algorithm that only works for classification. It supports shrinkage.

The ‘eigen’ solver is based on the optimization of the between class scatter to within class scatter ratio. It can be used for both classification and transform, and it supports shrinkage. However, the ‘eigen’ solver needs to compute the covariance matrix, so it might not be suitable for situations with a high number of features.

Examples:

Normal and Shrinkage Linear Discriminant Analysis for classification: Comparison of LDA classifiers with and without shrinkage.

References:

3.1.3 Kernel ridge regression

Kernel ridge regression (KRR) [M2012] combines *Ridge Regression* (linear least squares with l2-norm regularization) with the kernel trick. It thus learns a linear function in the space induced by the respective kernel and the data. For non-linear kernels, this corresponds to a non-linear function in the original space.

The form of the model learned by *KernelRidge* is identical to support vector regression (SVR). However, different loss functions are used: KRR uses squared error loss while support vector regression uses ϵ -insensitive loss, both combined with l2 regularization. In contrast to SVR, fitting *KernelRidge* can be done in closed-form and is typically faster for medium-sized datasets. On the other hand, the learned model is non-sparse and thus slower than SVR, which learns a sparse model for $\epsilon > 0$, at prediction-time.

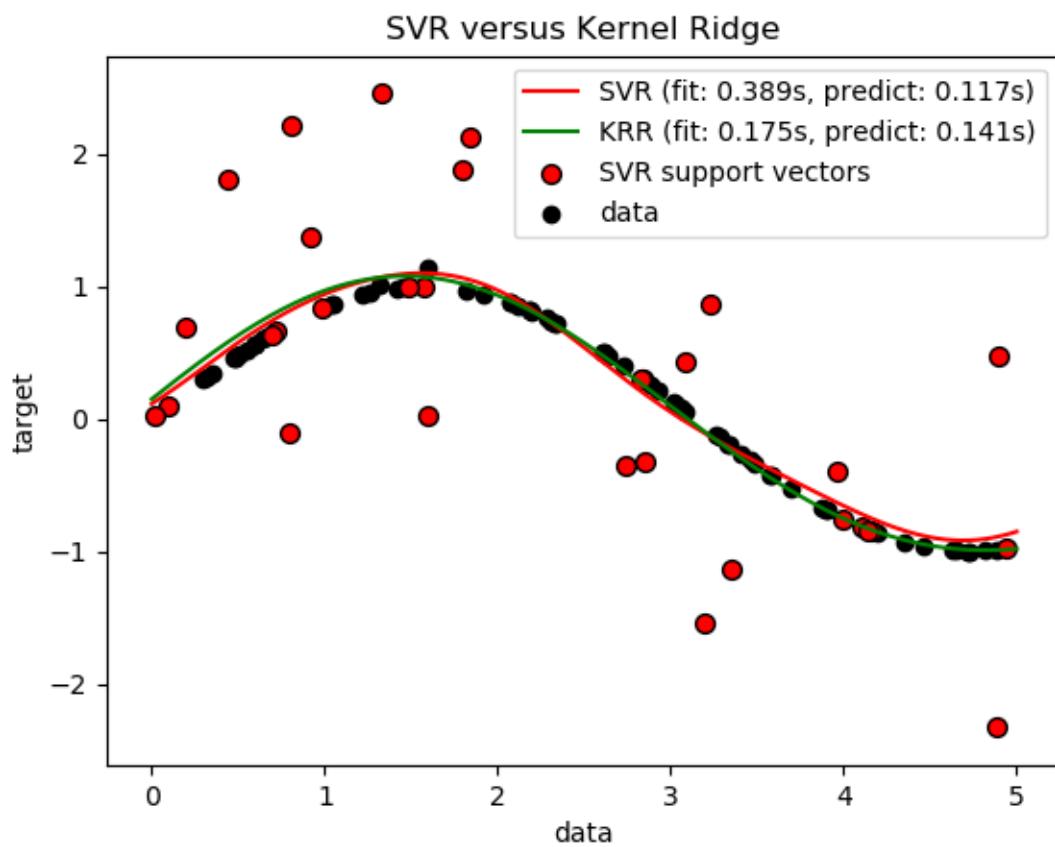
The following figure compares *KernelRidge* and SVR on an artificial dataset, which consists of a sinusoidal target function and strong noise added to every fifth datapoint. The learned model of *KernelRidge* and SVR is plotted, where both complexity/regularization and bandwidth of the RBF kernel have been optimized using grid-search. The learned functions are very similar; however, fitting *KernelRidge* is approx. seven times faster than fitting SVR (both with grid-search). However, prediction of 100000 target values is more than three times faster with SVR since it has learned a sparse model using only approx. 1/3 of the 100 training datapoints as support vectors.

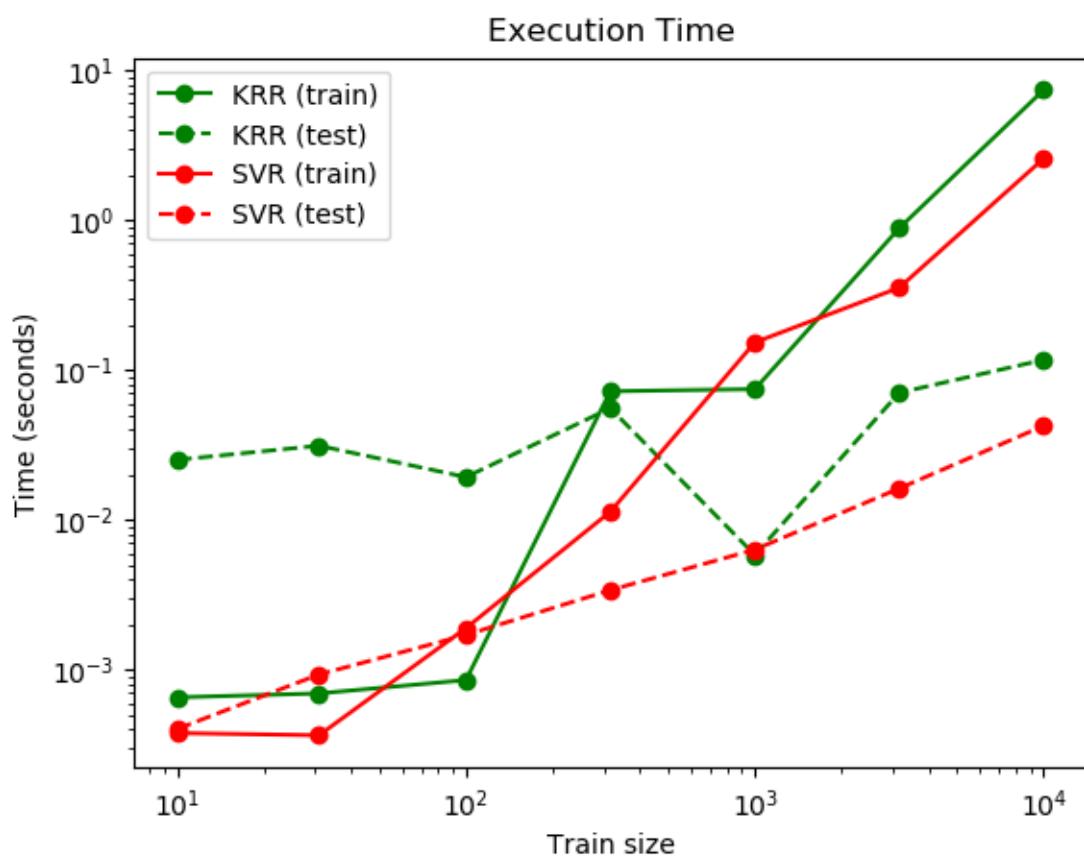
The next figure compares the time for fitting and prediction of *KernelRidge* and SVR for different sizes of the training set. Fitting *KernelRidge* is faster than SVR for medium-sized training sets (less than 1000 samples); however, for larger training sets SVR scales better. With regard to prediction time, SVR is faster than *KernelRidge* for all sizes of the training set because of the learned sparse solution. Note that the degree of sparsity and thus the prediction time depends on the parameters ϵ and C of the SVR; $\epsilon = 0$ would correspond to a dense model.

References:

3.1.4 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.

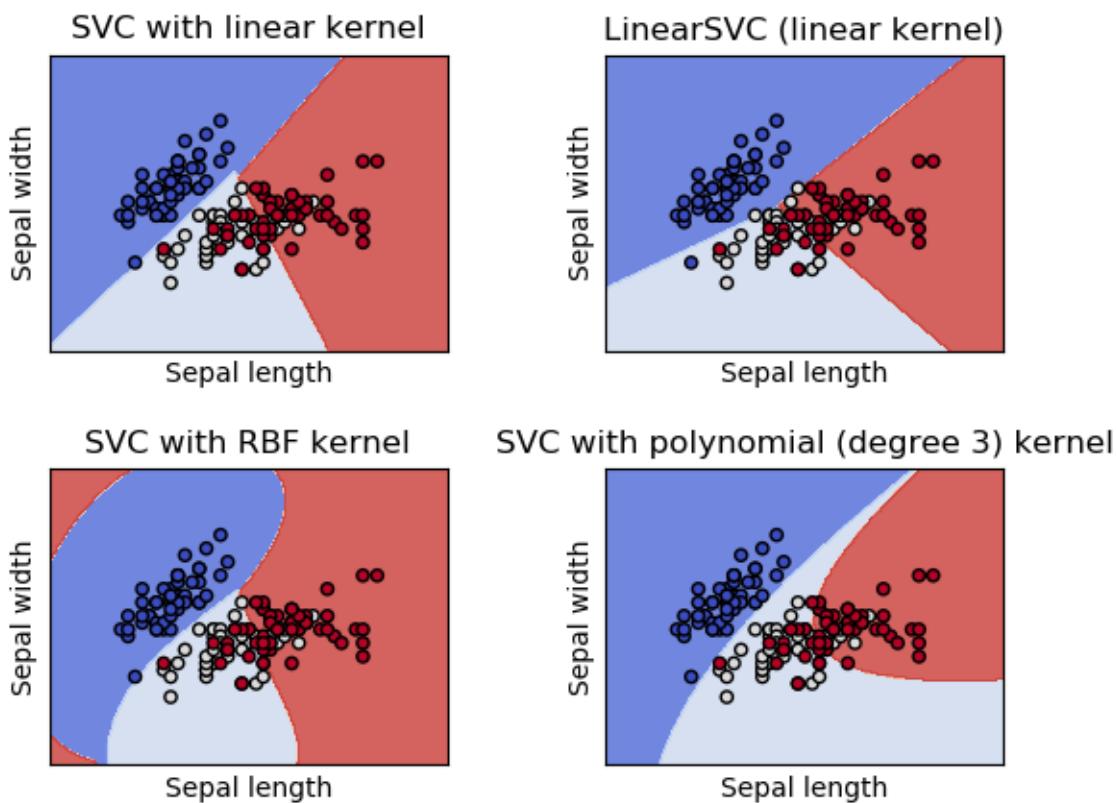
The disadvantages of support vector machines include:

- If the number of features is much greater than the number of samples, avoid over-fitting in choosing *Kernel functions* and regularization term is crucial.
- 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`.

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(gamma='scale')
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', 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]...)
```

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. To provide a consistent interface with other classifiers, the `decision_function_shape` option allows to monotonically transform the results of the “one-against-one” classifiers to a decision function of shape (`n_samples`, `n_classes`).

```
>>> X = [[0, 1], [2, 3]]
>>> Y = [0, 1, 2, 3]
>>> clf = svm.SVC(gamma='scale', decision_function_shape='ovo')
>>> clf.fit(X, Y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovo', degree=3, gamma='scale', 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
```

```
>>> clf.decision_function_shape = "ovr"
>>> dec = clf.decision_function([[1]])
>>> dec.shape[1] # 4 classes
4
```

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='squared_hinge', max_iter=1000,
          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 attributes `coef_` and `intercept_` have the shape `[n_class * (n_class - 1) / 2, n_features]` and `[n_class * (n_class - 1) / 2]` respectively. This is similar to the layout for `LinearSVC` described above, with each row now corresponding to a binary classifier. 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 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_2^0, v_2^1 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
$\alpha_{2,0}^1$	$\alpha_{2,1}^1$	

Scores and probabilities

The `decision_function` method of `SVC` and `NuSVC` 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 $< \frac{1}{2}$ 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`.

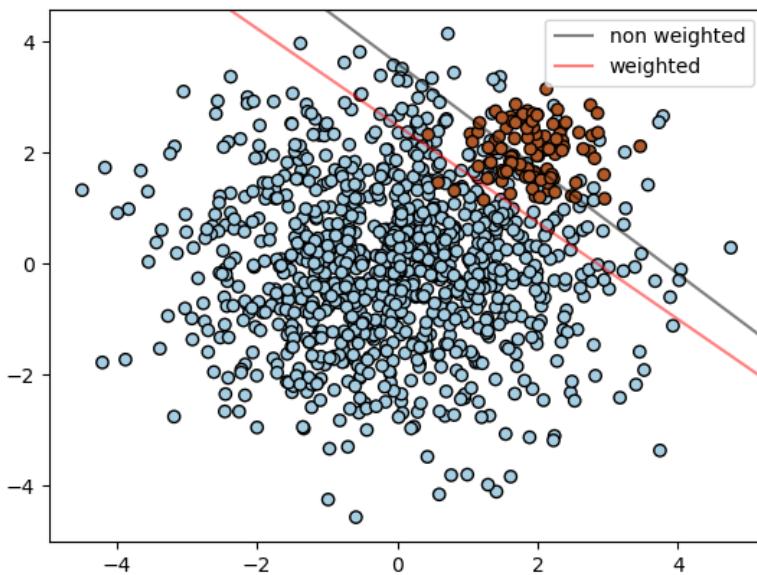
References:

- Wu, Lin and Weng, “Probability estimates for multi-class classification by pairwise coupling”, JMLR 5:975-1005, 2004.
- Platt “Probabilistic outputs for SVMs and comparisons to regularized likelihood methods”.

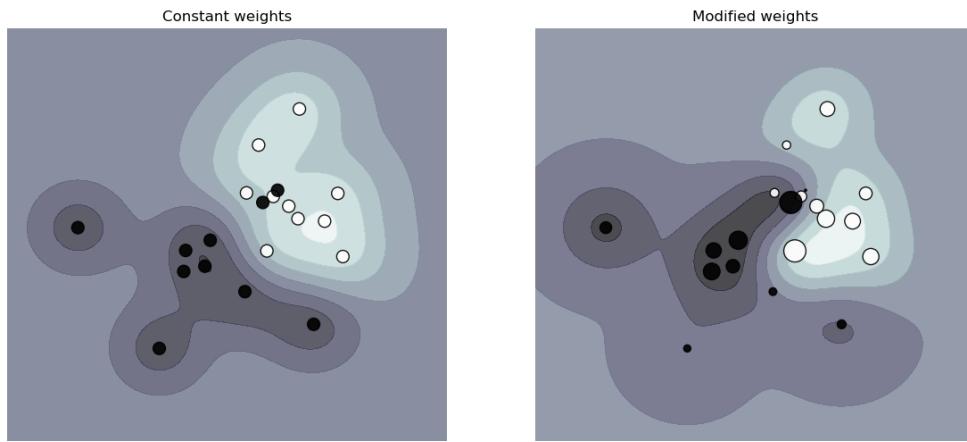
Unbalanced problems

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 $C * \text{value}$.



`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](#),

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 three different implementations of Support Vector Regression: `SVR`, `NuSVR` and `LinearSVR`. `LinearSVR` provides a faster implementation than `SVR` but only considers linear kernels, while `NuSVR` implements a slightly different formulation than `SVR` and `LinearSVR`. See [Implementation details](#) for further details.

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='auto_deprecated', kernel='rbf', max_iter=-1, 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*

Density estimation, novelty detection

The class `OneClassSVM` implements a One-Class SVM which is used in outlier detection.

See *Novelty and Outlier Detection* for the description and usage of `OneClassSVM`.

Complexity

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

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

Tips on Practical Use

- **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 given 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.

`LinearSVC` and `LinearSVR` are less sensitive to C when it becomes large, and prediction results stop improving after a certain threshold. Meanwhile, larger C values will take more time to train, sometimes up to 10 times longer, as shown by Fan et al. (2008)

- 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='balanced'` and/or try different penalty parameters C.
- Randomness of the underlying implementations:** The underlying implementations of [SVC](#) and [NuSVC](#) use a random number generator only to shuffle the data for probability estimation (when `probability` is set to `True`). This randomness can be controlled with the `random_state` parameter. If `probability` is set to `False` these estimators are not random and `random_state` has no effect on the results. The underlying [OneClassSVM](#) implementation is similar to the ones of [SVC](#) and [NuSVC](#). As no probability estimation is provided for [OneClassSVM](#), it is not random.

The underlying [LinearSVC](#) implementation uses a random number generator to select features when fitting the model with a dual coordinate descent (i.e when `dual` is set to `True`). It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller `tol` parameter. This randomness can also be controlled with the `random_state` parameter. When `dual` is set to `False` the underlying implementation of [LinearSVC](#) is not random and `random_state` has no effect on the results.

- 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](#).

References:

- Fan, Rong-En, et al., “LIBLINEAR: A library for large linear classification.”, Journal of machine learning research 9.Aug (2008): 1871-1874.

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'
```

Custom Kernels

You can define your own kernels by either giving the kernel as a python function or by precomputing the Gram matrix. Classifiers with custom kernels behave the same way as any other classifiers, except that:

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

Using Python functions as kernels

You can also use your own defined kernels by passing a function to the keyword `kernel` in the constructor.

Your kernel must take as arguments two matrices of shape `(n_samples_1, n_features)`, `(n_samples_2, n_features)` and return a kernel matrix of shape `(n_samples_1, n_samples_2)`.

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.*

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,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    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])
```

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.

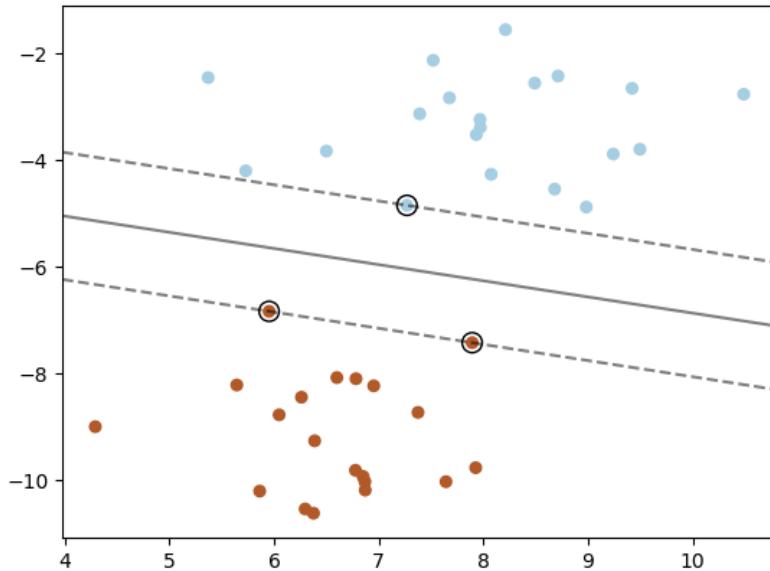
Proper choice of `C` and `gamma` is critical to the SVM's performance. One is advised to use `sklearn.model_selection.GridSearchCV` with `C` and `gamma` spaced exponentially far apart to choose good values.

Examples:

- `RBF SVM parameters`

Mathematical formulation

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



SVC

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

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

Its dual is

$$\begin{aligned} & \min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha \\ & \text{subject to } y^T \alpha = 0 \\ & \quad 0 \leq \alpha_i \leq C, i = 1, \dots, n \end{aligned}$$

where e is the vector of all ones, $C > 0$ is the upper bound, Q is an n by n positive semidefinite matrix, $Q_{ij} \equiv y_i y_j K(x_i, x_j)$, where $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ .

The decision function is:

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

Note: While SVM models derived from `libsvm` and `liblinear` use `C` as regularization parameter, most other estimators use `alpha`. The exact equivalence between the amount of regularization of two models depends on the exact objective function optimized by the model. For example, when the estimator used is `sklearn.linear_model.Ridge` regression, the relation between them is given as $C = \frac{1}{\alpha}$.

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

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 Learning, 20, 273-297 (1995).

NuSVC

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

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

SVR

Given training vectors $x_i \in \mathbb{R}^p$, $i=1, \dots, n$, and a vector $y \in \mathbb{R}^n$ ε -SVR solves the following primal problem:

$$\begin{aligned} & \min_{w, b, \zeta, \zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^n (\zeta_i + \zeta_i^*) \\ & \text{subject to } y_i - w^T \phi(x_i) - b \leq \varepsilon + \zeta_i, \\ & \quad w^T \phi(x_i) + b - y_i \leq \varepsilon + \zeta_i^*, \\ & \quad \zeta_i, \zeta_i^* \geq 0, i = 1, \dots, n \end{aligned}$$

Its dual is

$$\begin{aligned} & \min_{\alpha, \alpha^*} \frac{1}{2} (\alpha - \alpha^*)^T Q (\alpha - \alpha^*) + \varepsilon e^T (\alpha + \alpha^*) - y^T (\alpha - \alpha^*) \\ & \text{subject to } e^T (\alpha - \alpha^*) = 0 \\ & \quad 0 \leq \alpha_i, \alpha_i^* \leq C, i = 1, \dots, n \end{aligned}$$

where e is the vector of all ones, $C > 0$ is the upper bound, Q is an n by n positive semidefinite matrix, $Q_{ij} \equiv K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ .

The decision function is:

$$\sum_{i=1}^n (\alpha_i - \alpha_i^*) K(x_i, x) + \rho$$

These parameters can be accessed through the members `dual_coef_` which holds the difference $\alpha_i - \alpha_i^*$, `support_vectors_` which holds the support vectors, and `intercept_` which holds the independent term ρ

References:

- “A Tutorial on Support Vector Regression”, Alex J. Smola, Bernhard Schölkopf - Statistics and Computing archive Volume 14 Issue 3, August 2004, p. 199-222.

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.

3.1.5 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).

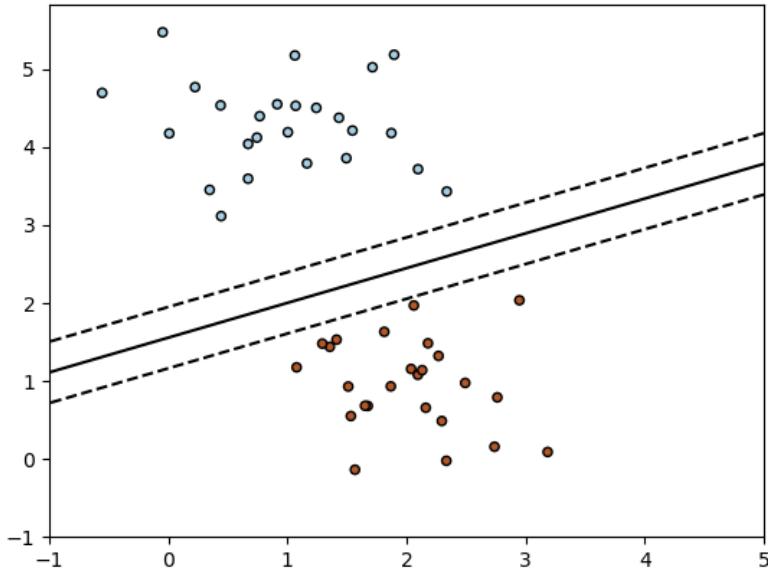
The disadvantages of Stochastic Gradient Descent include:

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

Classification

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

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", max_iter=5)
>>> clf.fit(X, y)
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
              early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
              l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=5,
              n_iter_no_change=5, n_jobs=None, penalty='l2', power_t=0.5,
              random_state=None, shuffle=True, tol=0.001,
              validation_fraction=0.1, 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([[9.9..., 9.9...]])
```

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

```
>>> clf.intercept_
array([-9.9...])
```

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([29.6...])
```

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", max_iter=5).fit(X, y)
>>> clf.predict_proba([[1., 1.]])
array([[0.00..., 0.99...]])
```

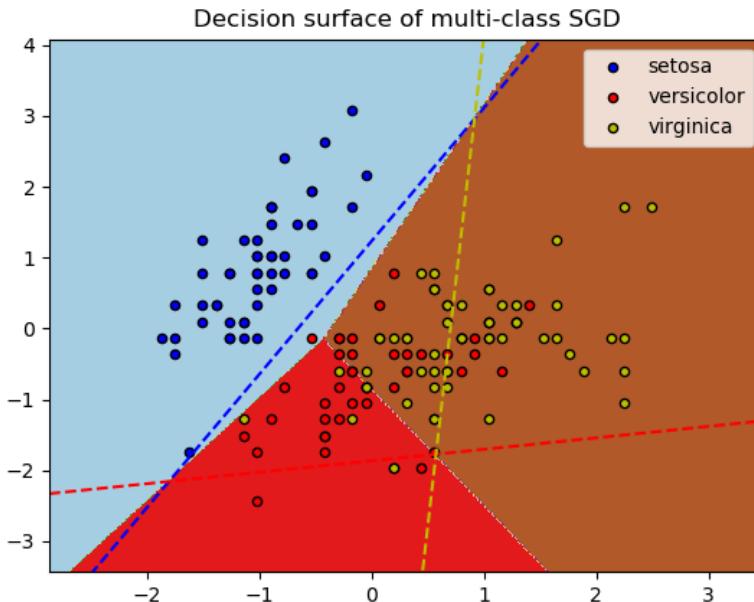
The concrete penalty can be set via the `penalty` parameter. SGD supports the following penalties:

- `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-dimensional 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 docstring of `SGDClassifier.fit` for further information.

Examples:

- `SGD: Maximum margin separating hyperplane`,
- `Plot multi-class SGD on the iris dataset`
- `SGD: Weighted samples`
- `Comparing various online solvers`

- *SVM: Separating hyperplane for unbalanced classes* (See the Note)

`SGDClassifier` supports averaged SGD (ASGD). Averaging can be enabled by setting `average=True`. ASGD works by averaging the coefficients of the plain SGD over each iteration over a sample. When using ASGD the learning rate can be larger and even constant leading on some datasets to a speed up in training time.

For classification with a logistic loss, another variant of SGD with an averaging strategy is available with Stochastic Average Gradient (SAG) algorithm, available as a solver in `LogisticRegression`.

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.

`SGDRegressor` supports averaged SGD as `SGDClassifier`. Averaging can be enabled by setting `average=True`.

For regression with a squared loss and a l2 penalty, another variant of SGD with an averaging strategy is available with Stochastic Average Gradient (SAG) algorithm, available as a solver in `Ridge`.

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*

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 \bar{p} is the average number of non-zero attributes per sample.

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

Stopping criterion

The classes `SGDClassifier` and `SGDRegressor` provide two criteria to stop the algorithm when a given level of convergence is reached:

- With `early_stopping=True`, the input data is split into a training set and a validation set. The model is then fitted on the training set, and the stopping criterion is based on the prediction score computed on the validation set. The size of the validation set can be changed with the parameter `validation_fraction`.
- With `early_stopping=False`, the model is fitted on the entire input data and the stopping criterion is based on the objective function computed on the input data.

In both cases, the criterion is evaluated once by epoch, and the algorithm stops when the criterion does not improve `n_iter_no_change` times in a row. The improvement is evaluated with a tolerance `tol`, and the algorithm stops in any case after a maximum number of iteration `max_iter`.

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 `max_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.
- We found that Averaged SGD works best with a larger number of features and a higher `eta0`

References:

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

Mathematical formulation

Given a set of training examples $(x_1, y_1), \dots, (x_n, y_n)$ where $x_i \in \mathbf{R}^m$ 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

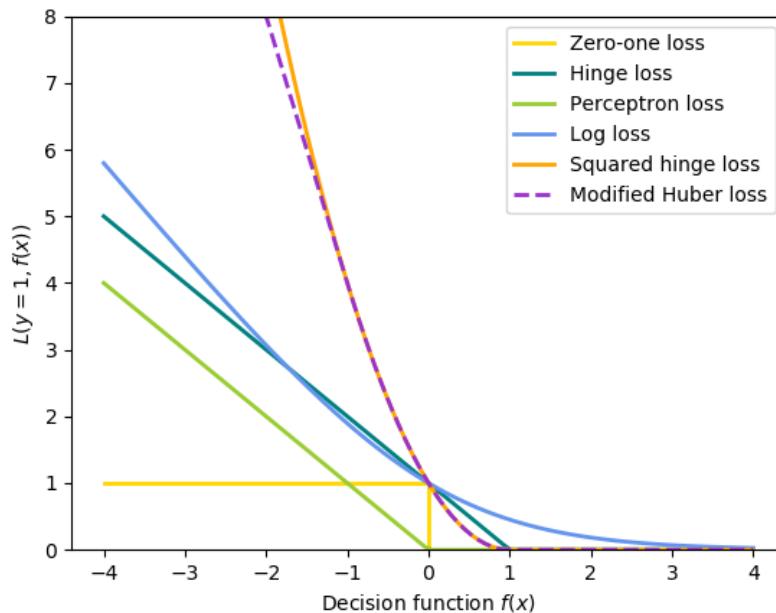
$$E(w, b) = \frac{1}{n} \sum_{i=1}^n L(y_i, f(x_i)) + \alpha R(w)$$

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:

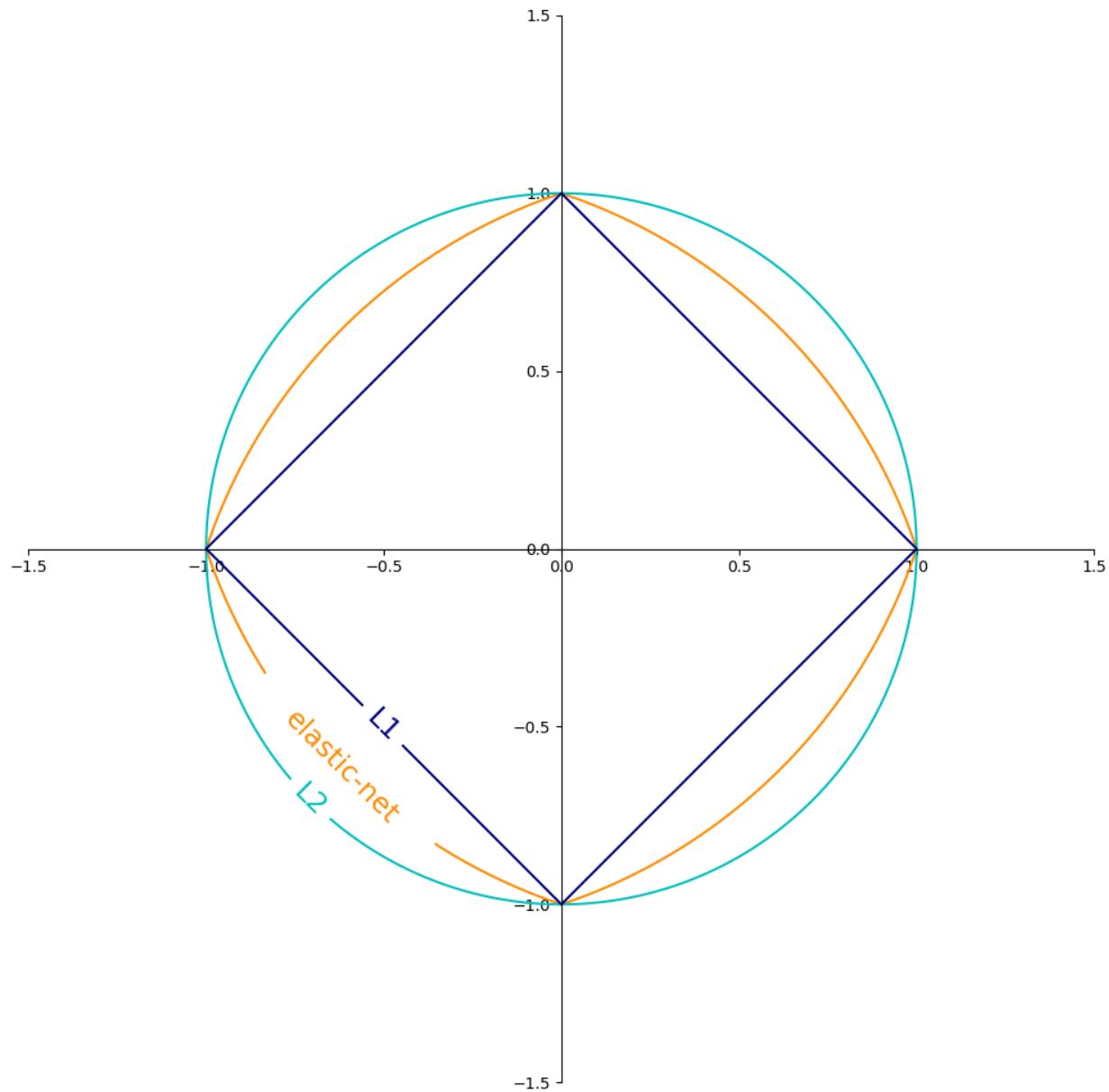
- L2 norm: $R(w) := \frac{1}{2} \sum_{i=1}^n w_i^2$,
- L1 norm: $R(w) := \sum_{i=1}^n |w_i|$, which leads to sparse solutions.
- Elastic Net: $R(w) := \frac{\rho}{2} \sum_{i=1}^n w_i^2 + (1 - \rho) \sum_{i=1}^n |w_i|$, a convex combination of L2 and L1, where ρ is given by `l1_ratio`.

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

SGD

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

The class `SGDClassifier` implements a first-order SGD learning routine. The algorithm iterates over the training



examples and for each example updates the model parameters according to the update rule given by

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

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

$$\eta^{(t)} = \frac{\text{eta}_0}{t^{\text{power_t}}}$$

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.

For an adaptively decreasing learning rate, use `learning_rate='adaptive'` and use `eta0` to specify the starting learning rate. When the stopping criterion is reached, the learning rate is divided by 5, and the algorithm does not stop. The algorithm stops when the learning rate goes below 1e-6.

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.
- “Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent” Xu, Wei

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 ℓ_1 -regularized log-linear models with cumulative penalty” Y. Tsu-ruoka, J. Tsujii, S. Ananiadou - In Proceedings of the AFNLP/ACL ‘09.

3.1.6 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 and 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.

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 $k + 1$ and k have identical distances but different labels, the result will depend on the ordering of the training data.

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.]])
```

The 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`.

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 nearest neighbors searches, including specification of query strategies, distance metrics, etc. For a list of available metrics, see the documentation of the [DistanceMetric](#) class.

Nearest Neighbors Classification

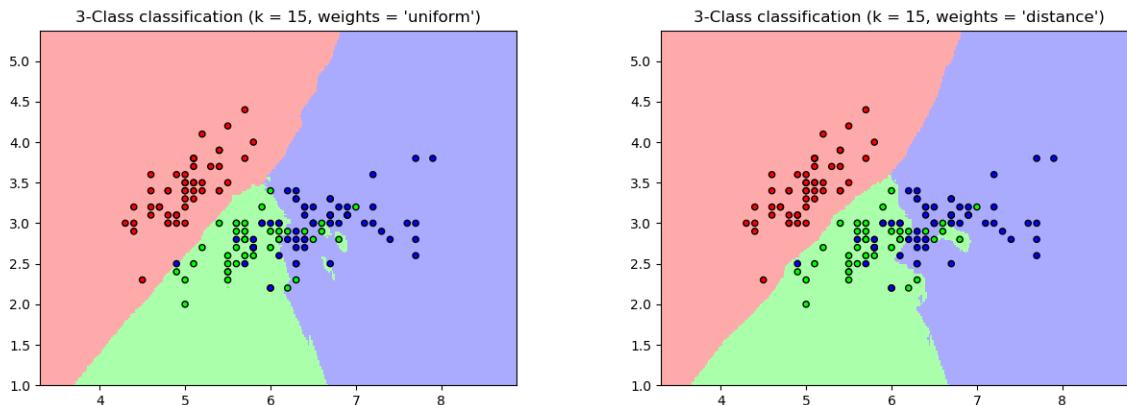
Neighbors-based classification is a type of *instance-based learning* or *non-generalizing learning*: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: 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 most commonly used technique. 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 to compute the weights.



Examples:

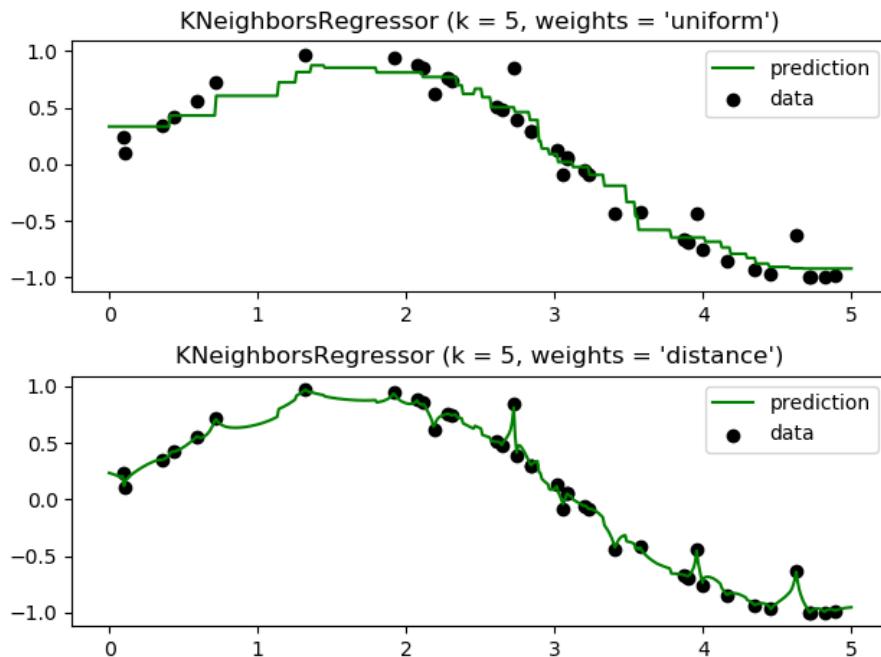
- [Nearest Neighbors Classification](#): an example of classification using nearest neighbors.

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 on 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 multi-output estimators



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

Nearest Neighbor Algorithms

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`.

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 orthotropic 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)

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)

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 '`brute`' if $k \geq N/2$, the input data is sparse, or `effective_metric_` isn't in the `VALID_METRICS` list for either '`kd_tree`' or '`ball_tree`'. Otherwise, it selects the first out of '`kd_tree`' and '`ball_tree`' that has `effective_metric_` in its `VALID_METRICS` list. 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.

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 $1 / \text{leaf_size}$ times the size of the training set.

`leaf_size` is not referenced for brute force queries.

Nearest Centroid Classifier

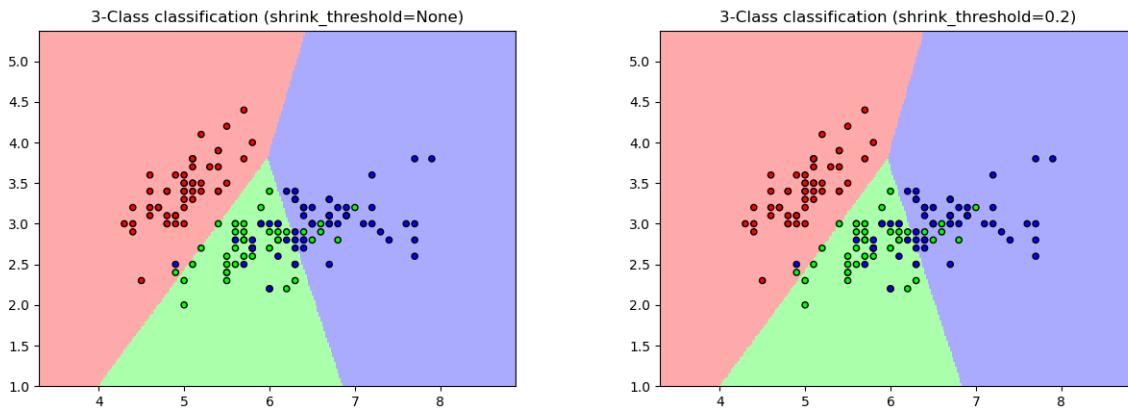
The `NearestCentroid` classifier is a simple algorithm that represents each class by the centroid of its members. In effect, this makes it similar to the label updating phase of the `sklearn.KMeans` algorithm. It also has no parameters to choose, making it a good baseline classifier. It does, however, 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.discriminant_analysis.LinearDiscriminantAnalysis`) and Quadratic Discriminant Analysis (`sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis`) 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]
```

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.

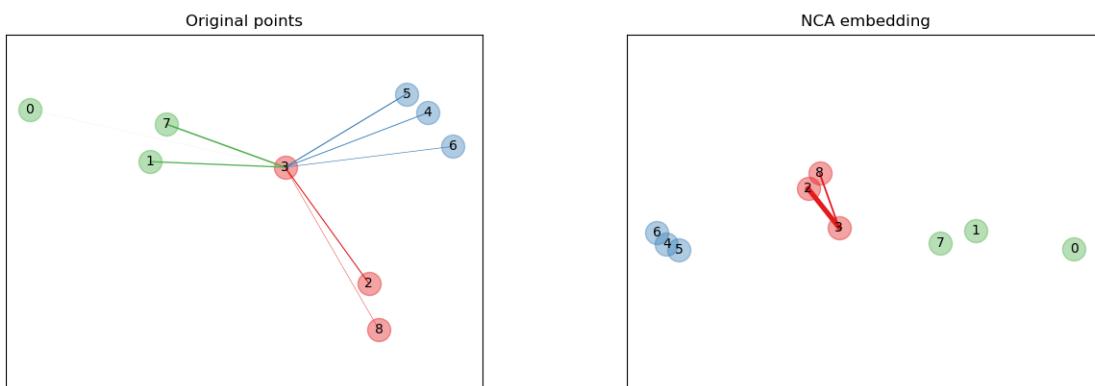


Examples:

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

Neighborhood Components Analysis

Neighborhood Components Analysis (NCA, `NeighborhoodComponentsAnalysis`) is a distance metric learning algorithm which aims to improve the accuracy of nearest neighbors classification compared to the standard Euclidean distance. The algorithm directly maximizes a stochastic variant of the leave-one-out k-nearest neighbors (KNN) score on the training set. It can also learn a low-dimensional linear projection of data that can be used for data visualization and fast classification.



In the above illustrating figure, we consider some points from a randomly generated dataset. We focus on the stochastic KNN classification of point no. 3. The thickness of a link between sample 3 and another point is proportional to their distance, and can be seen as the relative weight (or probability) that a stochastic nearest neighbor prediction rule would assign to this point. In the original space, sample 3 has many stochastic neighbors from various classes, so the right class is not very likely. However, in the projected space learned by NCA, the only stochastic neighbors with non-negligible weight are from the same class as sample 3, guaranteeing that the latter will be well classified. See the [mathematical formulation](#) for more details.

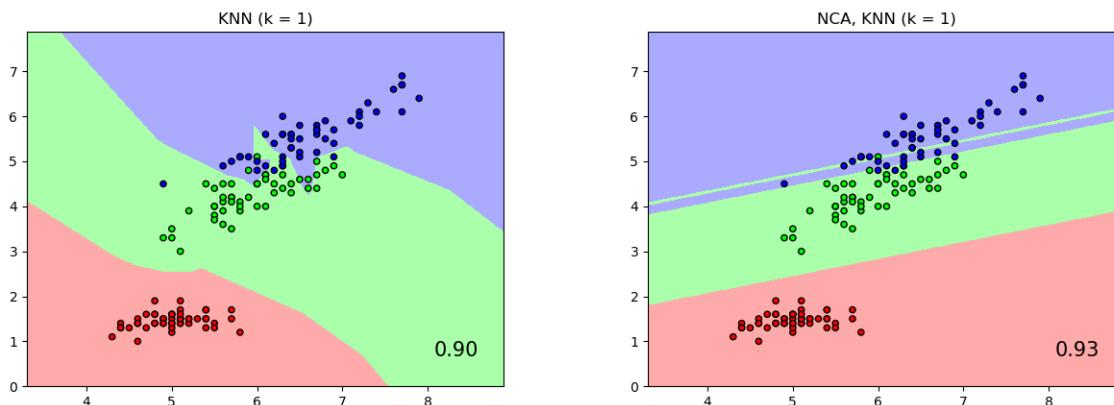
Classification

Combined with a nearest neighbors classifier (`KNeighborsClassifier`), NCA is attractive for classification because it can naturally handle multi-class problems without any increase in the model size, and does not introduce additional parameters that require fine-tuning by the user.

NCA classification has been shown to work well in practice for data sets of varying size and difficulty. In contrast to related methods such as Linear Discriminant Analysis, NCA does not make any assumptions about the class distributions. The nearest neighbor classification can naturally produce highly irregular decision boundaries.

To use this model for classification, one needs to combine a `NeighborhoodComponentsAnalysis` instance that learns the optimal transformation with a `KNeighborsClassifier` instance that performs the classification in the projected space. Here is an example using the two classes:

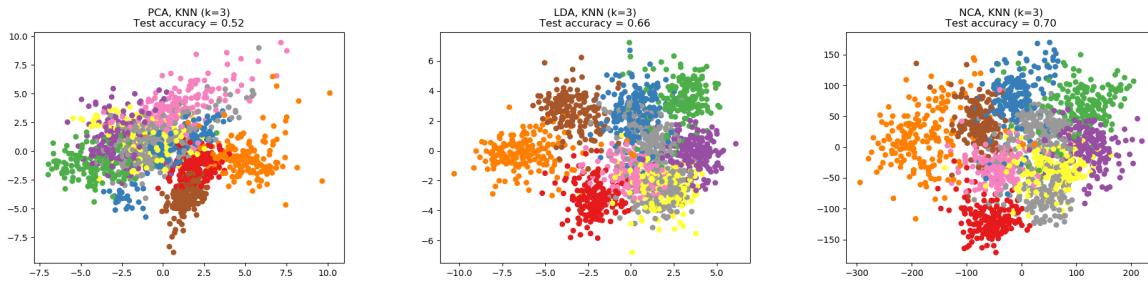
```
>>> from sklearn.neighbors import (NeighborhoodComponentsAnalysis,
... KNeighborsClassifier)
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.pipeline import Pipeline
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
... stratify=y, test_size=0.7, random_state=42)
>>> nca = NeighborhoodComponentsAnalysis(random_state=42)
>>> knn = KNeighborsClassifier(n_neighbors=3)
>>> nca_pipe = Pipeline([('nca', nca), ('knn', knn)])
>>> nca_pipe.fit(X_train, y_train)
Pipeline(...)
>>> print(nca_pipe.score(X_test, y_test))
0.96190476...
```



The plot shows decision boundaries for Nearest Neighbor Classification and Neighborhood Components Analysis classification on the iris dataset, when training and scoring on only two features, for visualisation purposes.

Dimensionality reduction

NCA can be used to perform supervised dimensionality reduction. The input data are projected onto a linear subspace consisting of the directions which minimize the NCA objective. The desired dimensionality can be set using the parameter `n_components`. For instance, the following figure shows a comparison of dimensionality reduction with Principal Component Analysis (`sklearn.decomposition.PCA`), Linear Discriminant Analysis (`sklearn.discriminant_analysis.LinearDiscriminantAnalysis`) and Neighborhood Component Analysis (`NeighborhoodComponentsAnalysis`) on the Digits dataset, a dataset with size $n_{samples} = 1797$ and $n_{features} = 64$. The data set is split into a training and a test set of equal size, then standardized. For evaluation the 3-nearest neighbor classification accuracy is computed on the 2-dimensional projected points found by each method. Each data sample belongs to one of 10 classes.



Examples:

- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

Mathematical formulation

The goal of NCA is to learn an optimal linear transformation matrix of size (`n_components`, `n_features`), which maximises the sum over all samples i of the probability p_i that i is correctly classified, i.e.:

$$\arg \max_L \sum_{i=0}^{N-1} p_i$$

with $N = n_{samples}$ and p_i the probability of sample i being correctly classified according to a stochastic nearest neighbors rule in the learned embedded space:

$$p_i = \sum_{j \in C_i} p_{ij}$$

where C_i is the set of points in the same class as sample i , and p_{ij} is the softmax over Euclidean distances in the embedded space:

$$p_{ij} = \frac{\exp(-\|Lx_i - Lx_j\|^2)}{\sum_{k \neq i} \exp(-\|Lx_i - Lx_k\|^2)}, \quad p_{ii} = 0$$

Mahalanobis distance

NCA can be seen as learning a (squared) Mahalanobis distance metric:

$$\|L(x_i - x_j)\|^2 = (x_i - x_j)^T M (x_i - x_j),$$

where $M = L^T L$ is a symmetric positive semi-definite matrix of size `(n_features, n_features)`.

Implementation

This implementation follows what is explained in the original paper¹. For the optimisation method, it currently uses scipy's L-BFGS-B with a full gradient computation at each iteration, to avoid to tune the learning rate and provide stable learning.

See the examples below and the docstring of `NeighborhoodComponentsAnalysis.fit` for further information.

Complexity

Training

NCA stores a matrix of pairwise distances, taking `n_samples ** 2` memory. Time complexity depends on the number of iterations done by the optimisation algorithm. However, one can set the maximum number of iterations with the argument `max_iter`. For each iteration, time complexity is $O(n_components \times n_samples \times \min(n_samples, n_features))$.

Transform

Here the `transform` operation returns LX^T , therefore its time complexity equals `n_components * n_features * n_samples_test`. There is no added space complexity in the operation.

References:

3.1.7 Gaussian Processes

Gaussian Processes (GP) are a generic supervised learning method designed to solve *regression* and *probabilistic classification* problems.

The advantages of Gaussian processes are:

- The prediction interpolates the observations (at least for regular kernels).
- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals and decide based on those if one should refit (online fitting, adaptive fitting) the prediction in some region of interest.
- Versatile: different `kernels` can be specified. Common kernels are provided, but it is also possible to specify custom kernels.

¹ “Neighbourhood Components Analysis”. *Advances in Neural Information*”, J. Goldberger, G. Hinton, S. Roweis, R. Salakhutdinov, *Advances in Neural Information Processing Systems*, Vol. 17, May 2005, pp. 513-520.

The disadvantages of Gaussian processes include:

- They are not sparse, i.e., they use the whole samples/features information to perform the prediction.
- They lose efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens.

Gaussian Process Regression (GPR)

The `GaussianProcessRegressor` implements Gaussian processes (GP) for regression purposes. For this, the prior of the GP needs to be specified. The prior mean is assumed to be constant and zero (for `normalize_y=False`) or the training data's mean (for `normalize_y=True`). The prior's covariance is specified by passing a `kernel` object. The hyperparameters of the kernel are optimized during fitting of `GaussianProcessRegressor` by maximizing the log-marginal-likelihood (LML) based on the passed `optimizer`. As the LML may have multiple local optima, the optimizer can be started repeatedly by specifying `n_restarts_optimizer`. The first run is always conducted starting from the initial hyperparameter values of the kernel; subsequent runs are conducted from hyperparameter values that have been chosen randomly from the range of allowed values. If the initial hyperparameters should be kept fixed, `None` can be passed as optimizer.

The noise level in the targets can be specified by passing it via the parameter `alpha`, either globally as a scalar or per datapoint. Note that a moderate noise level can also be helpful for dealing with numeric issues during fitting as it is effectively implemented as Tikhonov regularization, i.e., by adding it to the diagonal of the kernel matrix. An alternative to specifying the noise level explicitly is to include a `WhiteKernel` component into the kernel, which can estimate the global noise level from the data (see example below).

The implementation is based on Algorithm 2.1 of [RW2006]. In addition to the API of standard scikit-learn estimators, `GaussianProcessRegressor`:

- allows prediction without prior fitting (based on the GP prior)
- provides an additional method `sample_y(X)`, which evaluates samples drawn from the GPR (prior or posterior) at given inputs
- exposes a method `log_marginal_likelihood(theta)`, which can be used externally for other ways of selecting hyperparameters, e.g., via Markov chain Monte Carlo.

GPR examples

GPR with noise-level estimation

This example illustrates that GPR with a sum-kernel including a `WhiteKernel` can estimate the noise level of data. An illustration of the log-marginal-likelihood (LML) landscape shows that there exist two local maxima of LML.

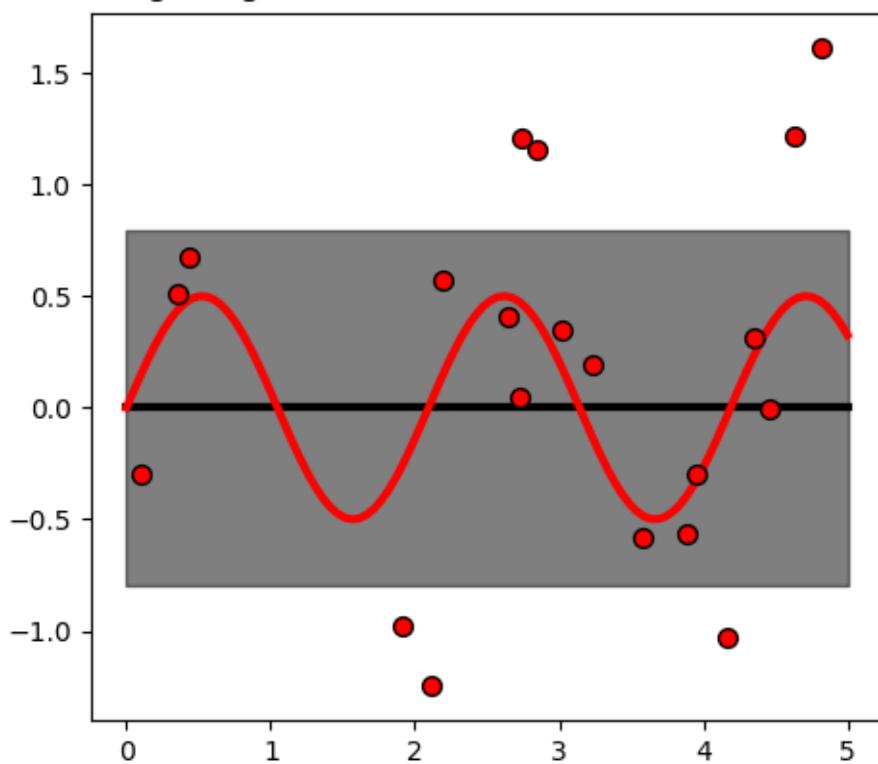
The first corresponds to a model with a high noise level and a large length scale, which explains all variations in the data by noise.

The second one has a smaller noise level and shorter length scale, which explains most of the variation by the noise-free functional relationship. The second model has a higher likelihood; however, depending on the initial value for the hyperparameters, the gradient-based optimization might also converge to the high-noise solution. It is thus important to repeat the optimization several times for different initializations.

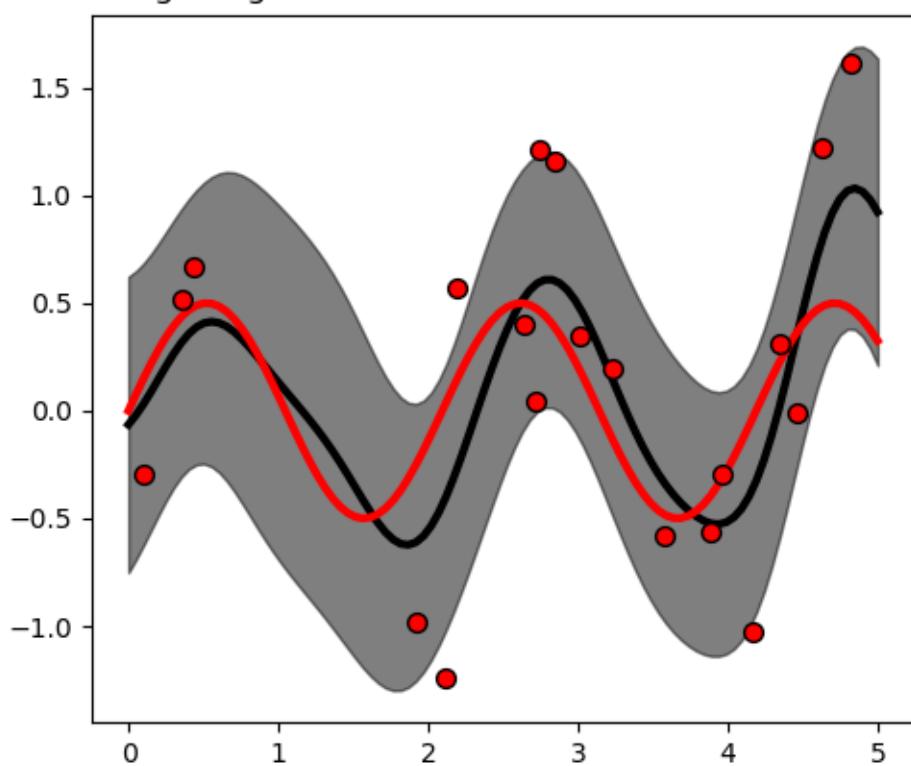
Comparison of GPR and Kernel Ridge Regression

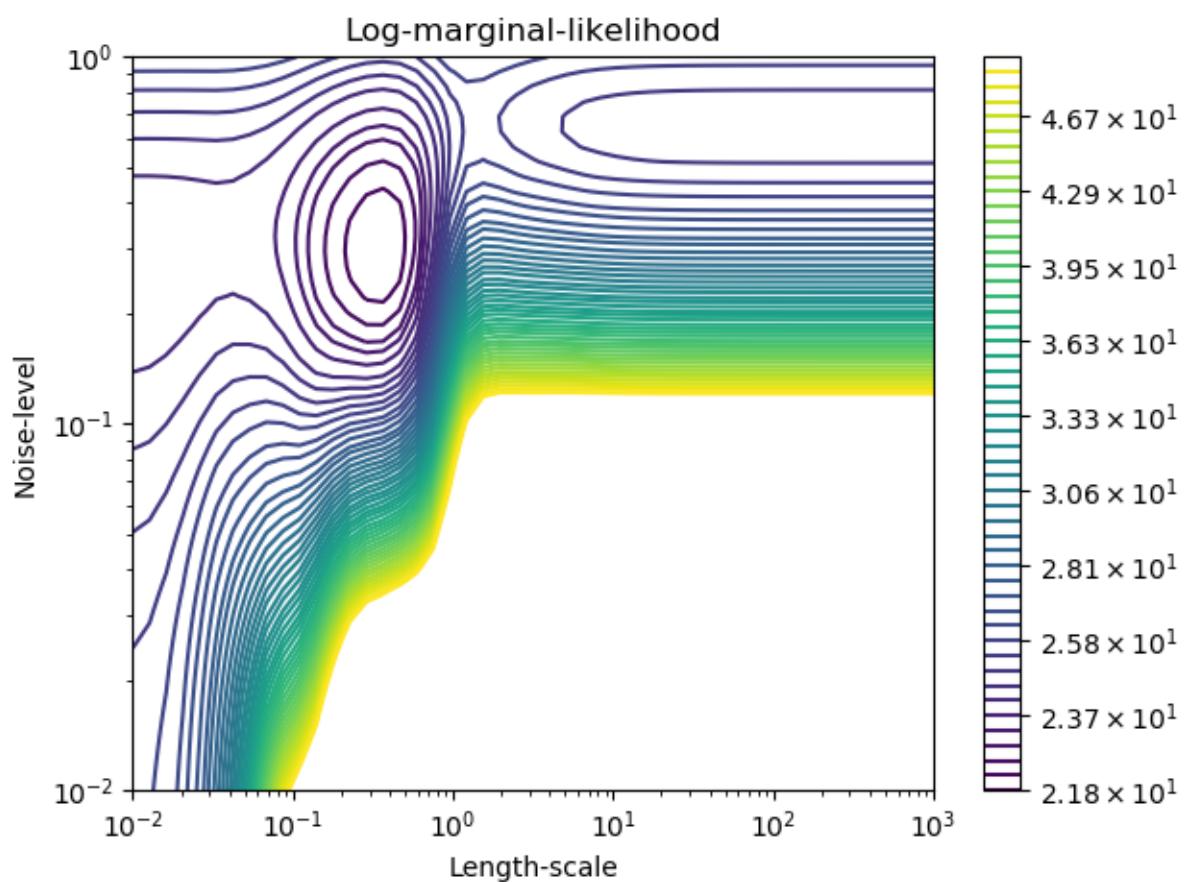
Both kernel ridge regression (KRR) and GPR learn a target function by employing internally the “kernel trick”. KRR learns a linear function in the space induced by the respective kernel which corresponds to a non-linear function in the original space. The linear function in the kernel space is chosen based on the mean-squared error loss with ridge regularization. GPR uses the kernel to define the covariance of a prior distribution over the target functions and uses

```
Initial: 1**2 * RBF(length_scale=100) + WhiteKernel(noise_level=1)
Optimum: 0.00316**2 * RBF(length_scale=109) + WhiteKernel(noise_level=0.6
Log-Marginal-Likelihood: -23.87233736198489
```



```
Initial: 1**2 * RBF(length_scale=1) + WhiteKernel(noise_level=1e-05)
Optimum: 0.64**2 * RBF(length_scale=0.365) + WhiteKernel(noise_level=0.29
Log-Marginal-Likelihood: -21.805090890162035
```

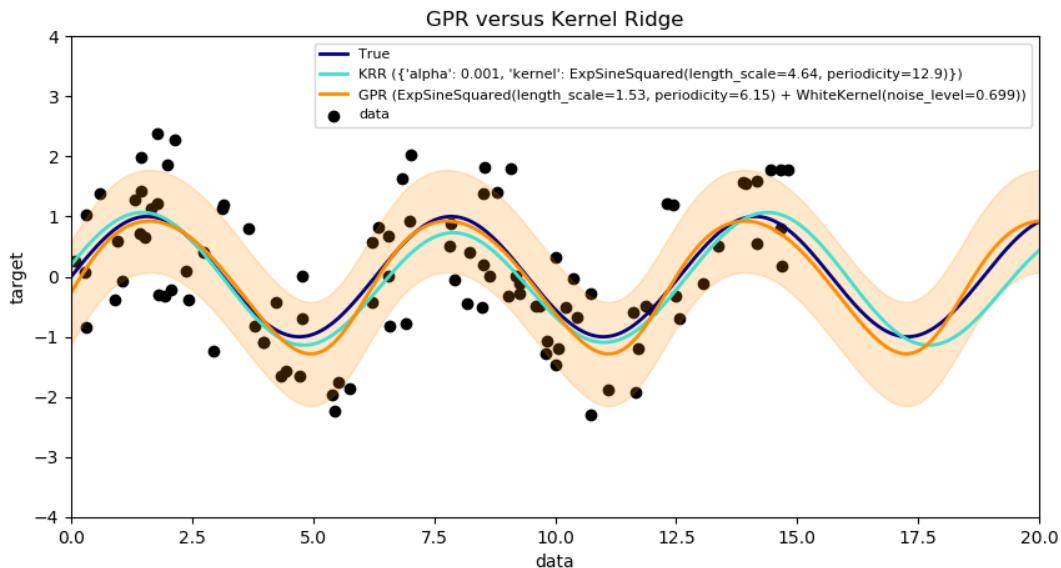




the observed training data to define a likelihood function. Based on Bayes theorem, a (Gaussian) posterior distribution over target functions is defined, whose mean is used for prediction.

A major difference is that GPR can choose the kernel's hyperparameters based on gradient-ascent on the marginal likelihood function while KRR needs to perform a grid search on a cross-validated loss function (mean-squared error loss). A further difference is that GPR learns a generative, probabilistic model of the target function and can thus provide meaningful confidence intervals and posterior samples along with the predictions while KRR only provides predictions.

The following figure illustrates both methods on an artificial dataset, which consists of a sinusoidal target function and strong noise. The figure compares the learned model of KRR and GPR based on a ExpSineSquared kernel, which is suited for learning periodic functions. The kernel's hyperparameters control the smoothness (length_scale) and periodicity of the kernel (periodicity). Moreover, the noise level of the data is learned explicitly by GPR by an additional WhiteKernel component in the kernel and by the regularization parameter alpha of KRR.



The figure shows that both methods learn reasonable models of the target function. GPR correctly identifies the periodicity of the function to be roughly $2 * \pi$ (6.28), while KRR chooses the doubled periodicity $4 * \pi$. Besides that, GPR provides reasonable confidence bounds on the prediction which are not available for KRR. A major difference between the two methods is the time required for fitting and predicting: while fitting KRR is fast in principle, the grid-search for hyperparameter optimization scales exponentially with the number of hyperparameters (“curse of dimensionality”). The gradient-based optimization of the parameters in GPR does not suffer from this exponential scaling and is thus considerably faster on this example with 3-dimensional hyperparameter space. The time for predicting is similar; however, generating the variance of the predictive distribution of GPR takes considerably longer than just predicting the mean.

GPR on Mauna Loa CO2 data

This example is based on Section 5.4.3 of [\[RW2006\]](#). It illustrates an example of complex kernel engineering and hyperparameter optimization using gradient ascent on the log-marginal-likelihood. The data consists of the monthly average atmospheric CO₂ concentrations (in parts per million by volume (ppmv)) collected at the Mauna Loa Observatory in Hawaii, between 1958 and 1997. The objective is to model the CO₂ concentration as a function of the time t.

The kernel is composed of several terms that are responsible for explaining different properties of the signal:

- a long term, smooth rising trend is to be explained by an RBF kernel. The RBF kernel with a large length-scale enforces this component to be smooth; it is not enforced that the trend is rising which leaves this choice to the GP. The specific length-scale and the amplitude are free hyperparameters.
- a seasonal component, which is to be explained by the periodic ExpSineSquared kernel with a fixed periodicity of 1 year. The length-scale of this periodic component, controlling its smoothness, is a free parameter. In order to allow decaying away from exact periodicity, the product with an RBF kernel is taken. The length-scale of this RBF component controls the decay time and is a further free parameter.
- smaller, medium term irregularities are to be explained by a RationalQuadratic kernel component, whose length-scale and alpha parameter, which determines the diffuseness of the length-scales, are to be determined. According to [\[RW2006\]](#), these irregularities can better be explained by a RationalQuadratic than an RBF kernel component, probably because it can accommodate several length-scales.
- a “noise” term, consisting of an RBF kernel contribution, which shall explain the correlated noise components such as local weather phenomena, and a WhiteKernel contribution for the white noise. The relative amplitudes and the RBF’s length scale are further free parameters.

Maximizing the log-marginal-likelihood after subtracting the target’s mean yields the following kernel with an LML of -83.214:

```
34.4**2 * RBF(length_scale=41.8)
+ 3.27**2 * RBF(length_scale=180) * ExpSineSquared(length_scale=1.44,
                                                    periodicity=1)
+ 0.446**2 * RationalQuadratic(alpha=17.7, length_scale=0.957)
+ 0.197**2 * RBF(length_scale=0.138) + WhiteKernel(noise_level=0.0336)
```

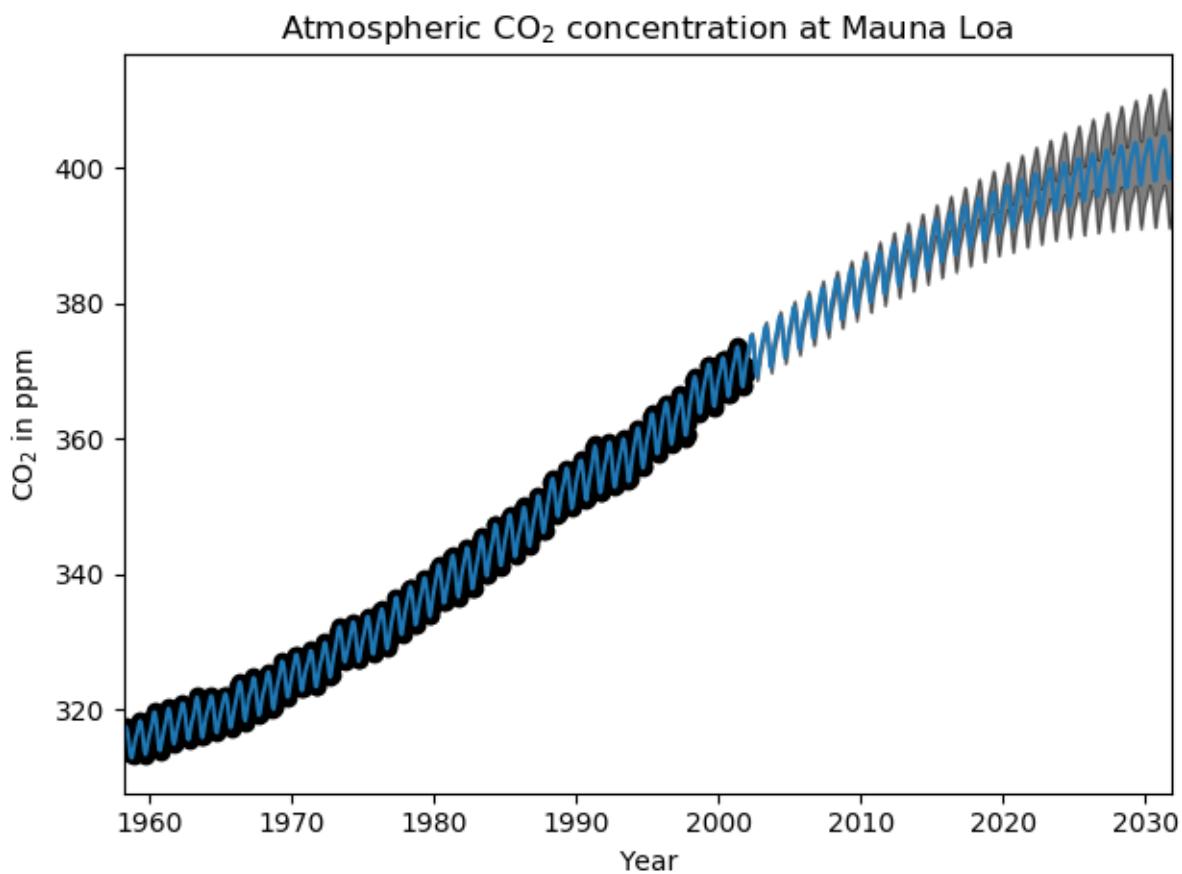
Thus, most of the target signal (34.4ppm) is explained by a long-term rising trend (length-scale 41.8 years). The periodic component has an amplitude of 3.27ppm, a decay time of 180 years and a length-scale of 1.44. The long decay time indicates that we have a locally very close to periodic seasonal component. The correlated noise has an amplitude of 0.197ppm with a length scale of 0.138 years and a white-noise contribution of 0.197ppm. Thus, the overall noise level is very small, indicating that the data can be very well explained by the model. The figure shows also that the model makes very confident predictions until around 2015

Gaussian Process Classification (GPC)

The `GaussianProcessClassifier` implements Gaussian processes (GP) for classification purposes, more specifically for probabilistic classification, where test predictions take the form of class probabilities. `GaussianProcessClassifier` places a GP prior on a latent function f , which is then squashed through a link function to obtain the probabilistic classification. The latent function f is a so-called nuisance function, whose values are not observed and are not relevant by themselves. Its purpose is to allow a convenient formulation of the model, and f is removed (integrated out) during prediction. `GaussianProcessClassifier` implements the logistic link function, for which the integral cannot be computed analytically but is easily approximated in the binary case.

In contrast to the regression setting, the posterior of the latent function f is not Gaussian even for a GP prior since a Gaussian likelihood is inappropriate for discrete class labels. Rather, a non-Gaussian likelihood corresponding to the logistic link function (logit) is used. `GaussianProcessClassifier` approximates the non-Gaussian posterior with a Gaussian based on the Laplace approximation. More details can be found in Chapter 3 of [\[RW2006\]](#).

The GP prior mean is assumed to be zero. The prior’s covariance is specified by passing a `kernel` object. The hyperparameters of the kernel are optimized during fitting of `GaussianProcessRegressor` by maximizing the log-marginal-likelihood (LML) based on the passed `optimizer`. As the LML may have multiple local optima, the optimizer can be started repeatedly by specifying `n_restarts_optimizer`. The first run is always conducted starting from the initial hyperparameter values of the kernel; subsequent runs are conducted from hyperparameter values that have been chosen randomly from the range of allowed values. If the initial hyperparameters should be kept fixed, `None` can be passed as `optimizer`.



`GaussianProcessClassifier` supports multi-class classification by performing either one-versus-rest or one-versus-one based training and prediction. In one-versus-rest, one binary Gaussian process classifier is fitted for each class, which is trained to separate this class from the rest. In “one_vs_one”, one binary Gaussian process classifier is fitted for each pair of classes, which is trained to separate these two classes. The predictions of these binary predictors are combined into multi-class predictions. See the section on [multi-class classification](#) for more details.

In the case of Gaussian process classification, “one_vs_one” might be computationally cheaper since it has to solve many problems involving only a subset of the whole training set rather than fewer problems on the whole dataset. Since Gaussian process classification scales cubically with the size of the dataset, this might be considerably faster. However, note that “one_vs_one” does not support predicting probability estimates but only plain predictions. Moreover, note that `GaussianProcessClassifier` does not (yet) implement a true multi-class Laplace approximation internally, but as discussed above is based on solving several binary classification tasks internally, which are combined using one-versus-rest or one-versus-one.

GPC examples

Probabilistic predictions with GPC

This example illustrates the predicted probability of GPC for an RBF kernel with different choices of the hyperparameters. The first figure shows the predicted probability of GPC with arbitrarily chosen hyperparameters and with the hyperparameters corresponding to the maximum log-marginal-likelihood (LML).

While the hyperparameters chosen by optimizing LML have a considerable larger LML, they perform slightly worse according to the log-loss on test data. The figure shows that this is because they exhibit a steep change of the class probabilities at the class boundaries (which is good) but have predicted probabilities close to 0.5 far away from the class boundaries (which is bad). This undesirable effect is caused by the Laplace approximation used internally by GPC.

The second figure shows the log-marginal-likelihood for different choices of the kernel’s hyperparameters, highlighting the two choices of the hyperparameters used in the first figure by black dots.

Illustration of GPC on the XOR dataset

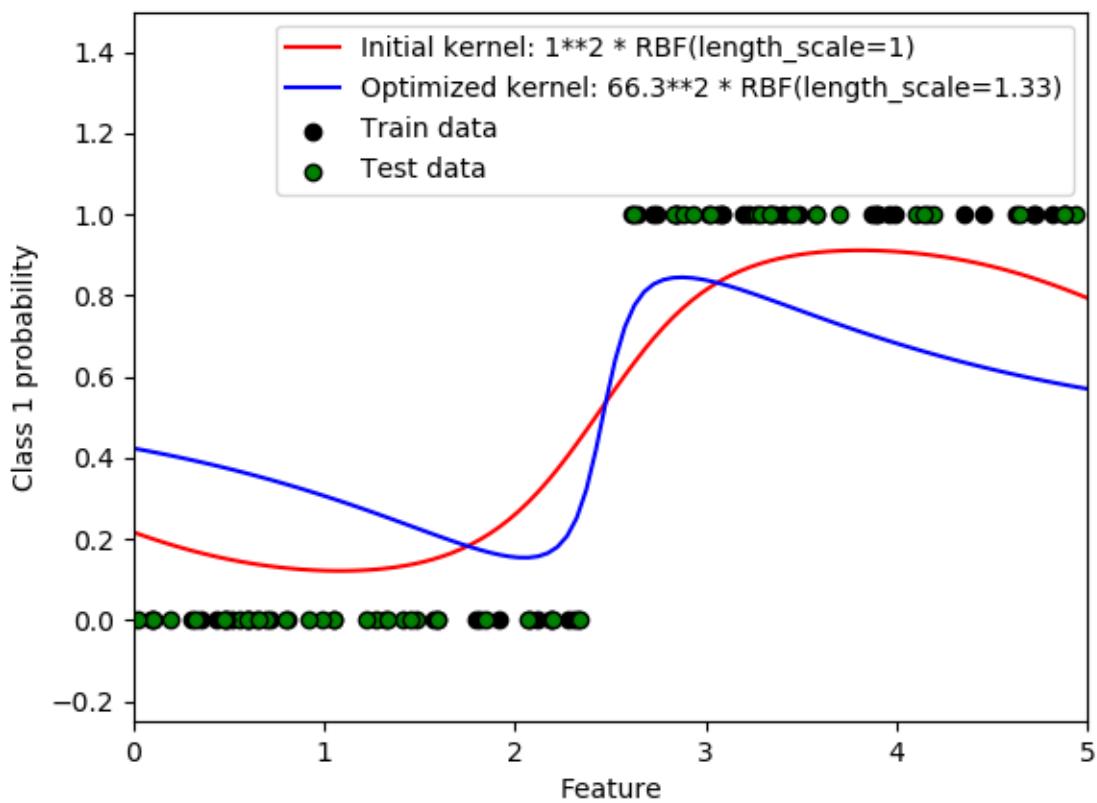
This example illustrates GPC on XOR data. Compared are a stationary, isotropic kernel (`RBF`) and a non-stationary kernel (`DotProduct`). On this particular dataset, the `DotProduct` kernel obtains considerably better results because the class-boundaries are linear and coincide with the coordinate axes. In practice, however, stationary kernels such as `RBF` often obtain better results.

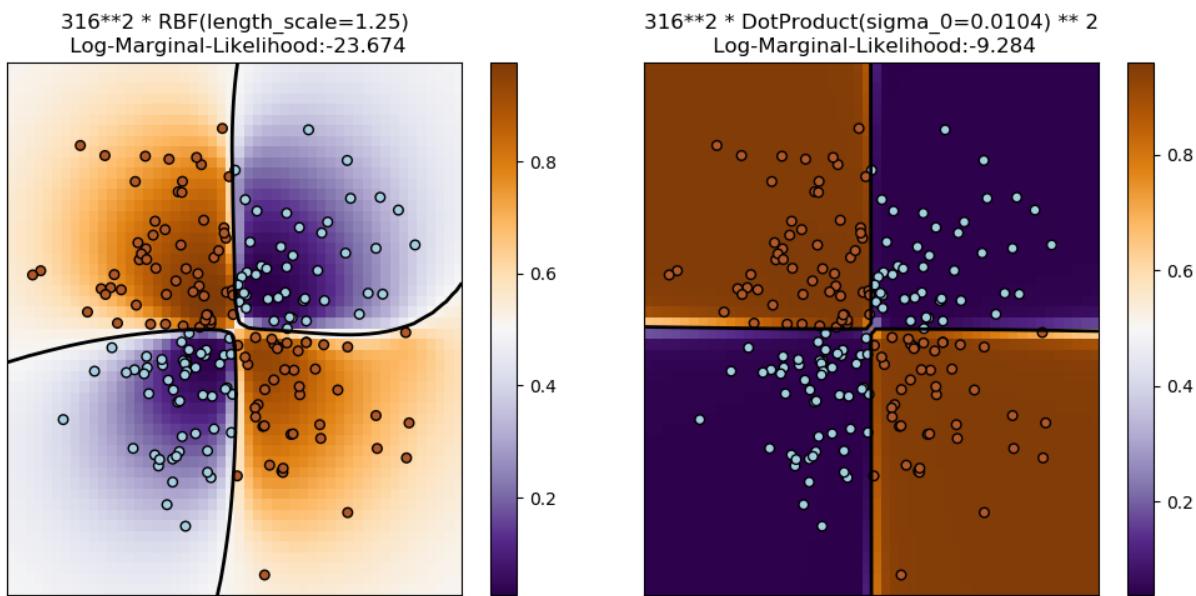
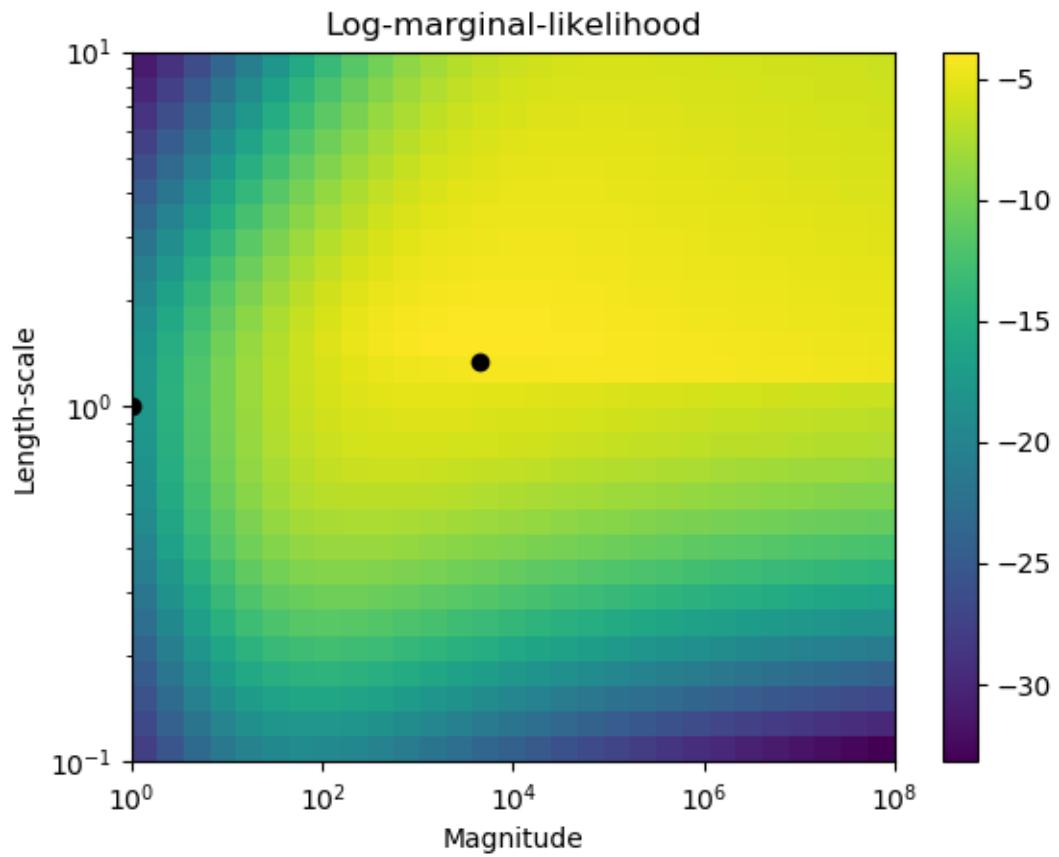
Gaussian process classification (GPC) on iris dataset

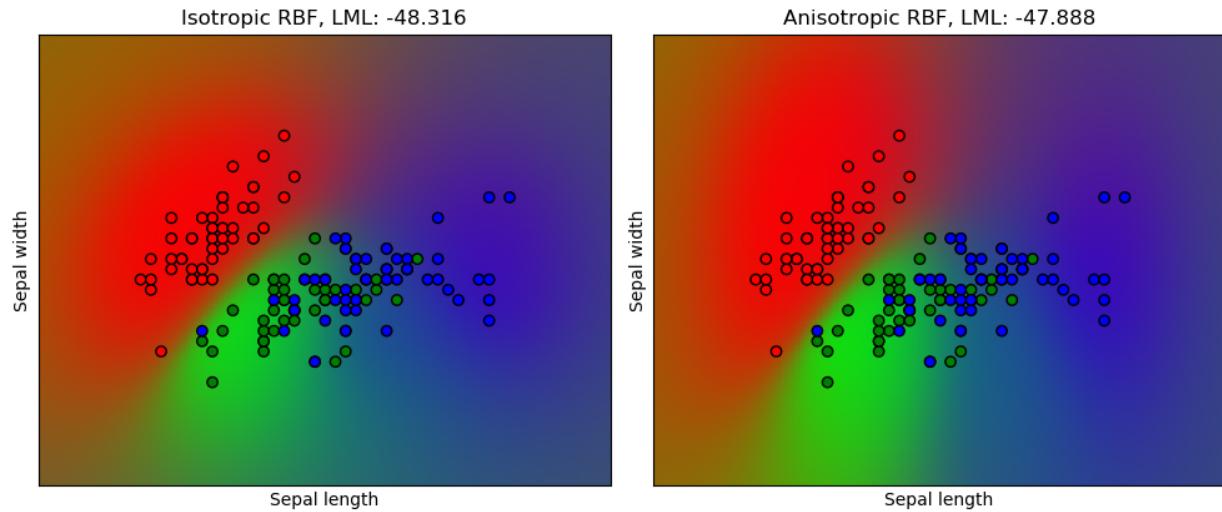
This example illustrates the predicted probability of GPC for an isotropic and anisotropic RBF kernel on a two-dimensional version for the iris-dataset. This illustrates the applicability of GPC to non-binary classification. The anisotropic RBF kernel obtains slightly higher log-marginal-likelihood by assigning different length-scales to the two feature dimensions.

Kernels for Gaussian Processes

Kernels (also called “covariance functions” in the context of GPs) are a crucial ingredient of GPs which determine the shape of prior and posterior of the GP. They encode the assumptions on the function being learned by defining the “similarity” of two datapoints combined with the assumption that similar datapoints should have similar target values. Two categories of kernels can be distinguished: stationary kernels depend only on the distance of two datapoints and not on their absolute values $k(x_i, x_j) = k(d(x_i, x_j))$ and are thus invariant to translations in the input space,







while non-stationary kernels depend also on the specific values of the datapoints. Stationary kernels can further be subdivided into isotropic and anisotropic kernels, where isotropic kernels are also invariant to rotations in the input space. For more details, we refer to Chapter 4 of [RW2006].

Gaussian Process Kernel API

The main usage of a [Kernel](#) is to compute the GP’s covariance between datapoints. For this, the method `__call__` of the kernel can be called. This method can either be used to compute the “auto-covariance” of all pairs of datapoints in a 2d array `X`, or the “cross-covariance” of all combinations of datapoints of a 2d array `X` with datapoints in a 2d array `Y`. The following identity holds true for all kernels `k` (except for the [WhiteKernel](#)): `k(X) == K(X, Y=X)`.

If only the diagonal of the auto-covariance is being used, the method `diag()` of a kernel can be called, which is more computationally efficient than the equivalent call to `__call__`: `np.diag(k(X, X)) == k.diag(X)`

Kernels are parameterized by a vector θ of hyperparameters. These hyperparameters can for instance control lengthscales or periodicity of a kernel (see below). All kernels support computing analytic gradients of the kernel’s auto-covariance with respect to θ via setting `eval_gradient=True` in the `__call__` method. This gradient is used by the Gaussian process (both regressor and classifier) in computing the gradient of the log-marginal-likelihood, which in turn is used to determine the value of θ , which maximizes the log-marginal-likelihood, via gradient ascent. For each hyperparameter, the initial value and the bounds need to be specified when creating an instance of the kernel. The current value of θ can be get and set via the property `theta` of the kernel object. Moreover, the bounds of the hyperparameters can be accessed by the property `bounds` of the kernel. Note that both properties (`theta` and `bounds`) return log-transformed values of the internally used values since those are typically more amenable to gradient-based optimization. The specification of each hyperparameter is stored in the form of an instance of [Hyperparameter](#) in the respective kernel. Note that a kernel using a hyperparameter with name “`x`” must have the attributes `self.x` and `self.x_bounds`.

The abstract base class for all kernels is [Kernel](#). `Kernel` implements a similar interface as `Estimator`, providing the methods `get_params()`, `set_params()`, and `clone()`. This allows setting kernel values also via meta-estimators such as `Pipeline` or `GridSearch`. Note that due to the nested structure of kernels (by applying kernel operators, see below), the names of kernel parameters might become relatively complicated. In general, for a binary kernel operator, parameters of the left operand are prefixed with `k1__` and parameters of the right operand with `k2__`. An additional convenience method is `clone_with_theta(theta)`, which returns a cloned version of the kernel

but with the hyperparameters set to theta. An illustrative example:

```
>>> from sklearn.gaussian_process.kernels import ConstantKernel, RBF
>>> kernel = ConstantKernel(constant_value=1.0, constant_value_bounds=(0.0, 10.0)) *_
<-RBF(length_scale=0.5, length_scale_bounds=(0.0, 10.0)) + RBF(length_scale=2.0,_
<-length_scale_bounds=(0.0, 10.0))
>>> for hyperparameter in kernel.hyperparameters: print(hyperparameter)
Hyperparameter(name='k1__k1__constant_value', value_type='numeric', bounds=array([[ 0. ,_
<, 10.]]), n_elements=1, fixed=False)
Hyperparameter(name='k1__k2__length_scale', value_type='numeric', bounds=array([[ 0.,_
<,10.]]), n_elements=1, fixed=False)
Hyperparameter(name='k2__length_scale', value_type='numeric', bounds=array([[ 0., 10.]]), n_elements=1, fixed=False)
>>> params = kernel.get_params()
>>> for key in sorted(params): print("%s : %s" % (key, params[key]))
k1 : 1**2 * RBF(length_scale=0.5)
k1__k1 : 1**2
k1__k1__constant_value : 1.0
k1__k1__constant_value_bounds : (0.0, 10.0)
k1__k2 : RBF(length_scale=0.5)
k1__k2__length_scale : 0.5
k1__k2__length_scale_bounds : (0.0, 10.0)
k2 : RBF(length_scale=2)
k2__length_scale : 2.0
k2__length_scale_bounds : (0.0, 10.0)
>>> print(kernel.theta) # Note: log-transformed
[ 0.          -0.69314718  0.69314718]
>>> print(kernel.bounds) # Note: log-transformed
[[      -inf 2.30258509]
 [      -inf 2.30258509]
 [      -inf 2.30258509]]
```

All Gaussian process kernels are interoperable with `sklearn.metrics.pairwise` and vice versa: instances of subclasses of `Kernel` can be passed as metric to `pairwise_kernels` from `sklearn.metrics.pairwise`. Moreover, kernel functions from `pairwise` can be used as GP kernels by using the wrapper class `PairwiseKernel`. The only caveat is that the gradient of the hyperparameters is not analytic but numeric and all those kernels support only isotropic distances. The parameter `gamma` is considered to be a hyperparameter and may be optimized. The other kernel parameters are set directly at initialization and are kept fixed.

Basic kernels

The `ConstantKernel` kernel can be used as part of a `Product` kernel where it scales the magnitude of the other factor (kernel) or as part of a `Sum` kernel, where it modifies the mean of the Gaussian process. It depends on a parameter `constant_value`. It is defined as:

$$k(x_i, x_j) = \text{constant_value} \quad \forall x_1, x_2$$

The main use-case of the `WhiteKernel` kernel is as part of a sum-kernel where it explains the noise-component of the signal. Tuning its parameter `noise_level` corresponds to estimating the noise-level. It is defined as:

$$k(x_i, x_j) = \text{noise_level} \text{ if } x_i == x_j \text{ else } 0$$

Kernel operators

Kernel operators take one or two base kernels and combine them into a new kernel. The `Sum` kernel takes two kernels `k1` and `k2` and combines them via $k_{\text{sum}}(X, Y) = k1(X, Y) + k2(X, Y)$. The `Product` kernel takes two kernels `k1`

and $k2$ and combines them via $k_{product}(X, Y) = k1(X, Y) * k2(X, Y)$. The *Exponentiation* kernel takes one base kernel and a scalar parameter $exponent$ and combines them via $k_{exp}(X, Y) = k(X, Y)^{exponent}$.

Radial-basis function (RBF) kernel

The *RBF* kernel is a stationary kernel. It is also known as the “squared exponential” kernel. It is parameterized by a length-scale parameter $l > 0$, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs x (anisotropic variant of the kernel). The kernel is given by:

$$k(x_i, x_j) = \exp\left(-\frac{1}{2}d(x_i/l, x_j/l)^2\right)$$

This kernel is infinitely differentiable, which implies that GPs with this kernel as covariance function have mean square derivatives of all orders, and are thus very smooth. The prior and posterior of a GP resulting from an RBF kernel are shown in the following figure:

Matérn kernel

The *Matern* kernel is a stationary kernel and a generalization of the *RBF* kernel. It has an additional parameter ν which controls the smoothness of the resulting function. It is parameterized by a length-scale parameter $l > 0$, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs x (anisotropic variant of the kernel). The kernel is given by:

$$k(x_i, x_j) = \sigma^2 \frac{1}{\Gamma(\nu)2^{\nu-1}} \left(\gamma \sqrt{2\nu} d(x_i/l, x_j/l) \right)^\nu K_\nu \left(\gamma \sqrt{2\nu} d(x_i/l, x_j/l) \right),$$

As $\nu \rightarrow \infty$, the Matérn kernel converges to the RBF kernel. When $\nu = 1/2$, the Matérn kernel becomes identical to the absolute exponential kernel, i.e.,

$$k(x_i, x_j) = \sigma^2 \exp\left(-\gamma d(x_i/l, x_j/l)\right) \quad \nu = \frac{1}{2}$$

In particular, $\nu = 3/2$:

$$k(x_i, x_j) = \sigma^2 \left(1 + \gamma \sqrt{3} d(x_i/l, x_j/l) \right) \exp\left(-\gamma \sqrt{3} d(x_i/l, x_j/l)\right) \quad \nu = \frac{3}{2}$$

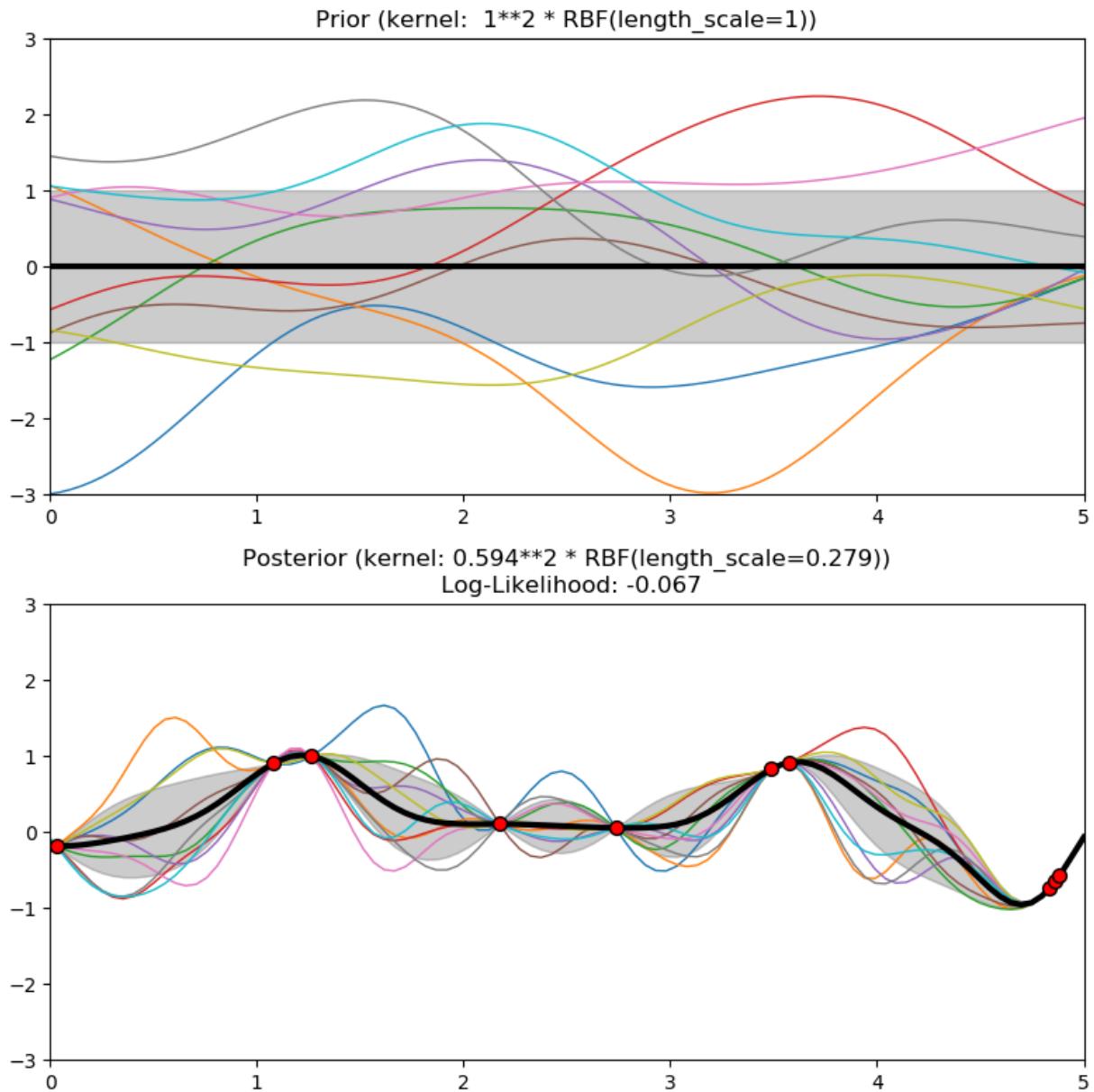
and $\nu = 5/2$:

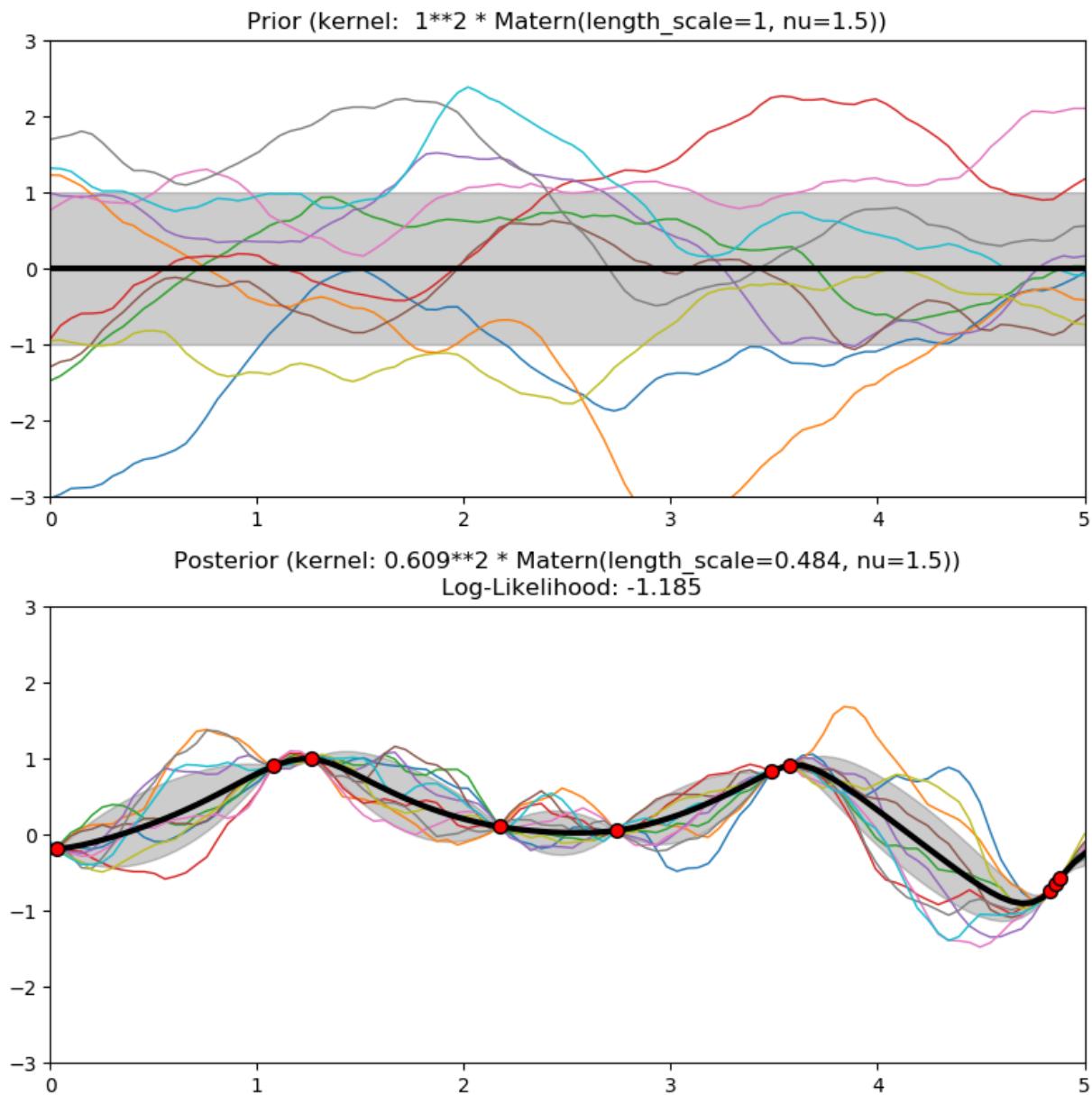
$$k(x_i, x_j) = \sigma^2 \left(1 + \gamma \sqrt{5} d(x_i/l, x_j/l) + \frac{5}{3} \gamma^2 d(x_i/l, x_j/l)^2 \right) \exp\left(-\gamma \sqrt{5} d(x_i/l, x_j/l)\right) \quad \nu = \frac{5}{2}$$

are popular choices for learning functions that are not infinitely differentiable (as assumed by the RBF kernel) but at least once ($\nu = 3/2$) or twice differentiable ($\nu = 5/2$).

The flexibility of controlling the smoothness of the learned function via ν allows adapting to the properties of the true underlying functional relation. The prior and posterior of a GP resulting from a Matérn kernel are shown in the following figure:

See [RW2006], pp84 for further details regarding the different variants of the Matérn kernel.



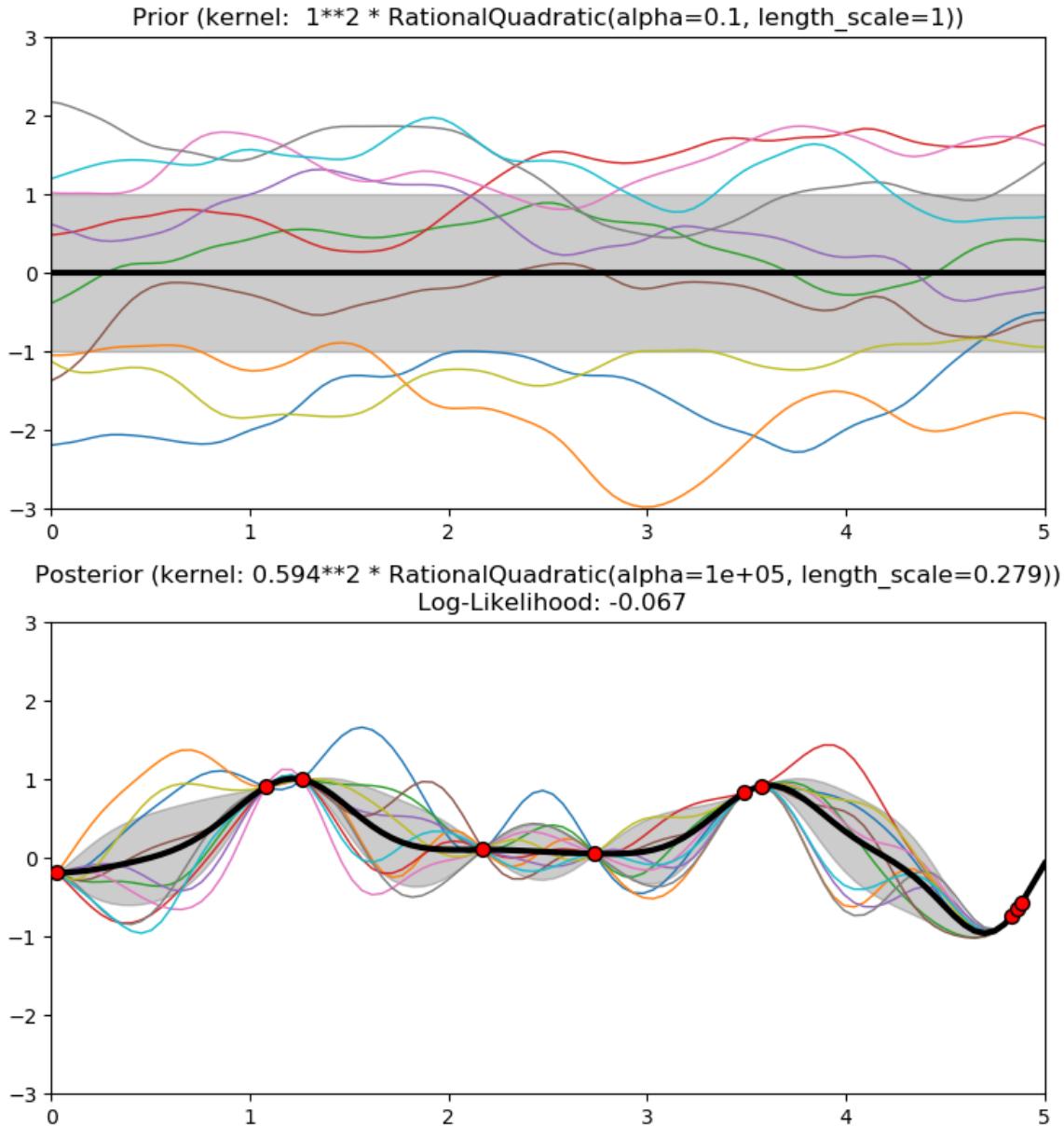


Rational quadratic kernel

The `RationalQuadratic` kernel can be seen as a scale mixture (an infinite sum) of `RBF` kernels with different characteristic length-scales. It is parameterized by a length-scale parameter $l > 0$ and a scale mixture parameter $\alpha > 0$. Only the isotropic variant where l is a scalar is supported at the moment. The kernel is given by:

$$k(x_i, x_j) = \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha}$$

The prior and posterior of a GP resulting from a `RationalQuadratic` kernel are shown in the following figure:

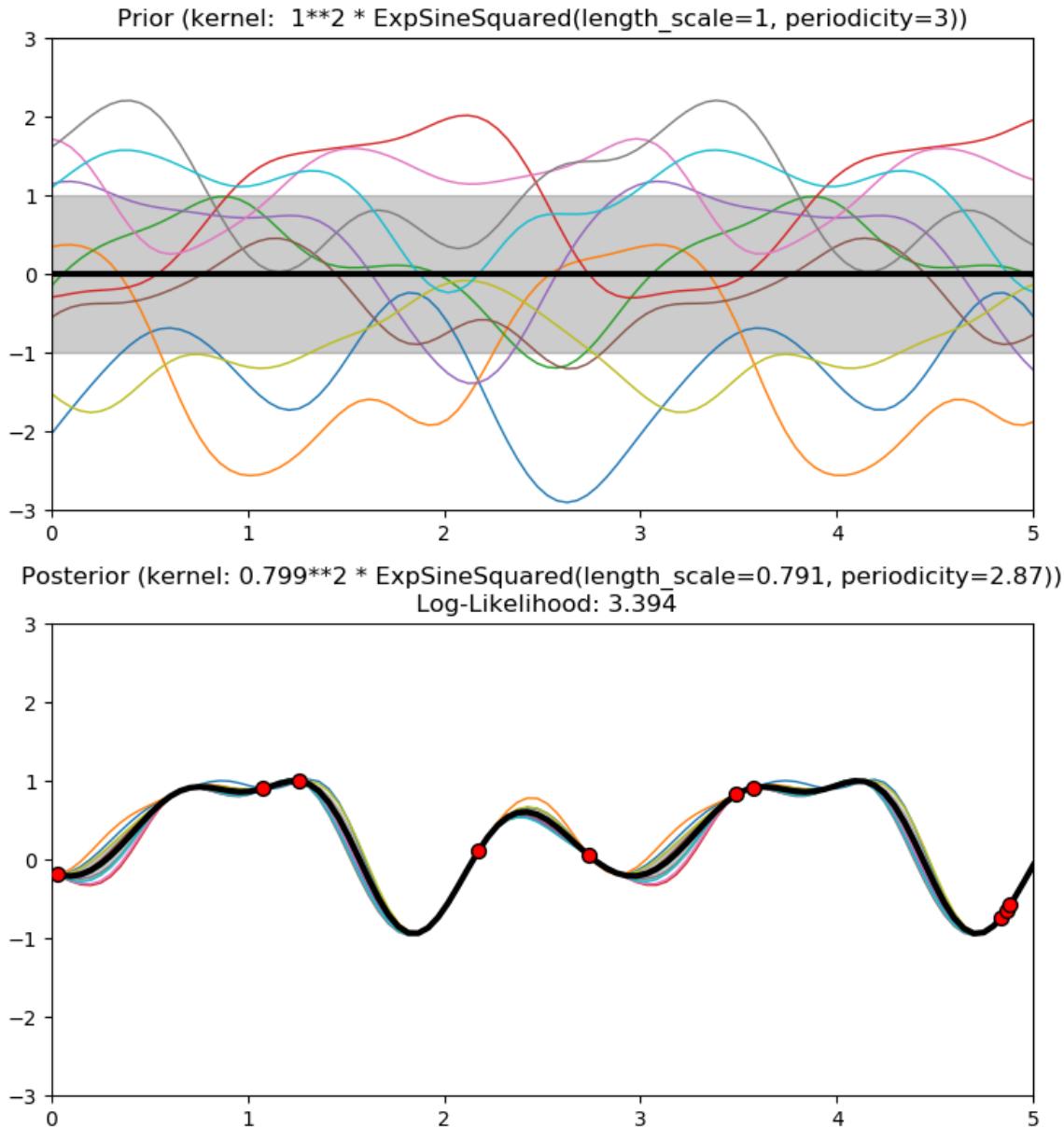


Exp-Sine-Squared kernel

The `ExpSineSquared` kernel allows modeling periodic functions. It is parameterized by a length-scale parameter $l > 0$ and a periodicity parameter $p > 0$. Only the isotropic variant where l is a scalar is supported at the moment. The kernel is given by:

$$k(x_i, x_j) = \exp\left(-2(\sin(\pi/p * d(x_i, x_j))/l)^2\right)$$

The prior and posterior of a GP resulting from an ExpSineSquared kernel are shown in the following figure:

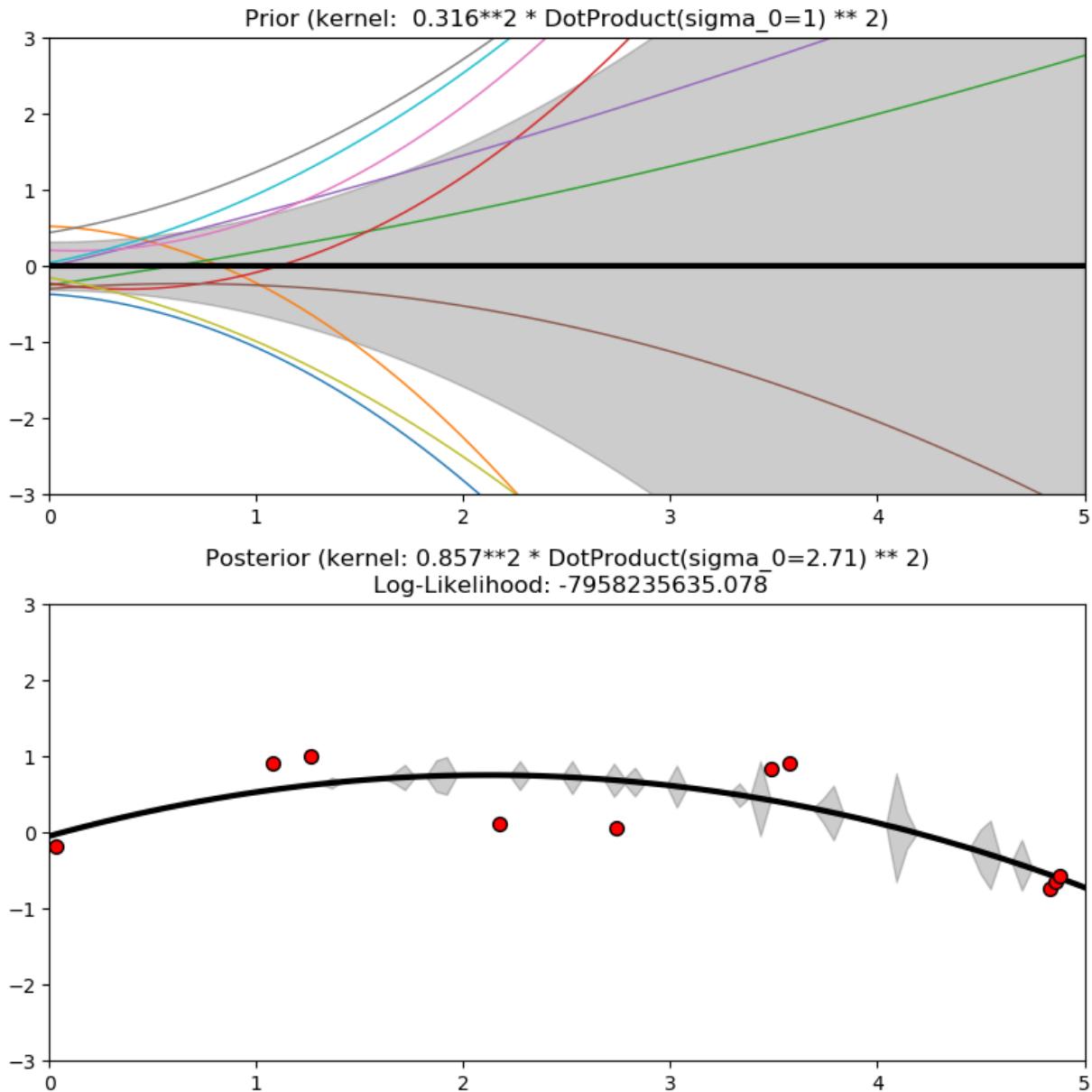


Dot-Product kernel

The [DotProduct](#) kernel is non-stationary and can be obtained from linear regression by putting $N(0, 1)$ priors on the coefficients of $x_d (d = 1, \dots, D)$ and a prior of $N(0, \sigma_0^2)$ on the bias. The [DotProduct](#) kernel is invariant to a rotation of the coordinates about the origin, but not translations. It is parameterized by a parameter σ_0^2 . For $\sigma_0^2 = 0$, the kernel is called the homogeneous linear kernel, otherwise it is inhomogeneous. The kernel is given by

$$k(x_i, x_j) = \sigma_0^2 + x_i \cdot x_j$$

The [DotProduct](#) kernel is commonly combined with exponentiation. An example with exponent 2 is shown in the following figure:

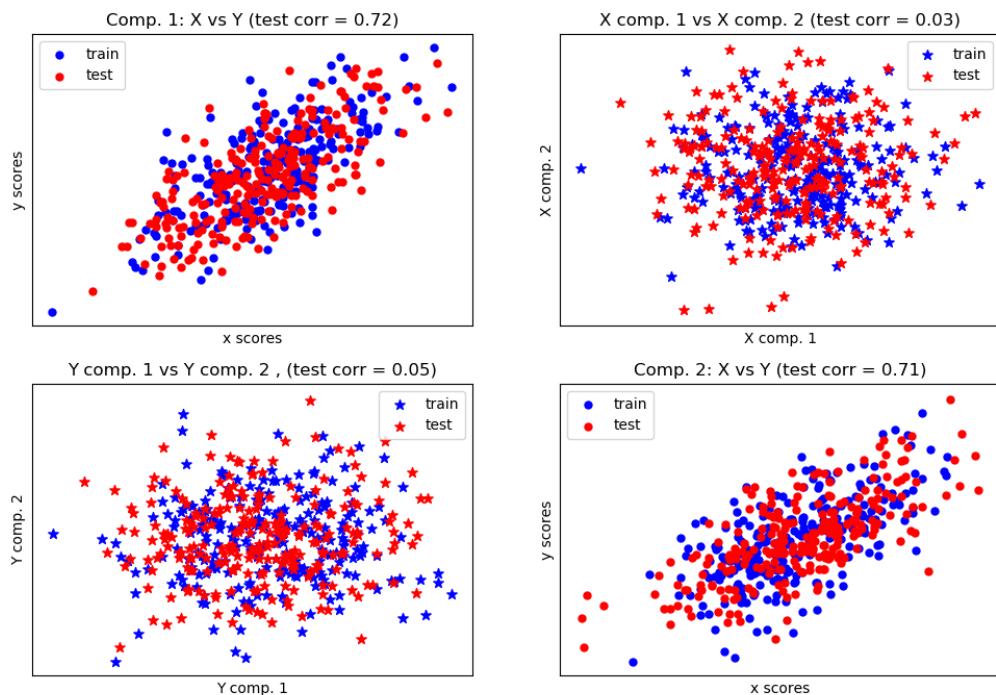


References

3.1.8 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*

3.1.9 Naive Bayes

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

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

Using the naive conditional 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

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

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

$$\begin{aligned} P(y | x_1, \dots, x_n) &\propto P(y) \prod_{i=1}^n P(x_i | y) \\ &\quad \Downarrow \\ \hat{y} &= \arg \max_y P(y) \prod_{i=1}^n P(x_i | y), \end{aligned}$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i | y)$; 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 $P(x_i | y)$.

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.

Gaussian Naive Bayes

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

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

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
```

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 $\theta_y = (\theta_{y1}, \dots, \theta_{yn})$ 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 $P(x_i | y)$ 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 $N_y = \sum_{i=1}^n N_{yi}$ 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.

Complement Naive Bayes

ComplementNB implements the complement naive Bayes (CNB) algorithm. CNB is an adaptation of the standard multinomial naive Bayes (MNB) algorithm that is particularly suited for imbalanced data sets. Specifically, CNB uses statistics from the *complement* of each class to compute the model's weights. The inventors of CNB show empirically that the parameter estimates for CNB are more stable than those for MNB. Further, CNB regularly outperforms MNB (often by a considerable margin) on text classification tasks. The procedure for calculating the weights is as follows:

$$\begin{aligned} \hat{\theta}_{ci} &= \frac{\alpha_i + \sum_{j:y_j \neq c} d_{ij}}{\alpha + \sum_{j:y_j \neq c} \sum_k d_{kj}} \\ w_{ci} &= \log \hat{\theta}_{ci} \\ w_{ci} &= \frac{w_{ci}}{\sum_j |w_{cj}|} \end{aligned}$$

where the summations are over all documents j not in class c , d_{ij} is either the count or tf-idf value of term i in document j , α_i is a smoothing hyperparameter like that found in MNB, and $\alpha = \sum_i \alpha_i$. The second normalization

addresses the tendency for longer documents to dominate parameter estimates in MNB. The classification rule is:

$$\hat{c} = \arg \min_c \sum_i t_i w_{ci}$$

i.e., a document is assigned to the class that is the *poorest* complement match.

References:

- Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). Tackling the poor assumptions of naive bayes text classifiers. In ICML (Vol. 3, pp. 616-623).

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

$$P(x_i | y) = P(i | y)x_i + (1 - P(i | y))(1 - x_i)$$

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. Palouras (2006). Spam filtering with Naive Bayes – Which Naive Bayes? 3rd Conf. on Email and Anti-Spam (CEAS).

Out-of-core naive Bayes model fitting

Naive Bayes models can be used to tackle large scale classification problems for which the full training set might not fit in memory. To handle this case, `MultinomialNB`, `BernoulliNB`, and `GaussianNB` 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](#). All naive Bayes classifiers support sample weighting.

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

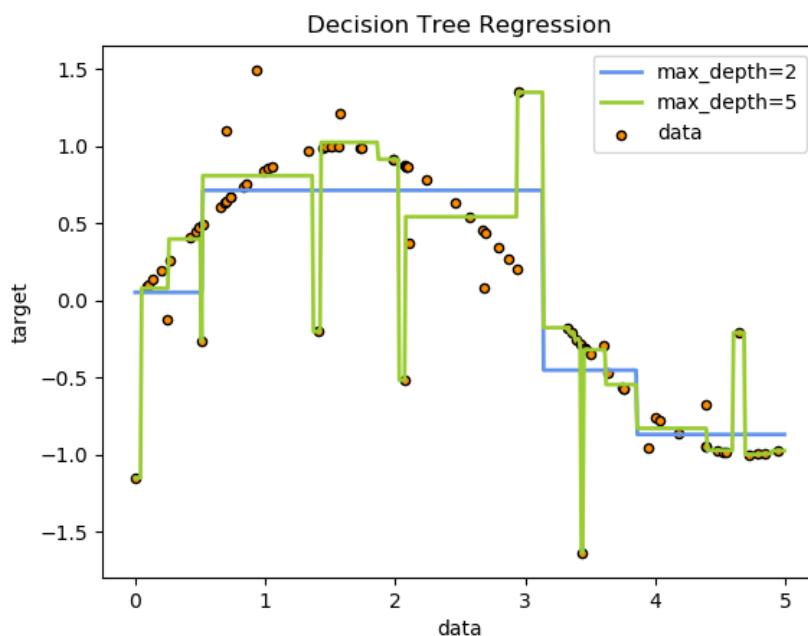
For an overview of available strategies in scikit-learn, see also the [out-of-core learning](#) documentation.

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.

3.1.10 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.

Classification

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

As with other classifiers, *DecisionTreeClassifier* takes as input two arrays: an array X, sparse or dense, 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 the class of samples:

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

Alternatively, the probability of each class can be predicted, which is the fraction of training samples of the same class in a leaf:

```
>>> clf.predict_proba([[2., 2.]])
array([[0., 1.]])
```

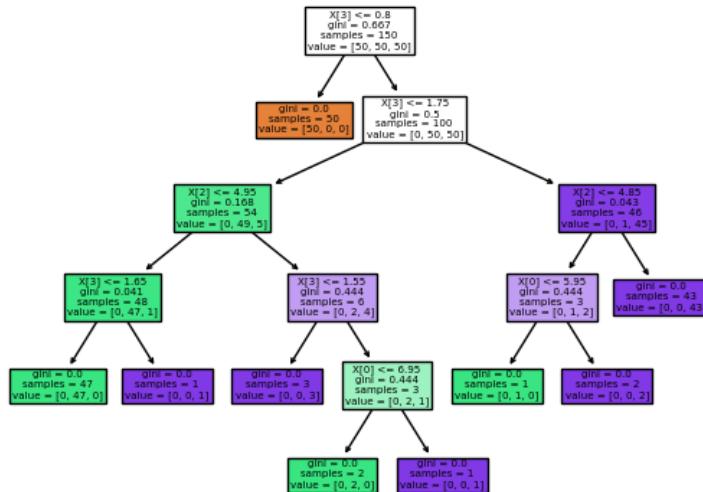
DecisionTreeClassifier is capable of both binary (where the labels are [-1, 1]) classification and multiclass (where the labels are [0, ..., K-1]) classification.

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, you can plot the tree with the `plot_tree` function:

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



We can also export the tree in `Graphviz` format using the `export_graphviz` exporter. If you use the `conda` package manager, the `graphviz` binaries

and the python package can be installed with

```
conda install python-graphviz
```

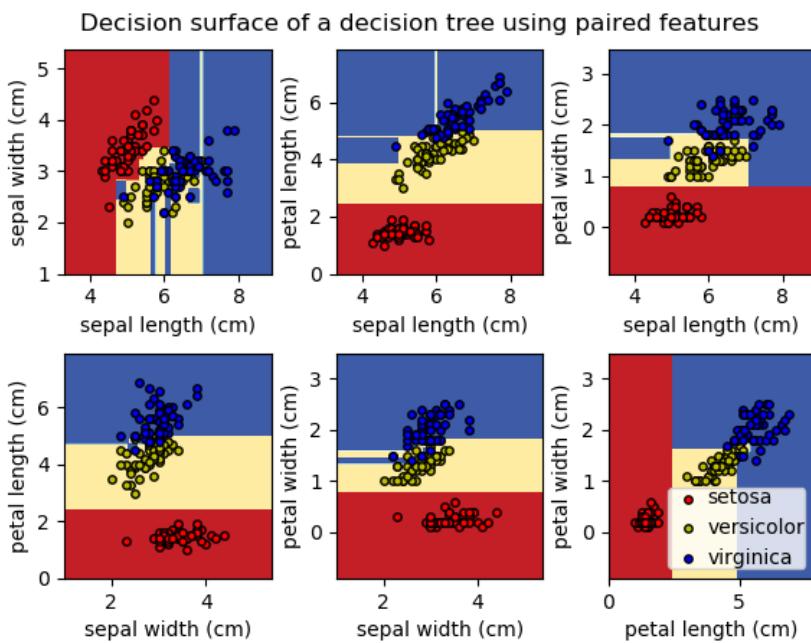
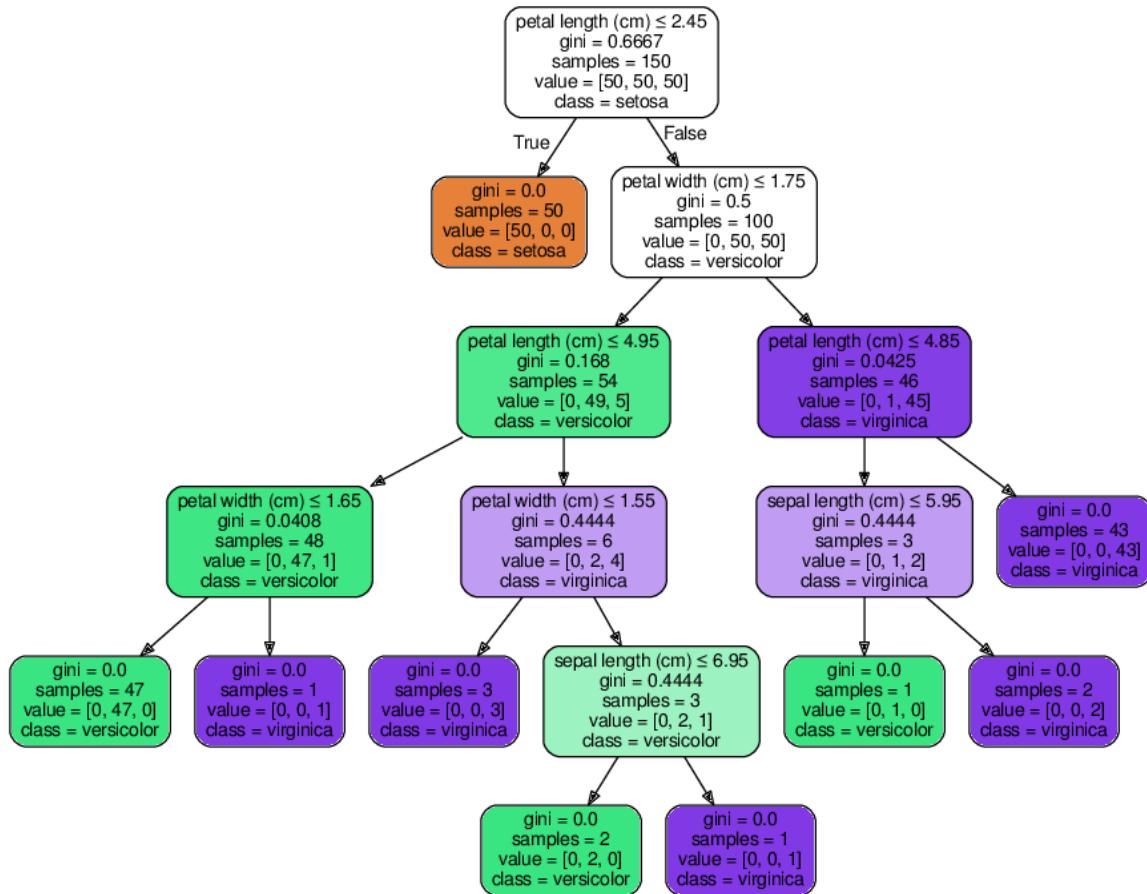
Alternatively binaries for `graphviz` can be downloaded from the `graphviz` project homepage, and the Python wrapper installed from `pypi` with `pip install graphviz`.

Below is an example `graphviz` export of the above tree trained on the entire `iris` dataset; the results are saved in an output file `iris.pdf`:

```
>>> import graphviz
>>> dot_data = tree.export_graphviz(clf, out_file=None)
>>> graph = graphviz.Source(dot_data)
>>> graph.render("iris")
```

The `export_graphviz` exporter also supports a variety of aesthetic options, including coloring nodes by their class (or value for regression) and using explicit variable and class names if desired. Jupyter notebooks also render these plots inline automatically:

```
>>> dot_data = tree.export_graphviz(clf, out_file=None,
...                                 feature_names=iris.feature_names,
...                                 class_names=iris.target_names,
...                                 filled=True, rounded=True,
...                                 special_characters=True)
>>> graph = graphviz.Source(dot_data)
>>> graph
```



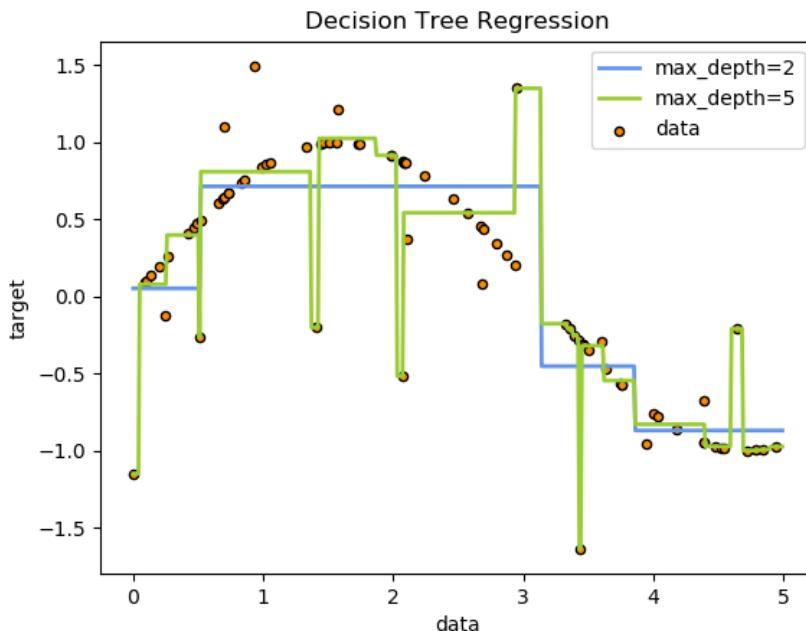
Alternatively, the tree can also be exported in textual format with the function `export_text`. This method doesn't require the installation of external libraries and is more compact:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.tree.export import export_text
>>> iris = load_iris()
>>> X = iris['data']
>>> y = iris['target']
>>> decision_tree = DecisionTreeClassifier(random_state=0, max_depth=2)
>>> decision_tree = decision_tree.fit(X, y)
>>> r = export_text(decision_tree, feature_names=iris['feature_names'])
>>> print(r)
|--- petal width (cm) <= 0.80
|   |--- class: 0
|--- petal width (cm) >  0.80
|   |--- petal width (cm) <= 1.75
|   |   |--- class: 1
|   |--- petal width (cm) >  1.75
|   |   |--- class: 2
```

Examples:

- Plot the decision surface of a decision tree on the iris dataset
- Understanding the decision tree structure

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](#)

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:

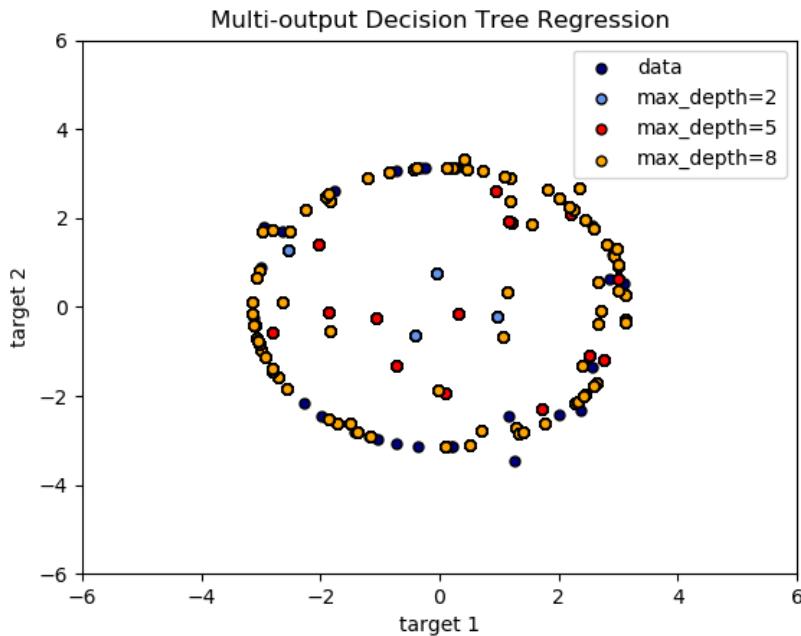
- Output n_output values upon predict;
- Output a list of n_output arrays of class probabilities upon predict_proba.

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

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 $O(n_{features} n_{samples}^2 \log(n_{samples}))$.

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. Presorting the feature over all relevant samples, and retaining a running label count, will reduce the complexity at each node to $O(n_{features} \log(n_{samples}))$, which results in a total cost of $O(n_{features} n_{samples} \log(n_{samples}))$. This is an option for all tree based algorithms. By default it is turned on for gradient boosting, where in general it makes training faster, but turned off for all other algorithms as it tends to slow down training when training deep trees.

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.

Face completion with multi-output estimators



- Consider performing dimensionality reduction ([PCA](#), [ICA](#), or [Feature selection](#)) beforehand to give your tree a better chance of finding features that are discriminative.
- [Understanding the decision tree structure](#) will help in gaining more insights about how the decision tree makes predictions, which is important for understanding the important features in the data.
- 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 ensure that multiple samples inform every decision in the tree, by controlling which splits will be considered. 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. If the sample size varies greatly, a float number can be used as percentage in these two parameters. While `min_samples_split` can create arbitrarily small leaves, `min_samples_leaf` guarantees that each leaf has a minimum size, avoiding low-variance, over-fit leaf nodes in regression problems. For classification with few classes, `min_samples_leaf=1` is often the best choice.
- Balance your dataset before training to prevent the tree from being biased toward the classes that are dominant. Class balancing can be done by sampling an equal number of samples from each class, or preferably by normalizing the sum of the sample weights (`sample_weight`) for each class to the same value. Also note that weight-based pre-pruning criteria, such as `min_weight_fraction_leaf`, will then be less biased toward dominant classes than criteria that are not aware of the sample weights, like `min_samples_leaf`.
- If the samples are weighted, it will be easier to optimize the tree structure using weight-based pre-pruning criterion such as `min_weight_fraction_leaf`, which ensure that leaf nodes contain at least a fraction of the overall sum of the sample weights.
- All decision trees use `np.float32` arrays internally. If training data is not in this format, a copy of the dataset will be made.
- If the input matrix `X` is very sparse, it is recommended to convert to sparse `csc_matrix` before calling `fit` and sparse `csr_matrix` before calling `predict`. Training time can be orders of magnitude faster for a sparse matrix input compared to a dense matrix when features have zero values in most of the samples.

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; however, scikit-learn implementation does not support categorical variables for now.

Mathematical formulation

Given training vectors $x_i \in R^n$, $i=1, \dots, l$ 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

$$\begin{aligned} Q_{left}(\theta) &= (x, y) | x_j \leq t_m \\ Q_{right}(\theta) &= Q \setminus Q_{left}(\theta) \end{aligned}$$

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, $N_m < \text{min}_samples$ or $N_m = 1$.

Classification criteria

If a target is a classification outcome taking on values $0, 1, \dots, K-1$, 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

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

Entropy

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

and Misclassification

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

where X_m is the training data in node m

Regression criteria

If the target is a continuous value, then for node m , representing a region R_m with N_m observations, common criteria to minimise as for determining locations for future splits are Mean Squared Error, which minimizes the L2 error using mean values at terminal nodes, and Mean Absolute Error, which minimizes the L1 error using median values at terminal nodes.

Mean Squared Error:

$$\bar{y}_m = \frac{1}{N_m} \sum_{i \in N_m} y_i$$

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

Mean Absolute Error:

$$\bar{y}_m = \frac{1}{N_m} \sum_{i \in N_m} y_i$$

$$H(X_m) = \frac{1}{N_m} \sum_{i \in N_m} |y_i - \bar{y}_m|$$

where X_m is the training data in node m

References:

- https://en.wikipedia.org/wiki/Decision_tree_learning
- https://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.

3.1.11 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, ...*

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. When using a subset of the available samples the generalization accuracy can be estimated with the out-of-bag samples by setting `oob_score=True`. 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

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: a sparse or dense 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}]).

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.

Furthermore, when splitting each node during the construction of a tree, the best split is found either from all input features or a random subset of size `max_features`. (See the *parameter tuning guidelines* for more details).

The purpose of these two sources of randomness is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit high variance and tend to overfit. The injected randomness in forests yield decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant 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.

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:

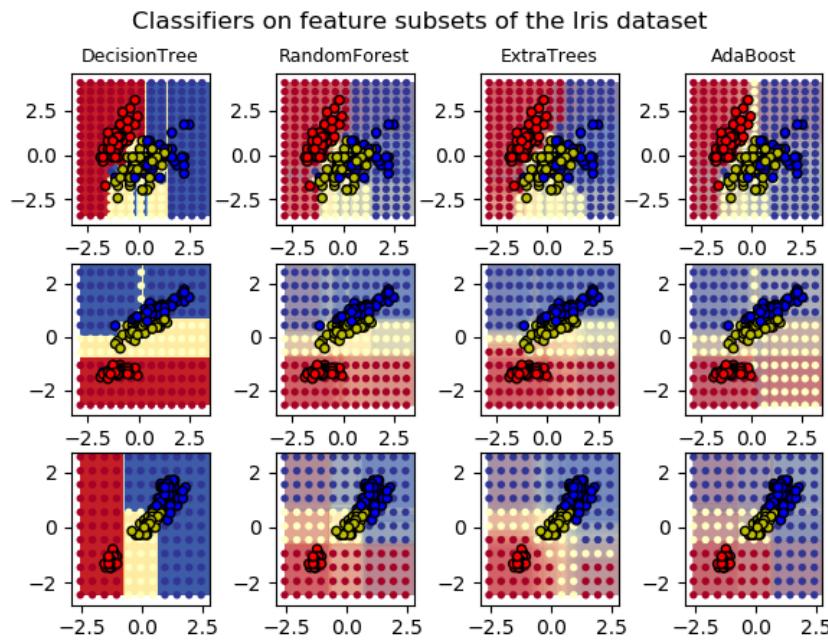
```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import make_blobs
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.tree import DecisionTreeClassifier

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

>>> clf = DecisionTreeClassifier(max_depth=None, min_samples_split=2,
...                                 random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.98...

>>> clf = RandomForestClassifier(n_estimators=10, max_depth=None,
...                               min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
```

```
>>> scores.mean()
0.999...
>>> clf = ExtraTreesClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean() > 0.999
True
```



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=None` (always considering all features instead of a random subset) for regression problems, and `max_features="sqrt"` (using a random subset of size $\sqrt{n_features}$) for classification tasks (where `n_features` is the number of features in the data). Good results are often achieved when setting `max_depth=None` in combination with `min_samples_split=2` (i.e., when fully developing the trees). Bear in mind though that these values are usually not optimal, and might result in models that consume a lot of RAM. The best parameter values should always be cross-validated. In addition, note that in random forests, bootstrap samples are used by default (`bootstrap=True`) while the default strategy for extra-trees is to use the whole dataset (`bootstrap=False`). When using bootstrap sampling the generalization accuracy can be estimated on the left out or out-of-bag samples. This can be enabled by setting `oob_score=True`.

Note: The size of the model with the default parameters is $O(M * N * \log(N))$, where M is the number of trees and N is the number of samples. In order to reduce the size of the model, you can change these parameters: `min_samples_split`, `max_leaf_nodes`, `max_depth` and `min_samples_leaf`.

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

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

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 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**. In scikit-learn, the fraction of samples a feature contributes to is combined with the decrease in impurity from splitting them to create a normalized estimate of the predictive power of that feature.

By **averaging** the estimates of predictive ability over several randomized trees one can **reduce the variance** of such an estimate and use it for feature selection. This is known as the mean decrease in impurity, or MDI. Refer to [L2014] for more information on MDI and feature importance evaluation with Random Forests.

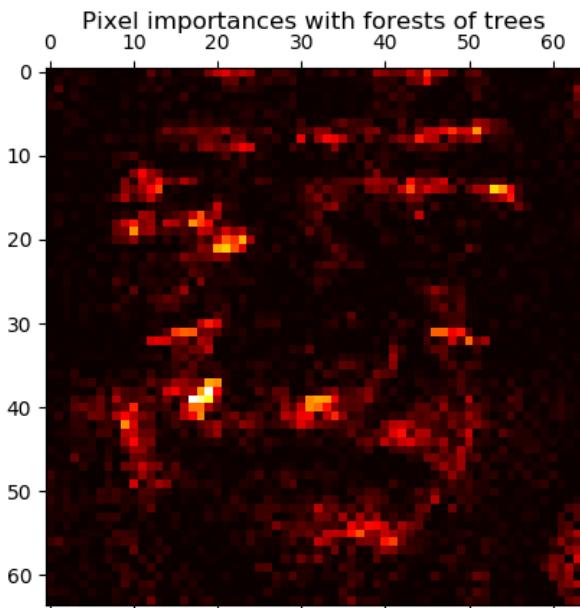
The following example shows a color-coded representation of the relative importances of each individual pixel for a face recognition task using a `ExtraTreesClassifier` model.

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*

References



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.
- [*Feature transformations with ensembles of trees*](#) compares supervised and unsupervised tree based feature transformations.

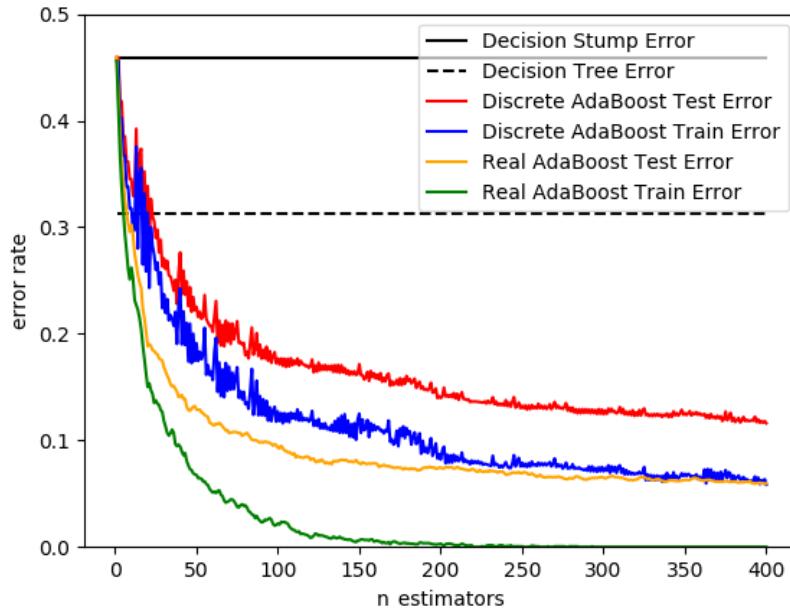
See also:

[*Manifold learning*](#) techniques can also be useful to derive non-linear representations of feature space, also these approaches focus also on dimensionality reduction.

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, \dots, 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].

Usage

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

```
>>> from sklearn.model_selection 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, cv=5)
>>> 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 to consider a split `min_samples_split`).

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

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.

Note: Scikit-learn 0.21 introduces two new experimental implementation of gradient boosting trees, namely `HistGradientBoostingClassifier` and `HistGradientBoostingRegressor`, inspired by `LightGBM`. These fast estimators first bin the input samples `X` into integer-valued bins (typically 256 bins) which tremendously reduces the number of splitting points to consider, and allow the algorithm to leverage integer-based data structures (histograms) instead of relying on sorted continuous values.

The new histogram-based estimators can be orders of magnitude faster than their continuous counterparts when the number of samples is larger than tens of thousands of samples. The API of these new estimators is slightly different, and some of the features from `GradientBoostingClassifier` and `GradientBoostingRegressor` are not yet supported.

These new estimators are still **experimental** for now: their predictions and their API might change without any deprecation cycle. To use them, you need to explicitly import `enable_hist_gradient_boosting`:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_hist_gradient_boosting # noqa
>>> # now you can import normally from ensemble
>>> from sklearn.ensemble import HistGradientBoostingClassifier
```

The following guide focuses on `GradientBoostingClassifier` and `GradientBoostingRegressor` only, which might be preferred for small sample sizes since binning may lead to split points that are too approximate in this setting.

Classification

`GradientBoostingClassifier` supports both binary and multi-class classification. 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 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`.

Regression

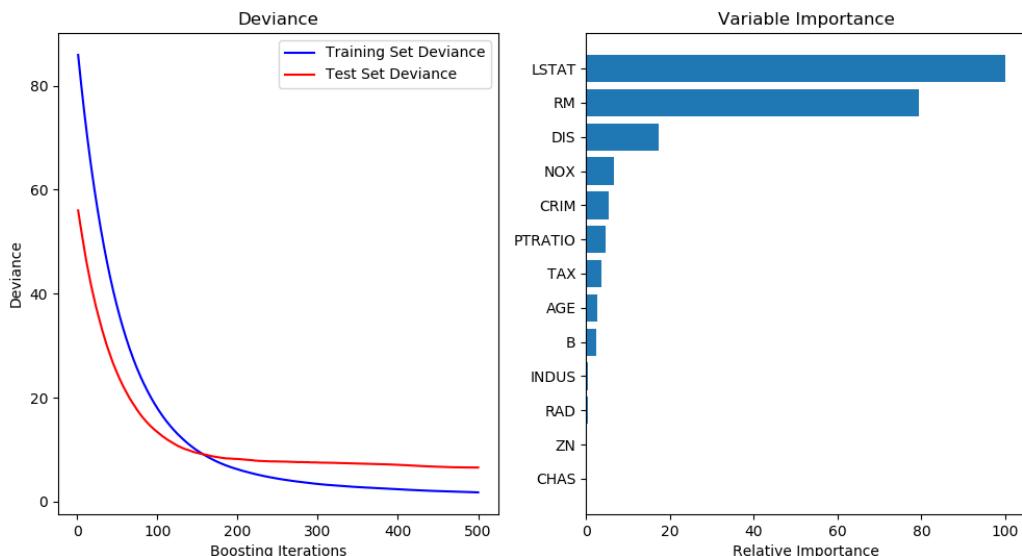
`GradientBoostingRegressor` supports a number of *different loss functions* for regression which can be specified via the argument `loss`; the default loss function for regression is least squares ('ls').

```
>>> 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`

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...
```

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`.

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 greedy fashion:

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

where the newly added tree h_m tries to minimize the loss L , given the previous ensemble F_{m-1} :

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

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.

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 $0 < \text{alpha} < 1$ 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.
 - Exponential loss ('exponential'): The same loss function as [AdaBoostClassifier](#). Less robust to mislabeled examples than 'deviance'; can only be used for binary classification.

Regularization

Shrinkage

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

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

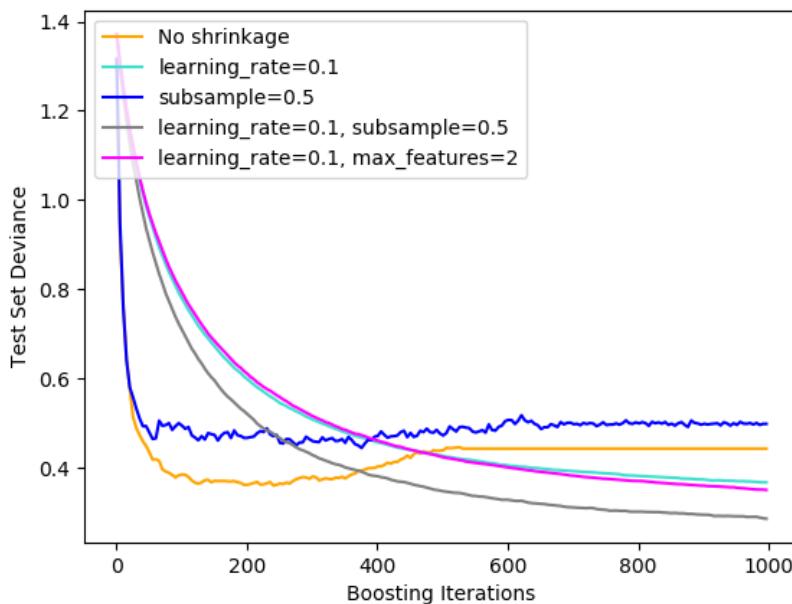
The parameter ν is also called the **learning rate** because it scales the step length the gradient descent procedure; it can be set via the `learning_rate` parameter.

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].

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](#)
- [OOB Errors for Random Forests](#)

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.

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([0.10..., 0.10..., 0.11..., ...])
```

Examples:

- Gradient Boosting regression

Voting Classifier

The idea behind the `VotingClassifier` is to combine conceptually different machine learning classifiers and use a majority vote or the average predicted probabilities (soft vote) to predict the class labels. Such a classifier can be useful for a set of equally well performing model in order to balance out their individual weaknesses.

Majority Class Labels (Majority/Hard Voting)

In majority voting, the predicted class label for a particular sample is the class label that represents the majority (mode) of the class labels predicted by each individual classifier.

E.g., if the prediction for a given sample is

- classifier 1 -> class 1
- classifier 2 -> class 1
- classifier 3 -> class 2

the `VotingClassifier` (with `voting='hard'`) would classify the sample as “class 1” based on the majority class label.

In the cases of a tie, the `VotingClassifier` will select the class based on the ascending sort order. E.g., in the following scenario

- classifier 1 -> class 2
- classifier 2 -> class 1

the class label 1 will be assigned to the sample.

Usage

The following example shows how to fit the majority rule classifier:

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import VotingClassifier

>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, 1:3], iris.target

>>> clf1 = LogisticRegression(solver='lbfgs', multi_class='multinomial',
...                             random_state=1)
>>> clf2 = RandomForestClassifier(n_estimators=50, random_state=1)
>>> clf3 = GaussianNB()

>>> eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], 
...     voting='hard')

>>> for clf, label in zip([clf1, clf2, clf3, eclf], ['Logistic Regression', 'Random
...     Forest', 'naive Bayes', 'Ensemble']):
```

```

...     scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
...     print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), 
...                                                 label))
Accuracy: 0.95 (+/- 0.04) [Logistic Regression]
Accuracy: 0.94 (+/- 0.04) [Random Forest]
Accuracy: 0.91 (+/- 0.04) [naive Bayes]
Accuracy: 0.95 (+/- 0.04) [Ensemble]

```

Weighted Average Probabilities (Soft Voting)

In contrast to majority voting (hard voting), soft voting returns the class label as argmax of the sum of predicted probabilities.

Specific weights can be assigned to each classifier via the `weights` parameter. When weights are provided, the predicted class probabilities for each classifier are collected, multiplied by the classifier weight, and averaged. The final class label is then derived from the class label with the highest average probability.

To illustrate this with a simple example, let's assume we have 3 classifiers and a 3-class classification problems where we assign equal weights to all classifiers: $w_1=1$, $w_2=1$, $w_3=1$.

The weighted average probabilities for a sample would then be calculated as follows:

classifier	class 1	class 2	class 3
classifier 1	$w_1 * 0.2$	$w_1 * 0.5$	$w_1 * 0.3$
classifier 2	$w_2 * 0.6$	$w_2 * 0.3$	$w_2 * 0.1$
classifier 3	$w_3 * 0.3$	$w_3 * 0.4$	$w_3 * 0.3$
weighted average	0.37	0.4	0.23

Here, the predicted class label is 2, since it has the highest average probability.

The following example illustrates how the decision regions may change when a soft `VotingClassifier` is used based on an linear Support Vector Machine, a Decision Tree, and a K-nearest neighbor classifier:

```

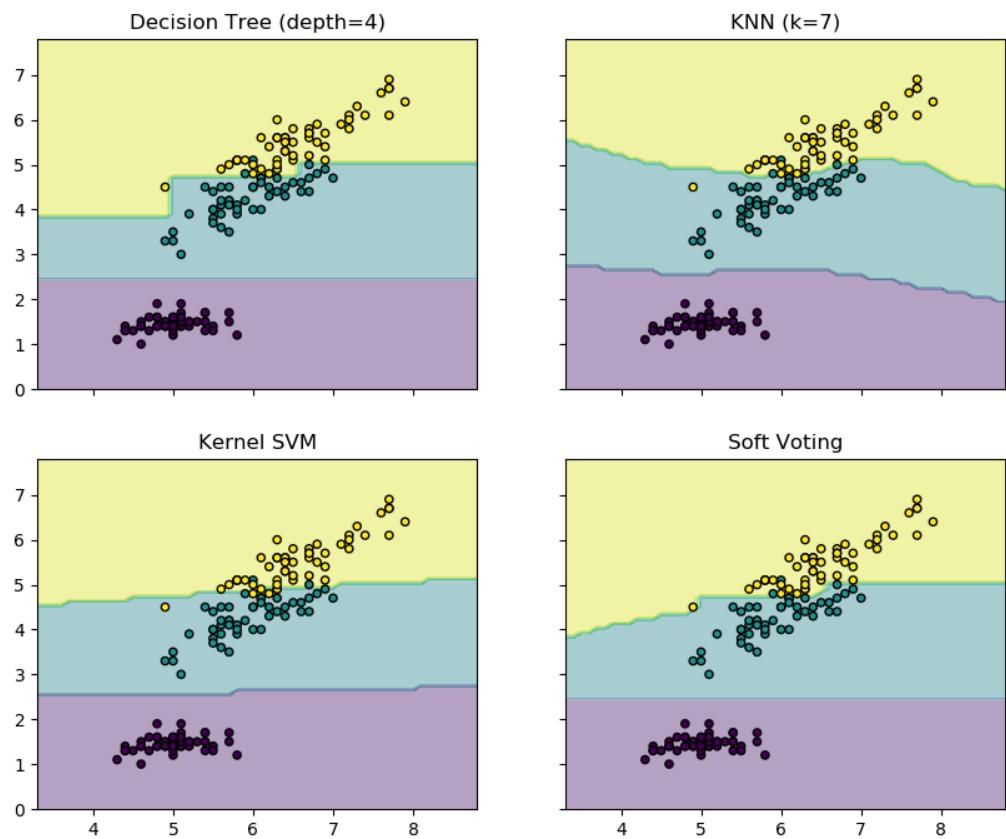
>>> from sklearn import datasets
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.svm import SVC
>>> from itertools import product
>>> from sklearn.ensemble import VotingClassifier

>>> # Loading some example data
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [0, 2]]
>>> y = iris.target

>>> # Training classifiers
>>> clf1 = DecisionTreeClassifier(max_depth=4)
>>> clf2 = KNeighborsClassifier(n_neighbors=7)
>>> clf3 = SVC(gamma='scale', kernel='rbf', probability=True)
>>> eclf = VotingClassifier(estimators=[('dt', clf1), ('knn', clf2), ('svc', clf3)],
...                           voting='soft', weights=[2, 1, 2])

>>> clf1 = clf1.fit(X, y)
>>> clf2 = clf2.fit(X, y)
>>> clf3 = clf3.fit(X, y)
>>> eclf = eclf.fit(X, y)

```



Using the `VotingClassifier` with `GridSearchCV`

The `VotingClassifier` can also be used together with `GridSearchCV` in order to tune the hyperparameters of the individual estimators:

```
>>> from sklearn.model_selection import GridSearchCV
>>> clf1 = LogisticRegression(solver='lbfgs', multi_class='multinomial',
...                             random_state=1)
>>> clf2 = RandomForestClassifier(random_state=1)
>>> clf3 = GaussianNB()
>>> eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],_
...                           voting='soft')

>>> params = {'lr__C': [1.0, 100.0], 'rf__n_estimators': [20, 200]}

>>> grid = GridSearchCV(estimator=eclf, param_grid=params, cv=5)
>>> grid = grid.fit(iris.data, iris.target)
```

Usage

In order to predict the class labels based on the predicted class-probabilities (scikit-learn estimators in the `VotingClassifier` must support `predict_proba` method):

```
>>> eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],_
...                           voting='soft')
```

Optionally, weights can be provided for the individual classifiers:

```
>>> eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],_
...                           voting='soft', weights=[2, 5, 1])
```

Voting Regressor

The idea behind the `VotingRegressor` is to combine conceptually different machine learning regressors and return the average predicted values. Such a regressor can be useful for a set of equally well performing models in order to balance out their individual weaknesses.

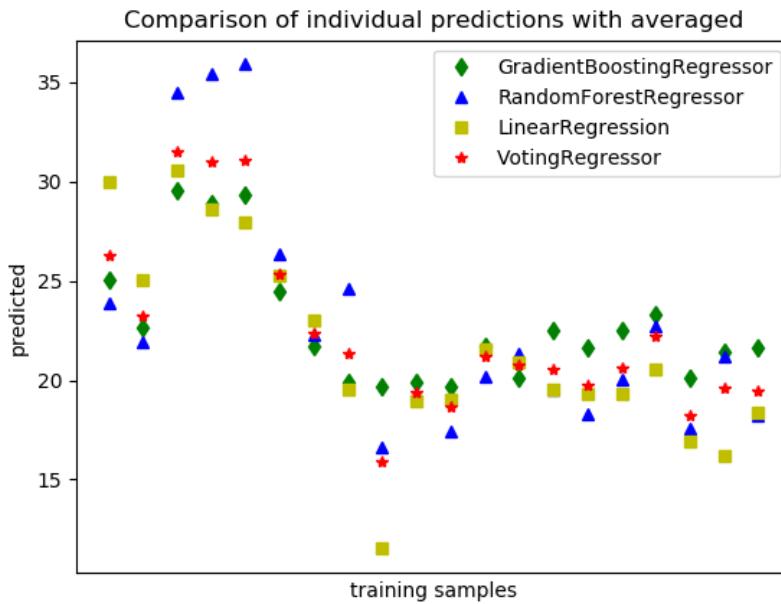
The following example shows how to fit the `VotingRegressor`:

```
>>> from sklearn import datasets
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.ensemble import VotingRegressor

>>> # Loading some example data
>>> boston = datasets.load_boston()
>>> X = boston.data
>>> y = boston.target

>>> # Training classifiers
>>> reg1 = GradientBoostingRegressor(random_state=1, n_estimators=10)
>>> reg2 = RandomForestRegressor(random_state=1, n_estimators=10)
>>> reg3 = LinearRegression()
```

```
>>> ereg = VotingRegressor(estimators=[('gb', reg1), ('rf', reg2), ('lr', reg3)])
>>> ereg.fit(X, y)
```



Examples:

- Plot individual and voting regression predictions

3.1.12 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. Multitarget regression is also supported.

- **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 regression** assigns each sample a set of target values. This can be thought of as predicting several properties for each data-point, such as wind direction and magnitude at a certain location.

- **Multioutput-multiclass classification** and **multi-task classification** means that a single estimator has to handle several joint classification tasks. This is both a generalization of the multi-label classification task, which only considers binary classification, as well as a generalization of the multi-class classification task. *The output format is a 2d numpy array or sparse matrix.*

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”; and “blue” or “green” for a second output variable that takes possible values in a finite set of colors such as “green”, “red”, “blue”, “yellow”...

This means that any classifiers handling multi-output multiclass or multi-task classification tasks, support 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:**

- `sklearn.naive_bayes.BernoulliNB`
- `sklearn.tree.DecisionTreeClassifier`
- `sklearn.tree.ExtraTreeClassifier`
- `sklearn.ensemble.ExtraTreesClassifier`
- `sklearn.naive_bayes.GaussianNB`
- `sklearn.neighbors.KNeighborsClassifier`
- `sklearn.semi_supervised.LabelPropagation`
- `sklearn.semi_supervised.LabelSpreading`
- `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`
- `sklearn.svm.LinearSVC` (setting `multi_class="crammer_singer"`)
- `sklearn.linear_model.LogisticRegression` (setting `multi_class="multinomial"`)
- `sklearn.linear_model.LogisticRegressionCV` (setting `multi_class="multinomial"`)
- `sklearn.neural_network.MLPClassifier`
- `sklearn.neighbors.NearestCentroid`
- `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis`
- `sklearn.neighbors.RadiusNeighborsClassifier`
- `sklearn.ensemble.RandomForestClassifier`
- `sklearn.linear_model.RidgeClassifier`
- `sklearn.linear_model.RidgeClassifierCV`

- **Multiclass as One-Vs-One:**

- `sklearn.svm.NuSVC`
- `sklearn.svm.SVC`.

- `sklearn.gaussian_process.GaussianProcessClassifier` (setting `multi_class = "one_vs_one"`)

- **Multiclass as One-Vs-All:**

- `sklearn.ensemble.GradientBoostingClassifier`
- `sklearn.gaussian_process.GaussianProcessClassifier` (setting `multi_class = "one_vs_rest"`)
- `sklearn.svm.LinearSVC` (setting `multi_class="ovr"`)
- `sklearn.linear_model.LogisticRegression` (setting `multi_class="ovr"`)
- `sklearn.linear_model.LogisticRegressionCV` (setting `multi_class="ovr"`)
- `sklearn.linear_model.SGDClassifier`
- `sklearn.linear_model.Perceptron`
- `sklearn.linear_model.PassiveAggressiveClassifier`

- **Support multilabel:**

- `sklearn.tree.DecisionTreeClassifier`
- `sklearn.tree.ExtraTreeClassifier`
- `sklearn.ensemble.ExtraTreesClassifier`
- `sklearn.neighbors.KNeighborsClassifier`
- `sklearn.neural_network.MLPClassifier`
- `sklearn.neighbors.RadiusNeighborsClassifier`
- `sklearn.ensemble.RandomForestClassifier`
- `sklearn.linear_model.RidgeClassifierCV`

- **Support multiclass-multioutput:**

- `sklearn.tree.DecisionTreeClassifier`
- `sklearn.tree.ExtraTreeClassifier`
- `sklearn.ensemble.ExtraTreesClassifier`
- `sklearn.neighbors.KNeighborsClassifier`
- `sklearn.neighbors.RadiusNeighborsClassifier`
- `sklearn.ensemble.RandomForestClassifier`

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

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 `MultiLabelBinarizer` transformer can be used to convert between a collection of collections of labels and the indicator format.

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> y = [[2, 3, 4], [2], [0, 1, 3], [0, 1, 2, 3, 4], [0, 1, 2]]
>>> MultiLabelBinarizer().fit_transform(y)
array([[0, 0, 1, 1, 1],
       [0, 0, 1, 0, 0],
       [1, 1, 0, 1, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 0, 0]])
```

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 only one classifier, 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.

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, 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, 2, 2, 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])
```

Multilabel learning

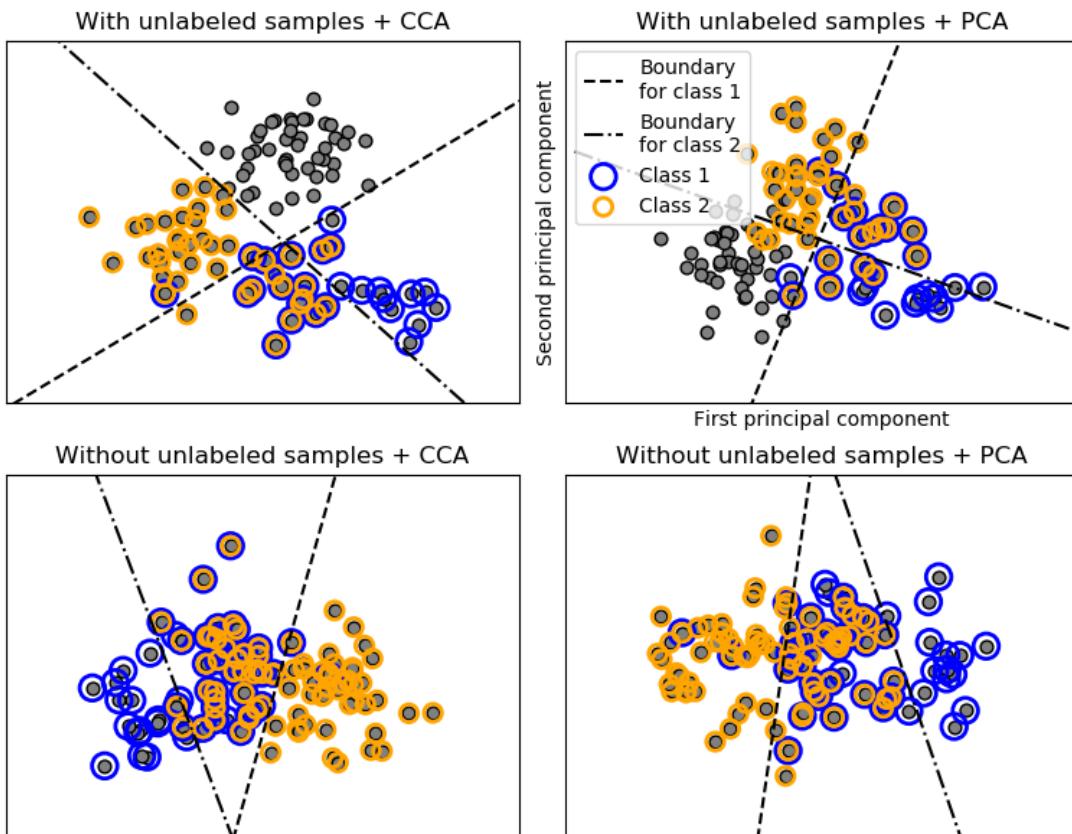
`OneVsRestClassifier` also supports multilabel classification. To use this feature, feed the classifier an indicator matrix, in which cell $[i, j]$ indicates the presence of label j in sample i .

Examples:

- *Multilabel classification*

One-Vs-One

`OneVsOneClassifier` constructs one classifier per pair of classes. At prediction time, the class which received the most votes is selected. In the event of a tie (among two classes with an equal number of votes), it selects the class with the highest aggregate classification confidence by summing over the pair-wise classification confidence levels computed by the underlying binary classifiers.



Since it requires to fit $n_{\text{classes}} * (n_{\text{classes}} - 1) / 2$ classifiers, this method is usually slower than one-vs-the-rest, due to its $O(n_{\text{classes}}^2)$ 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. The decision function is the result of a monotonic transformation of the one-versus-one classification.

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, 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, 2, 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])
```

References:

- “Pattern Recognition and Machine Learning. Springer”, Christopher M. Bishop, page 183, (First Edition)

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 in³ 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 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.

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, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 2, 2, 2, 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,
```

References:

³ “The error coding method and PICTs”, James G., Hastie T., Journal of Computational and Graphical statistics 7, 1998.

- “Solving multiclass learning problems via error-correcting output codes”, Dietterich T., Bakiri G., Journal of Artificial Intelligence Research 2, 1995.
- “The Elements of Statistical Learning”, Hastie T., Tibshirani R., Friedman J., page 606 (second-edition) 2008.

Multiooutput regression

Multiooutput regression support can be added to any regressor with MultiOutputRegressor. This strategy consists of fitting one regressor per target. Since each target is represented by exactly one regressor it is possible to gain knowledge about the target by inspecting its corresponding regressor. As MultiOutputRegressor fits one regressor per target it can not take advantage of correlations between targets.

Below is an example of multiooutput regression:

```
>>> from sklearn.datasets import make_regression
>>> from sklearn.multioutput import MultiOutputRegressor
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> X, y = make_regression(n_samples=10, n_targets=3, random_state=1)
>>> MultiOutputRegressor(GradientBoostingRegressor(random_state=0)).fit(X, y).
->predict(X)
array([[-154.75474165, -147.03498585, -50.03812219],
       [ 7.12165031,   5.12914884, -81.46081961],
       [-187.8948621 , -100.44373091,  13.88978285],
       [-141.62745778,  95.02891072, -191.48204257],
       [ 97.03260883, 165.34867495, 139.52003279],
       [ 123.92529176, 21.25719016, -7.84253 ],
       [-122.25193977, -85.16443186, -107.12274212],
       [ -30.170388 , -94.80956739,  12.16979946],
       [ 140.72667194, 176.50941682, -17.50447799],
       [ 149.37967282, -81.15699552, -5.72850319]])
```

Multiooutput classification

Multiooutput classification support can be added to any classifier with MultiOutputClassifier. This strategy consists of fitting one classifier per target. This allows multiple target variable classifications. The purpose of this class is to extend estimators to be able to estimate a series of target functions ($f_1, f_2, f_3 \dots, f_n$) that are trained on a single X predictor matrix to predict a series of responses ($y_1, y_2, y_3 \dots, y_n$).

Below is an example of multiooutput classification:

```
>>> from sklearn.datasets import make_classification
>>> from sklearn.multioutput import MultiOutputClassifier
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.utils import shuffle
>>> import numpy as np
>>> X, y1 = make_classification(n_samples=10, n_features=100, n_informative=30, n_
->classes=3, random_state=1)
>>> y2 = shuffle(y1, random_state=1)
>>> y3 = shuffle(y1, random_state=2)
>>> Y = np.vstack((y1, y2, y3)).T
>>> n_samples, n_features = X.shape # 10,100
>>> n_outputs = Y.shape[1] # 3
>>> n_classes = 3
>>> forest = RandomForestClassifier(n_estimators=100, random_state=1)
>>> multi_target_forest = MultiOutputClassifier(forest, n_jobs=-1)
```

```
>>> multi_target_forest.fit(X, Y).predict(X)
array([[2, 2, 0],
       [1, 2, 1],
       [2, 1, 0],
       [0, 0, 2],
       [0, 2, 1],
       [0, 0, 2],
       [1, 1, 0],
       [1, 1, 1],
       [0, 0, 2],
       [2, 0, 0]])
```

Classifier Chain

Classifier chains (see `ClassifierChain`) are a way of combining a number of binary classifiers into a single multi-label model that is capable of exploiting correlations among targets.

For a multi-label classification problem with N classes, N binary classifiers are assigned an integer between 0 and $N-1$. These integers define the order of models in the chain. Each classifier is then fit on the available training data plus the true labels of the classes whose models were assigned a lower number.

When predicting, the true labels will not be available. Instead the predictions of each model are passed on to the subsequent models in the chain to be used as features.

Clearly the order of the chain is important. The first model in the chain has no information about the other labels while the last model in the chain has features indicating the presence of all of the other labels. In general one does not know the optimal ordering of the models in the chain so typically many randomly ordered chains are fit and their predictions are averaged together.

References:

Jesse Read, Bernhard Pfahringer, Geoff Holmes, Eibe Frank, “Classifier Chains for Multi-label Classification”, 2009.

Regressor Chain

Regressor chains (see `RegressorChain`) is analogous to `ClassifierChain` as a way of combining a number of regressions into a single multi-target model that is capable of exploiting correlations among targets.

3.1.13 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.

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

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

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 zero.

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 scores and p-values (or only scores for `SelectKBest` and `SelectPercentile`):

- For regression: `f_regression`, `mutual_info_regression`
- For classification: `chi2`, `f_classif`, `mutual_info_classif`

The methods based on F-test estimate the degree of linear dependency between two random variables. On the other hand, mutual information methods can capture any kind of statistical dependency, but being nonparametric, they require more samples for accurate estimation.

Feature selection with sparse data

If you use sparse data (i.e. data represented as sparse matrices), `chi2`, `mutual_info_regression`, `mutual_info_classif` 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*](#)
- [*Comparison of F-test and mutual information*](#)

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 the importance of each feature is obtained either through a `coef_` attribute or through a `feature_importances_` attribute. Then, the least important features are pruned from current set of 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.

Feature selection using SelectFromModel

[`SelectFromModel`](#) is a meta-transformer that can be used along with any estimator that has a `coef_` or `feature_importances_` attribute after fitting. The features are considered unimportant and removed, if the corresponding `coef_` or `feature_importances_` values are below the provided `threshold` parameter. Apart from specifying the threshold numerically, there are built-in heuristics for finding a threshold using a string argument. Available heuristics are “mean”, “median” and float multiples of these like “0.1*mean”.

For examples on how it is to be used refer to the sections below.

Examples

- *Feature selection using SelectFromModel and LassoCV*: Selecting the two most important features from the Boston dataset without knowing the threshold beforehand.

L1-based feature selection

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 can be used along with `feature_selection.SelectFromModel` to select the non-zero coefficients. 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
>>> from sklearn.feature_selection import SelectFromModel
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> lsvc = LinearSVC(C=0.01, penalty="l1", dual=False).fit(X, y)
>>> model = SelectFromModel(lsvc, prefit=True)
>>> X_new = model.transform(X)
>>> 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 be 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://users.isr.ist.utl.pt/~aguiar/CS_notes.pdf

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 (when coupled with the `sklearn.feature_selection.SelectFromModel` meta-transformer):

```
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectFromModel
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X.shape
(150, 4)
>>> clf = ExtraTreesClassifier(n_estimators=50)
>>> clf = clf.fit(X, y)
>>> clf.feature_importances_
array([ 0.04...,  0.05...,  0.4...,  0.4...])
>>> model = SelectFromModel(clf, prefit=True)
>>> X_new = model.transform(X)
>>> 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.

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

In this snippet we make use of a `sklearn.svm.LinearSVC` coupled with `sklearn.feature_selection.SelectFromModel` 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.

3.1.14 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`.

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.

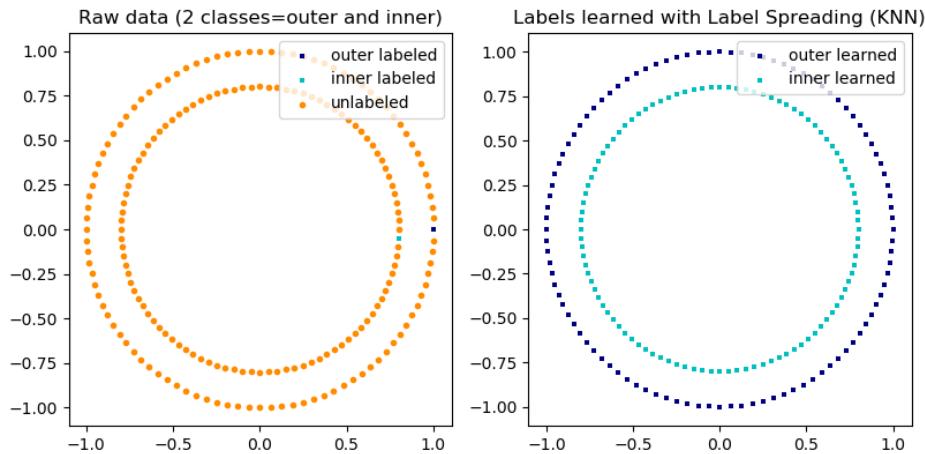


Fig. 3.1: **An illustration of label-propagation:** the structure of unlabeled observations is consistent with the class structure, and thus the class label can be propagated to the unlabeled observations of the training set.

`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 $\alpha = 0$. This clamping factor can be relaxed, to say $\alpha = 0.2$, which means that we will always retain 80 percent of our original label distribution, but the algorithm gets to change its 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: Demonstrating performance*
- *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 https://research.microsoft.com/en-us/people/nicolasl/efficient_ssl.pdf

3.1.15 Isotonic regression

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

$$\begin{aligned} & \text{minimize } \sum_i w_i (y_i - \hat{y}_i)^2 \\ & \text{subject to } \hat{y}_{\min} = \hat{y}_1 \leq \hat{y}_2 \dots \leq \hat{y}_n = \hat{y}_{\max} \end{aligned}$$

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.

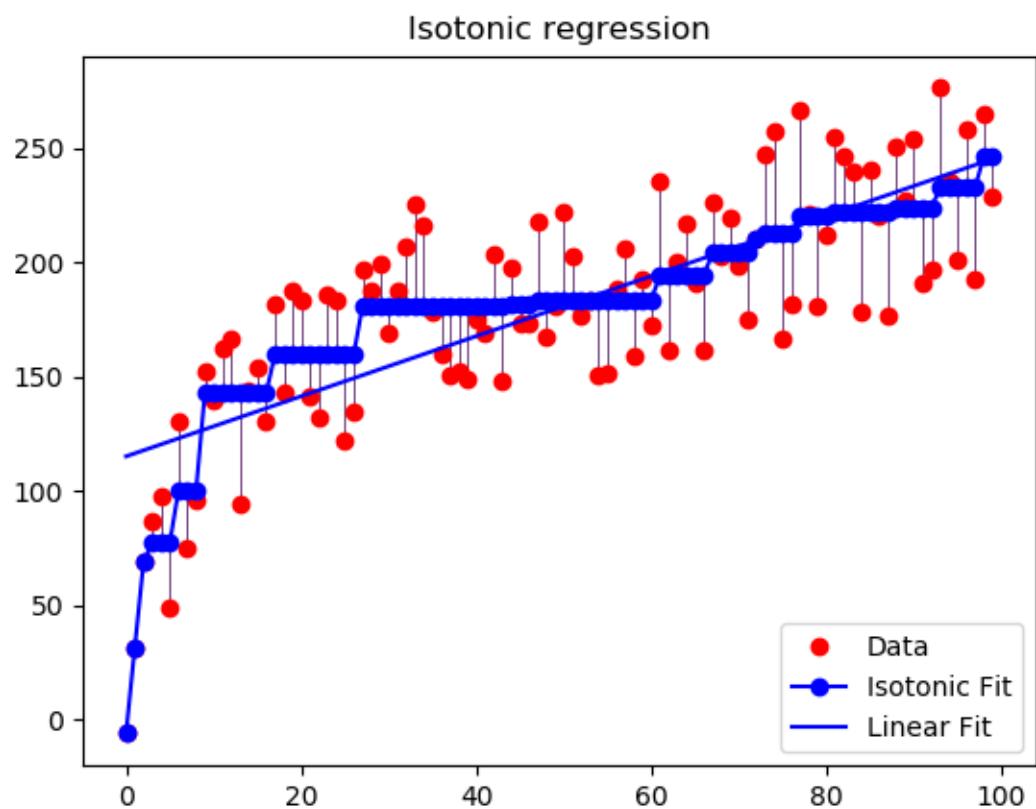
3.1.16 Probability calibration

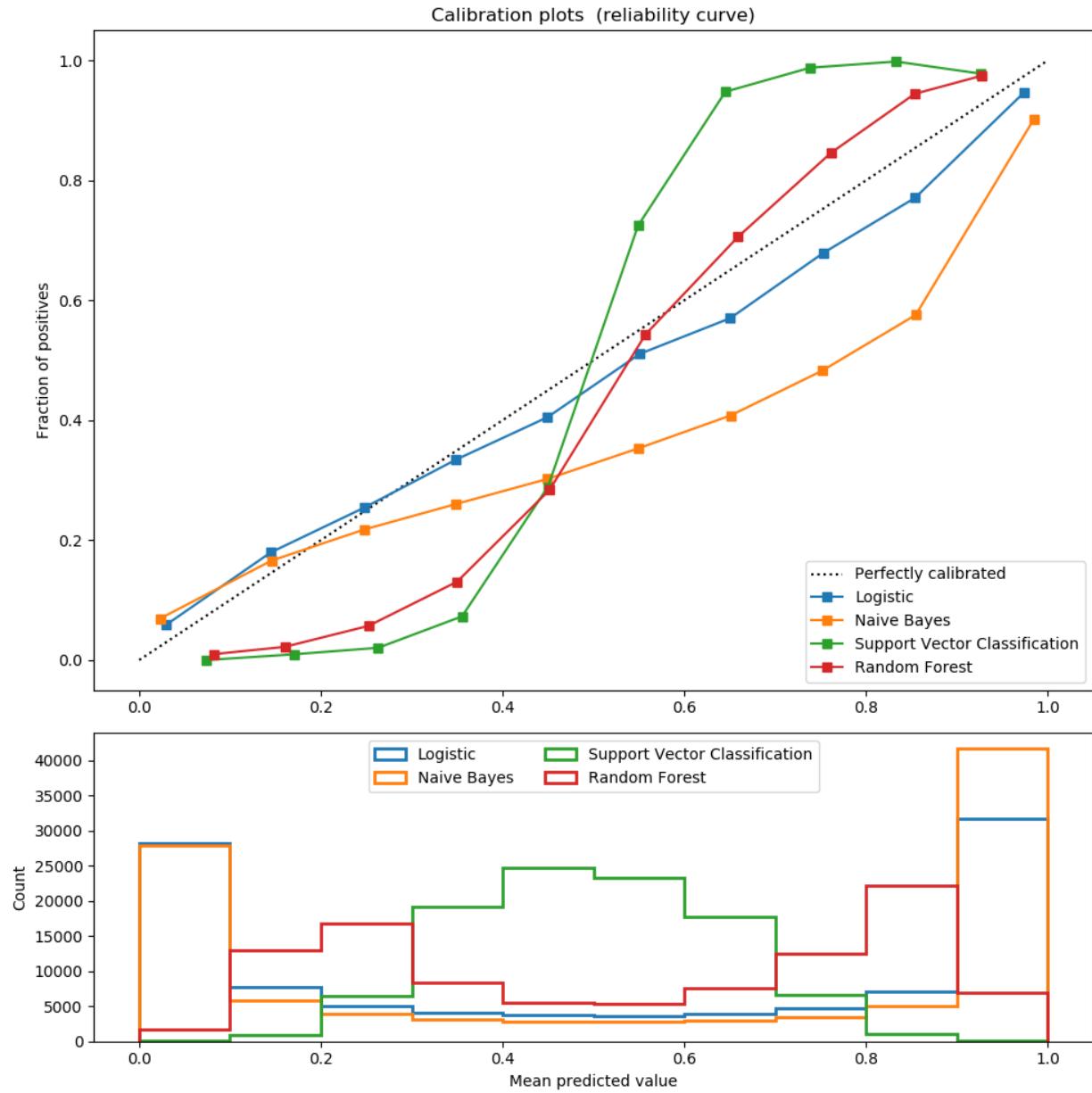
When performing classification you often want not only to predict the class label, but also obtain a probability of the respective label. This probability gives you some kind of confidence on the prediction. Some models can give you poor estimates of the class probabilities and some even do not support probability prediction. The calibration module allows you to better calibrate the probabilities of a given model, or to add support for probability prediction.

Well calibrated classifiers are probabilistic classifiers for which the output of the `predict_proba` method can be directly interpreted as a confidence level. For instance, a well calibrated (binary) classifier should classify the samples such that among the samples to which it gave a `predict_proba` value close to 0.8, approximately 80% actually belong to the positive class. The following plot compares how well the probabilistic predictions of different classifiers are calibrated:

`LogisticRegression` returns well calibrated predictions by default as it directly optimizes log-loss. In contrast, the other methods return biased probabilities; with different biases per method:

- `GaussianNB` tends to push probabilities to 0 or 1 (note the counts in the histograms). This is mainly because it makes the assumption that features are conditionally independent given the class, which is not the case in this dataset which contains 2 redundant features.





- `RandomForestClassifier` shows the opposite behavior: the histograms show peaks at approximately 0.2 and 0.9 probability, while probabilities close to 0 or 1 are very rare. An explanation for this is given by Niculescu-Mizil and Caruana⁴: “Methods such as bagging and random forests that average predictions from a base set of models can have difficulty making predictions near 0 and 1 because variance in the underlying base models will bias predictions that should be near zero or one away from these values. Because predictions are restricted to the interval [0,1], errors caused by variance tend to be one-sided near zero and one. For example, if a model should predict $p = 0$ for a case, the only way bagging can achieve this is if all bagged trees predict zero. If we add noise to the trees that bagging is averaging over, this noise will cause some trees to predict values larger than 0 for this case, thus moving the average prediction of the bagged ensemble away from 0. We observe this effect most strongly with random forests because the base-level trees trained with random forests have relatively high variance due to feature subsetting.” As a result, the calibration curve also referred to as the reliability diagram (Wilks 1995⁵) shows a characteristic sigmoid shape, indicating that the classifier could trust its “intuition” more and return probabilities closer to 0 or 1 typically.
- Linear Support Vector Classification (`LinearSVC`) shows an even more sigmoid curve as the RandomForestClassifier, which is typical for maximum-margin methods (compare Niculescu-Mizil and Caruana⁴), which focus on hard samples that are close to the decision boundary (the support vectors).

Two approaches for performing calibration of probabilistic predictions are provided: a parametric approach based on Platt’s sigmoid model and a non-parametric approach based on isotonic regression (`sklearn.isotonic`). Probability calibration should be done on new data not used for model fitting. The class `CalibratedClassifierCV` uses a cross-validation generator and estimates for each split the model parameter on the train samples and the calibration of the test samples. The probabilities predicted for the folds are then averaged. Already fitted classifiers can be calibrated by `CalibratedClassifierCV` via the parameter `cv=`”prefit”. In this case, the user has to take care manually that data for model fitting and calibration are disjoint.

The following images demonstrate the benefit of probability calibration. The first image present a dataset with 2 classes and 3 blobs of data. The blob in the middle contains random samples of each class. The probability for the samples in this blob should be 0.5.

The following image shows on the data above the estimated probability using a Gaussian naive Bayes classifier without calibration, with a sigmoid calibration and with a non-parametric isotonic calibration. One can observe that the non-parametric model provides the most accurate probability estimates for samples in the middle, i.e., 0.5.

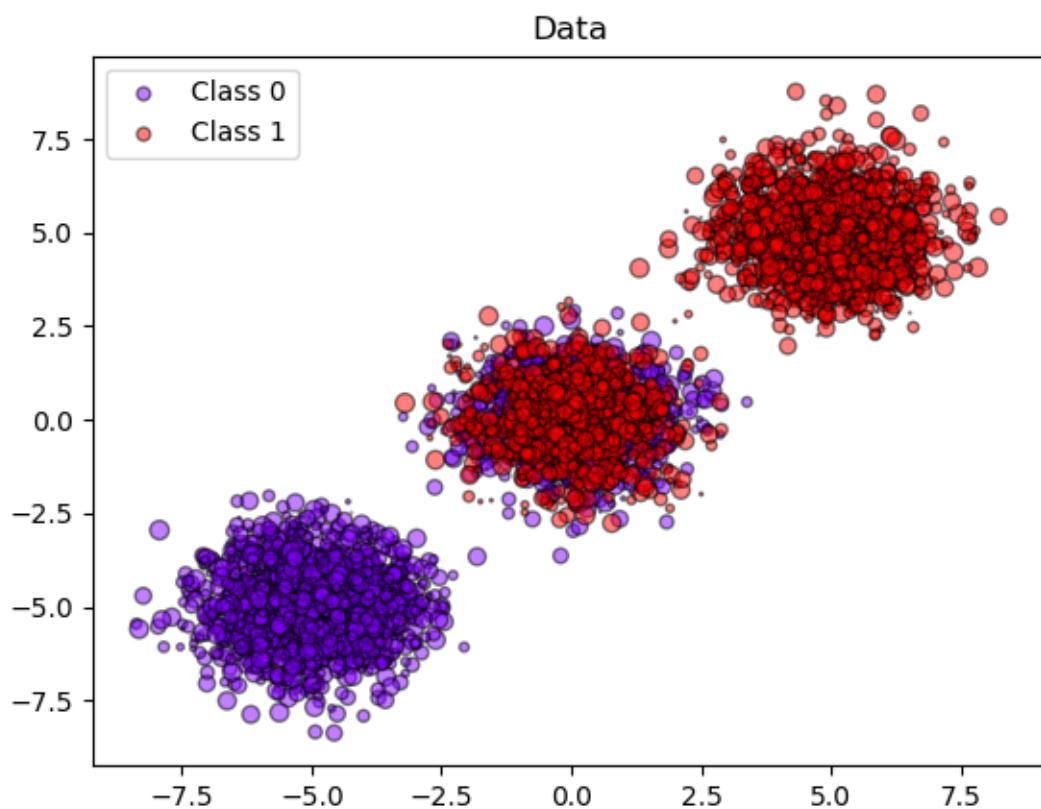
The following experiment is performed on an artificial dataset for binary classification with 100,000 samples (1,000 of them are used for model fitting) with 20 features. Of the 20 features, only 2 are informative and 10 are redundant. The figure shows the estimated probabilities obtained with logistic regression, a linear support-vector classifier (SVC), and linear SVC with both isotonic calibration and sigmoid calibration. The Brier score is a metric which is a combination of calibration loss and refinement loss, `brier_score_loss`, reported in the legend (the smaller the better). Calibration loss is defined as the mean squared deviation from empirical probabilities derived from the slope of ROC segments. Refinement loss can be defined as the expected optimal loss as measured by the area under the optimal cost curve.

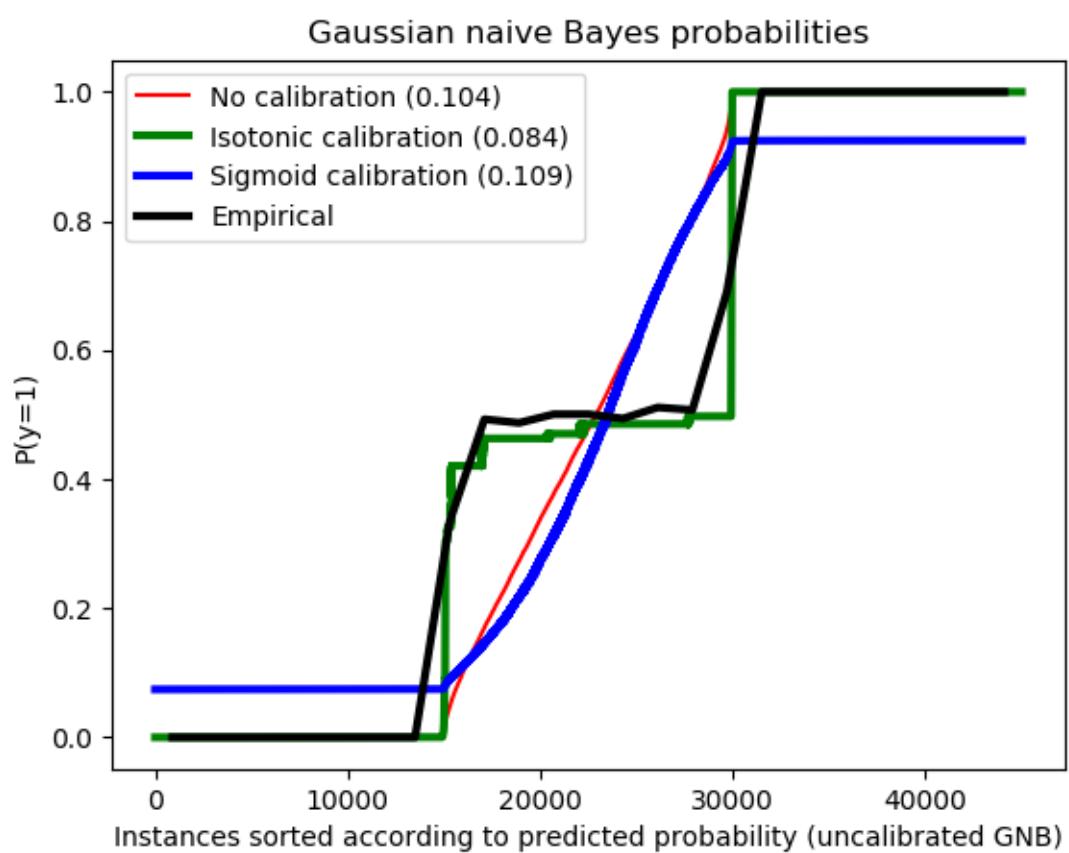
One can observe here that logistic regression is well calibrated as its curve is nearly diagonal. Linear SVC’s calibration curve or reliability diagram has a sigmoid curve, which is typical for an under-confident classifier. In the case of LinearSVC, this is caused by the margin property of the hinge loss, which lets the model focus on hard samples that are close to the decision boundary (the support vectors). Both kinds of calibration can fix this issue and yield nearly identical results. The next figure shows the calibration curve of Gaussian naive Bayes on the same data, with both kinds of calibration and also without calibration.

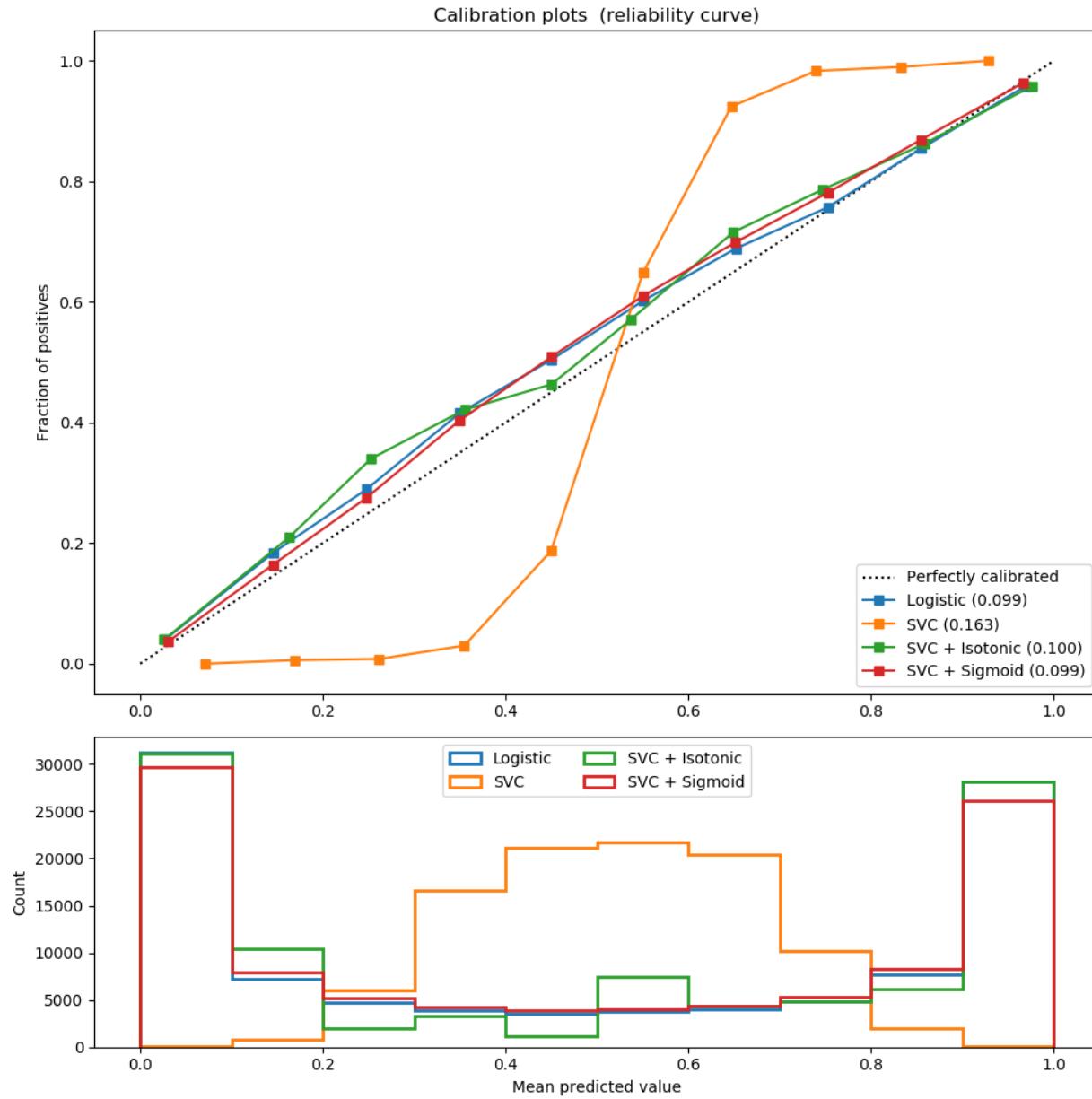
One can see that Gaussian naive Bayes performs very badly but does so in an other way than linear SVC: While linear SVC exhibited a sigmoid calibration curve, Gaussian naive Bayes’ calibration curve has a transposed-sigmoid shape. This is typical for an over-confident classifier. In this case, the classifier’s overconfidence is caused by the redundant features which violate the naive Bayes assumption of feature-independence.

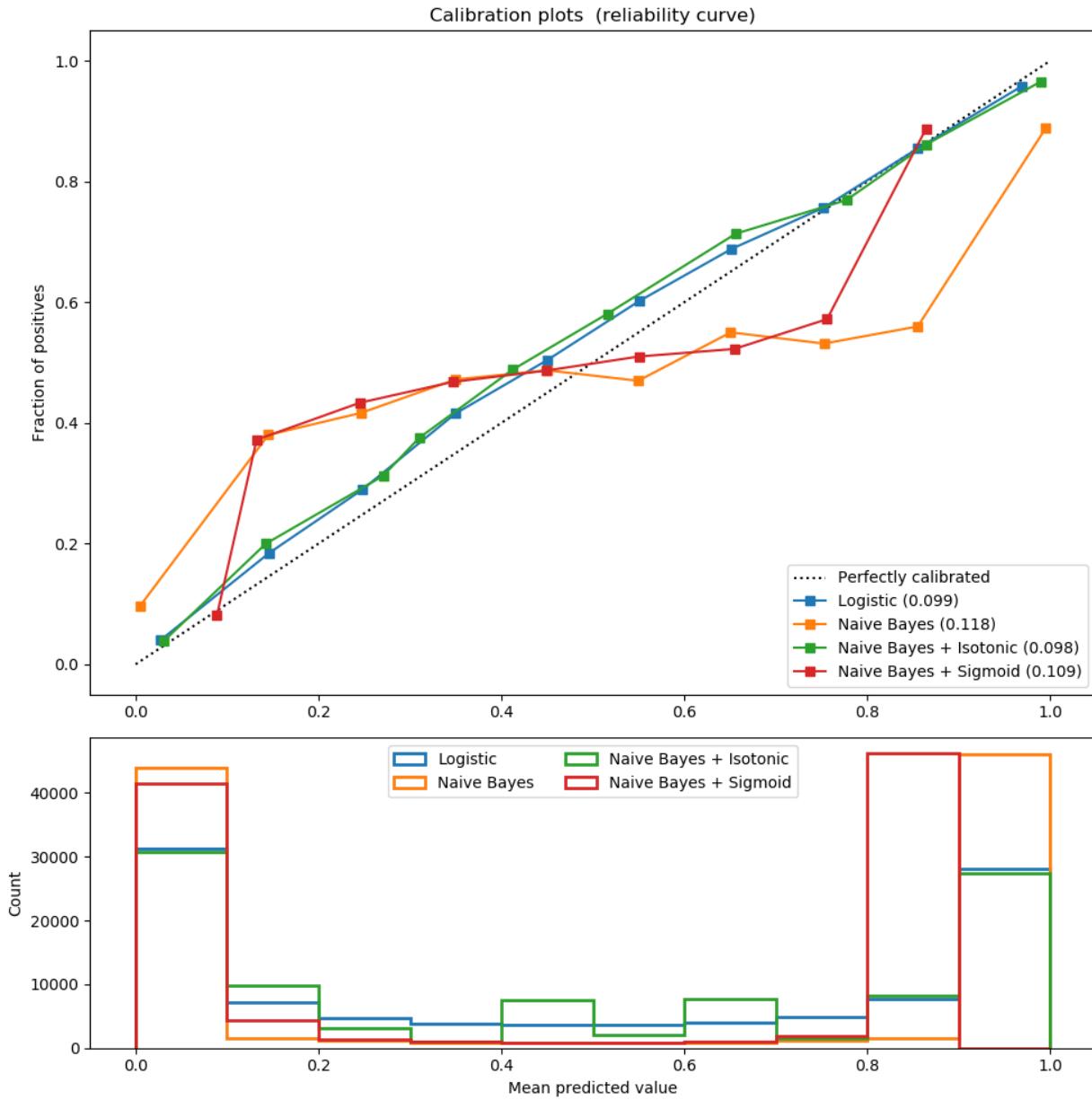
⁴ Predicting Good Probabilities with Supervised Learning, A. Niculescu-Mizil & R. Caruana, ICML 2005

⁵ On the combination of forecast probabilities for consecutive precipitation periods. Wea. Forecasting, 5, 640–650., Wilks, D. S., 1990a





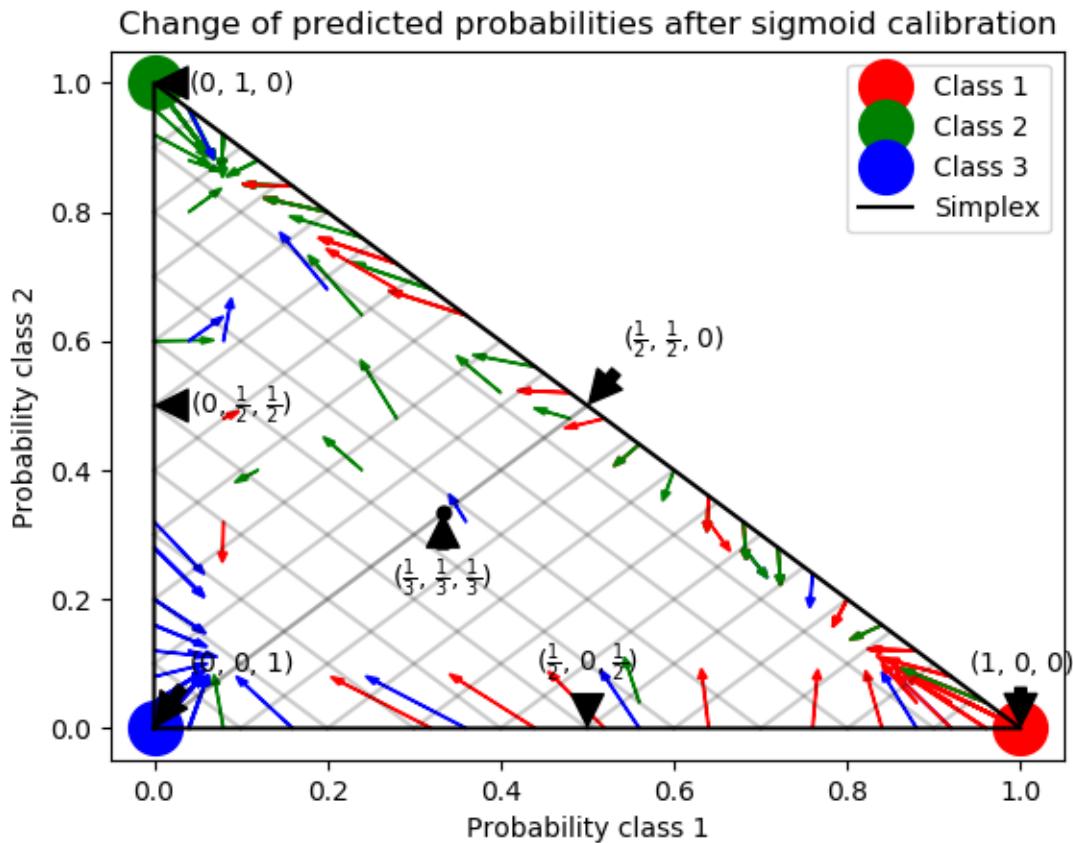




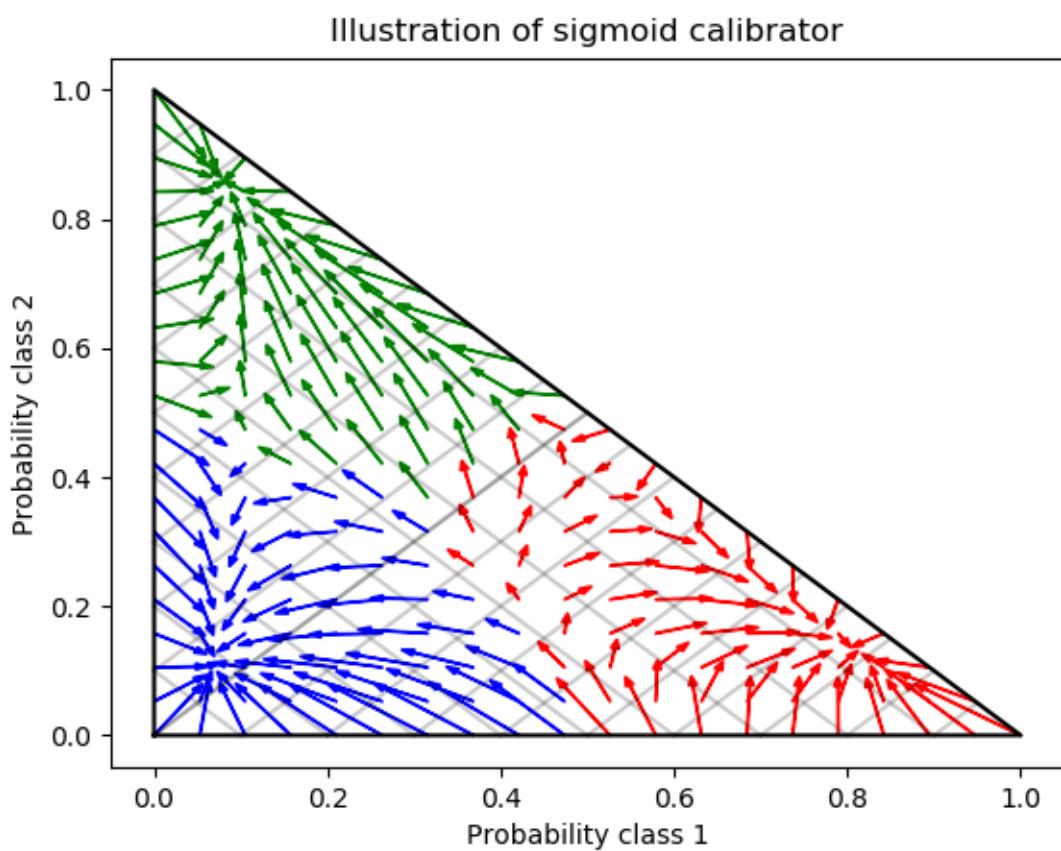
Calibration of the probabilities of Gaussian naive Bayes with isotonic regression can fix this issue as can be seen from the nearly diagonal calibration curve. Sigmoid calibration also improves the brier score slightly, albeit not as strongly as the non-parametric isotonic calibration. This is an intrinsic limitation of sigmoid calibration, whose parametric form assumes a sigmoid rather than a transposed-sigmoid curve. The non-parametric isotonic calibration model, however, makes no such strong assumptions and can deal with either shape, provided that there is sufficient calibration data. In general, sigmoid calibration is preferable in cases where the calibration curve is sigmoid and where there is limited calibration data, while isotonic calibration is preferable for non-sigmoid calibration curves and in situations where large amounts of data are available for calibration.

`CalibratedClassifierCV` can also deal with classification tasks that involve more than two classes if the base estimator can do so. In this case, the classifier is calibrated first for each class separately in an one-vs-rest fashion. When predicting probabilities for unseen data, the calibrated probabilities for each class are predicted separately. As those probabilities do not necessarily sum to one, a postprocessing is performed to normalize them.

The next image illustrates how sigmoid calibration changes predicted probabilities for a 3-class classification problem. Illustrated is the standard 2-simplex, where the three corners correspond to the three classes. Arrows point from the probability vectors predicted by an uncalibrated classifier to the probability vectors predicted by the same classifier after sigmoid calibration on a hold-out validation set. Colors indicate the true class of an instance (red: class 1, green: class 2, blue: class 3).



The base classifier is a random forest classifier with 25 base estimators (trees). If this classifier is trained on all 800 training datapoints, it is overly confident in its predictions and thus incurs a large log-loss. Calibrating an identical classifier, which was trained on 600 datapoints, with method='sigmoid' on the remaining 200 datapoints reduces the confidence of the predictions, i.e., moves the probability vectors from the edges of the simplex towards the center:



This calibration results in a lower log-loss. Note that an alternative would have been to increase the number of base estimators which would have resulted in a similar decrease in log-loss.

References:

- Obtaining calibrated probability estimates from decision trees and naive Bayesian classifiers, B. Zadrozny & C. Elkan, ICML 2001
- Transforming Classifier Scores into Accurate Multiclass Probability Estimates, B. Zadrozny & C. Elkan, (KDD 2002)
- Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods, J. Platt, (1999)

3.1.17 Neural network models (supervised)

Warning: This implementation is not intended for large-scale applications. In particular, scikit-learn offers no GPU support. For much faster, GPU-based implementations, as well as frameworks offering much more flexibility to build deep learning architectures, see [Related Projects](#).

Multi-layer Perceptron

Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function $f(\cdot) : R^m \rightarrow R^o$ by training on a dataset, where m is the number of dimensions for input and o is the number of dimensions for output. Given a set of features $X = x_1, x_2, \dots, x_m$ and a target y , it can learn a non-linear function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers. Figure 1 shows a one hidden layer MLP with scalar output.

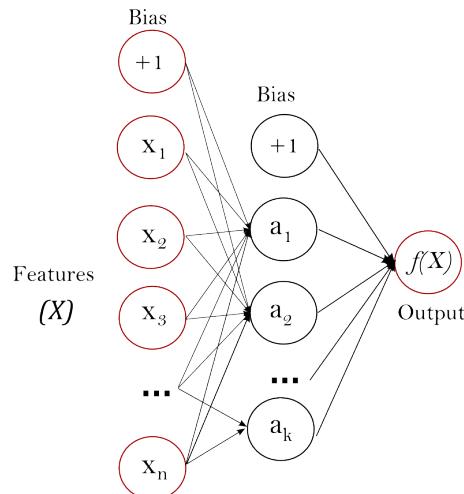


Fig. 3.2: **Figure 1 : One hidden layer MLP.**

The leftmost layer, known as the input layer, consists of a set of neurons $\{x_i | x_1, x_2, \dots, x_m\}$ representing the input features. Each neuron in the hidden layer transforms the values from the previous layer with a weighted linear summation $w_1x_1 + w_2x_2 + \dots + w_mx_m$, followed by a non-linear activation function $g(\cdot) : R \rightarrow R$ - like the hyperbolic tan function. The output layer receives the values from the last hidden layer and transforms them into output values.

The module contains the public attributes `coefs_` and `intercepts_`. `coefs_` is a list of weight matrices, where weight matrix at index i represents the weights between layer i and layer $i+1$. `intercepts_` is a list of bias vectors, where the vector at index i represents the bias values added to layer $i+1$.

The advantages of Multi-layer Perceptron are:

- Capability to learn non-linear models.
- Capability to learn models in real-time (on-line learning) using `partial_fit`.

The disadvantages of Multi-layer Perceptron (MLP) include:

- MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy.
- MLP requires tuning a number of hyperparameters such as the number of hidden neurons, layers, and iterations.
- MLP is sensitive to feature scaling.

Please see [Tips on Practical Use](#) section that addresses some of these disadvantages.

Classification

Class `MLPClassifier` implements a multi-layer perceptron (MLP) algorithm that trains using Backpropagation.

MLP trains on two arrays: array `X` of size (`n_samples`, `n_features`), which holds the training samples represented as floating point feature vectors; and array `y` of size (`n_samples`,), which holds the target values (class labels) for the training samples:

```
>>> from sklearn.neural_network import MLPClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                      hidden_layer_sizes=(5, 2), random_state=1)
...
>>> clf.fit(X, y)
MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
              beta_1=0.9, beta_2=0.999, early_stopping=False,
              epsilon=1e-08, hidden_layer_sizes=(5, 2),
              learning_rate='constant', learning_rate_init=0.001,
              max_iter=200, momentum=0.9, n_iter_no_change=10,
              nesterovs_momentum=True, power_t=0.5, random_state=1,
              shuffle=True, solver='lbfgs', tol=0.0001,
              validation_fraction=0.1, verbose=False, warm_start=False)
```

After fitting (training), the model can predict labels for new samples:

```
>>> clf.predict([[2., 2.], [-1., -2.]])
array([1, 0])
```

MLP can fit a non-linear model to the training data. `clf.coefs_` contains the weight matrices that constitute the model parameters:

```
>>> [coef.shape for coef in clf.coefs_]
[(2, 5), (5, 2), (2, 1)]
```

Currently, `MLPClassifier` supports only the Cross-Entropy loss function, which allows probability estimates by running the `predict_proba` method.

MLP trains using Backpropagation. More precisely, it trains using some form of gradient descent and the gradients are calculated using Backpropagation. For classification, it minimizes the Cross-Entropy loss function, giving a vector of probability estimates $P(y|x)$ per sample x :

```
>>> clf.predict_proba([[2., 2.], [1., 2.]])
array([[1.967...e-04, 9.998...-01],
       [1.967...e-04, 9.998...-01]])
```

`MLPClassifier` supports multi-class classification by applying Softmax as the output function.

Further, the model supports *multi-label classification* in which a sample can belong to more than one class. For each class, the raw output passes through the logistic function. Values larger or equal to 0.5 are rounded to 1, otherwise to 0. For a predicted output of a sample, the indices where the value is 1 represents the assigned classes of that sample:

```
>>> X = [[0., 0.], [1., 1.]]
>>> y = [[0, 1], [1, 1]]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                      hidden_layer_sizes=(15,), random_state=1)
...
>>> clf.fit(X, y)
MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
              beta_1=0.9, beta_2=0.999, early_stopping=False,
              epsilon=1e-08, hidden_layer_sizes=(15,),
              learning_rate='constant', learning_rate_init=0.001,
              max_iter=200, momentum=0.9, n_iter_no_change=10,
              nesterovs_momentum=True, power_t=0.5, random_state=1,
              shuffle=True, solver='lbfgs', tol=0.0001,
              validation_fraction=0.1, verbose=False, warm_start=False)
>>> clf.predict([[1., 2.]])
array([[1, 1]])
>>> clf.predict([[0., 0.]])
array([[0, 1]])
```

See the examples below and the docstring of `MLPClassifier.fit` for further information.

Examples:

- [Compare Stochastic learning strategies for MLPClassifier](#)
- [Visualization of MLP weights on MNIST](#)

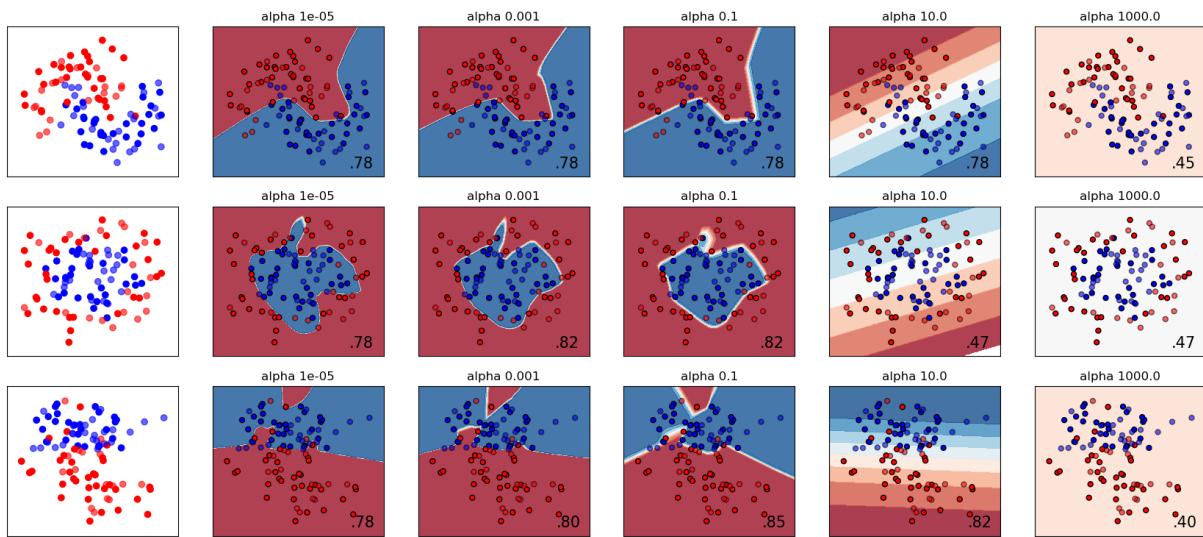
Regression

Class `MLPRegressor` implements a multi-layer perceptron (MLP) that trains using backpropagation with no activation function in the output layer, which can also be seen as using the identity function as activation function. Therefore, it uses the square error as the loss function, and the output is a set of continuous values.

`MLPRegressor` also supports multi-output regression, in which a sample can have more than one target.

Regularization

Both `MLPRegressor` and `MLPClassifier` use parameter `alpha` for regularization (L2 regularization) term which helps in avoiding overfitting by penalizing weights with large magnitudes. Following plot displays varying decision function with value of `alpha`.



See the examples below for further information.

Examples:

- [Varying regularization in Multi-layer Perceptron](#)

Algorithms

MLP trains using [Stochastic Gradient Descent](#), [Adam](#), or [L-BFGS](#). Stochastic Gradient Descent (SGD) updates parameters using the gradient of the loss function with respect to a parameter that needs adaptation, i.e.

$$w \leftarrow w - \eta \left(\alpha \frac{\partial R(w)}{\partial w} + \frac{\partial Loss}{\partial w} \right)$$

where η is the learning rate which controls the step-size in the parameter space search. $Loss$ is the loss function used for the network.

More details can be found in the documentation of [SGD](#)

Adam is similar to SGD in a sense that it is a stochastic optimizer, but it can automatically adjust the amount to update parameters based on adaptive estimates of lower-order moments.

With SGD or Adam, training supports online and mini-batch learning.

L-BFGS is a solver that approximates the Hessian matrix which represents the second-order partial derivative of a function. Further it approximates the inverse of the Hessian matrix to perform parameter updates. The implementation uses the Scipy version of [L-BFGS](#).

If the selected solver is ‘L-BFGS’, training does not support online nor mini-batch learning.

Complexity

Suppose there are n training samples, m features, k hidden layers, each containing h neurons - for simplicity, and o output neurons. The time complexity of backpropagation is $O(n \cdot m \cdot h^k \cdot o \cdot i)$, where i is the number of iterations. Since backpropagation has a high time complexity, it is advisable to start with smaller number of hidden neurons and few hidden layers for training.

Mathematical formulation

Given a set of training examples $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ where $x_i \in \mathbf{R}^n$ and $y_i \in \{0, 1\}$, a one hidden layer one hidden neuron MLP learns the function $f(x) = W_2g(W_1^T x + b_1) + b_2$ where $W_1 \in \mathbf{R}^m$ and $W_2, b_1, b_2 \in \mathbf{R}$ are model parameters. W_1, W_2 represent the weights of the input layer and hidden layer, respectively; and b_1, b_2 represent the bias added to the hidden layer and the output layer, respectively. $g(\cdot) : R \rightarrow R$ is the activation function, set by default as the hyperbolic tan. It is given as,

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

For binary classification, $f(x)$ passes through the logistic function $g(z) = 1/(1+e^{-z})$ to obtain output values between zero and one. A threshold, set to 0.5, would assign samples of outputs larger or equal 0.5 to the positive class, and the rest to the negative class.

If there are more than two classes, $f(x)$ itself would be a vector of size (n_classes,). Instead of passing through logistic function, it passes through the softmax function, which is written as,

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{l=1}^K \exp(z_l)}$$

where z_i represents the i th element of the input to softmax, which corresponds to class i , and K is the number of classes. The result is a vector containing the probabilities that sample x belong to each class. The output is the class with the highest probability.

In regression, the output remains as $f(x)$; therefore, output activation function is just the identity function.

MLP uses different loss functions depending on the problem type. The loss function for classification is Cross-Entropy, which in binary case is given as,

$$\text{Loss}(\hat{y}, y, W) = -y \ln \hat{y} - (1 - y) \ln (1 - \hat{y}) + \alpha ||W||_2^2$$

where $\alpha ||W||_2^2$ is an L2-regularization term (aka penalty) that penalizes complex models; and $\alpha > 0$ is a non-negative hyperparameter that controls the magnitude of the penalty.

For regression, MLP uses the Square Error loss function; written as,

$$\text{Loss}(\hat{y}, y, W) = \frac{1}{2} ||\hat{y} - y||_2^2 + \frac{\alpha}{2} ||W||_2^2$$

Starting from initial random weights, multi-layer perceptron (MLP) minimizes the loss function by repeatedly updating these weights. After computing the loss, a backward pass propagates it from the output layer to the previous layers, providing each weight parameter with an update value meant to decrease the loss.

In gradient descent, the gradient ∇Loss_W of the loss with respect to the weights is computed and deducted from W . More formally, this is expressed as,

$$W^{i+1} = W^i - \epsilon \nabla \text{Loss}_W^i$$

where i is the iteration step, and ϵ is the learning rate with a value larger than 0.

The algorithm stops when it reaches a preset maximum number of iterations; or when the improvement in loss is below a certain, small number.

Tips on Practical Use

- Multi-layer Perceptron 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 you must apply the *same* scaling to the test set for meaningful results. You can use StandardScaler for standardization.

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

An alternative and recommended approach is to use StandardScaler in a Pipeline

- Finding a reasonable regularization parameter α is best done using GridSearchCV, usually in the range $10^{-4} \dots 10^{-1}$.
- Empirically, we observed that L-BFGS converges faster and with better solutions on small datasets. For relatively large datasets, however, Adam is very robust. It usually converges quickly and gives pretty good performance. *Stochastic Gradient Descent* with momentum or nesterov's momentum, on the other hand, can perform better than those two algorithms if learning rate is correctly tuned.

More control with warm_start

If you want more control over stopping criteria or learning rate in SGD, or want to do additional monitoring, using `warm_start=True` and `max_iter=1` and iterating yourself can be helpful:

```
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = MLPClassifier(hidden_layer_sizes=(15,), random_state=1, max_iter=1, warm_
>>> start=True)
>>> for i in range(10):
...     clf.fit(X, y)
...     # additional monitoring / inspection
MLPClassifier(...)
```

References:

- “Learning representations by back-propagating errors.” Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams.
- “Stochastic Gradient Descent” L. Bottou - Website, 2010.
- “Backpropagation” Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen - Website, 2011.
- “Efficient BackProp” Y. LeCun, L. Bottou, G. Orr, K. Müller - In Neural Networks: Tricks of the Trade 1998.
- “Adam: A method for stochastic optimization.” Kingma, Diederik, and Jimmy Ba. arXiv preprint arXiv:1412.6980 (2014).

3.2 Unsupervised learning

3.2.1 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.

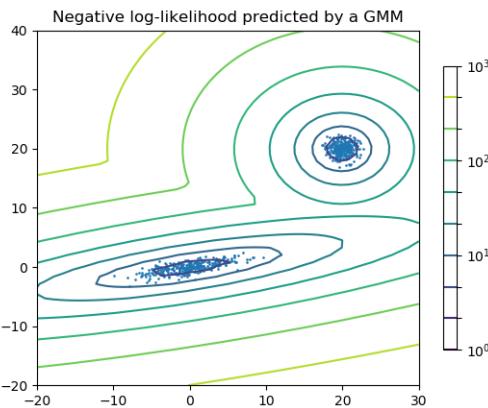


Fig. 3.3: Two-component Gaussian mixture model: data points, and equi-probability surfaces of the model.

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.

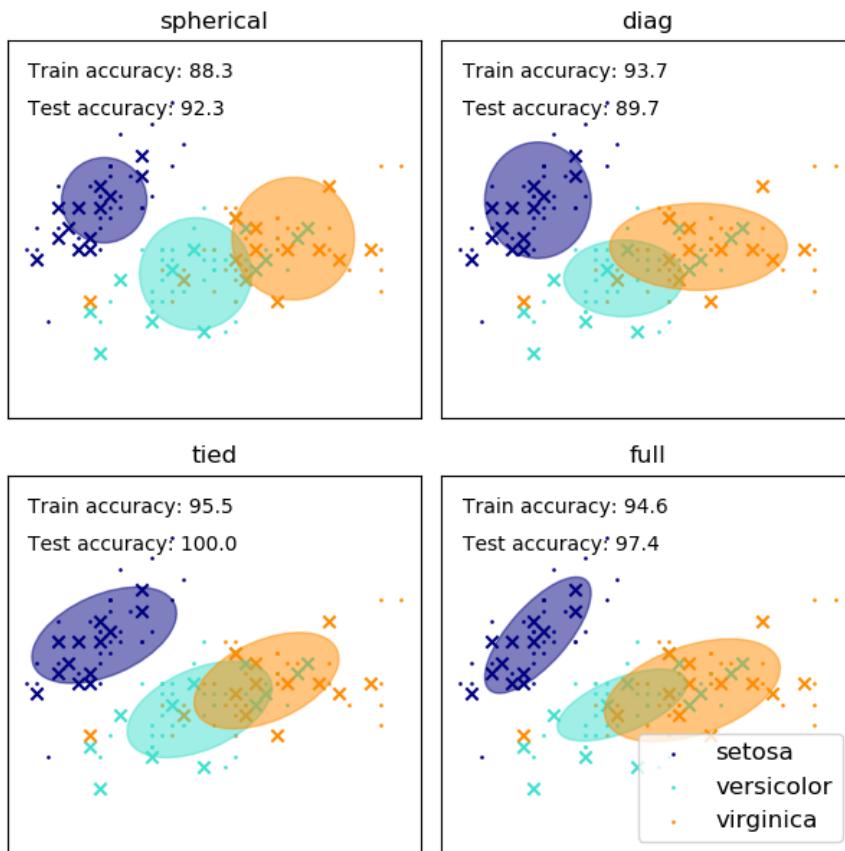
Gaussian Mixture

The `GaussianMixture` 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 `GaussianMixture.fit` method is provided that learns a Gaussian Mixture Model from train data. Given test data, it can assign to each sample the Gaussian it mostly probably belong to using the `GaussianMixture.predict` method.

The `GaussianMixture` comes with different options to constrain the covariance of the difference classes estimated: spherical, diagonal, tied or full covariance.

Examples:

- See [GMM covariances](#) for an example of using the Gaussian mixture as clustering on the iris dataset.
- See [Density Estimation for a Gaussian mixture](#) for an example on plotting the density estimation.



Pros and cons of class GaussianMixture

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.

Cons

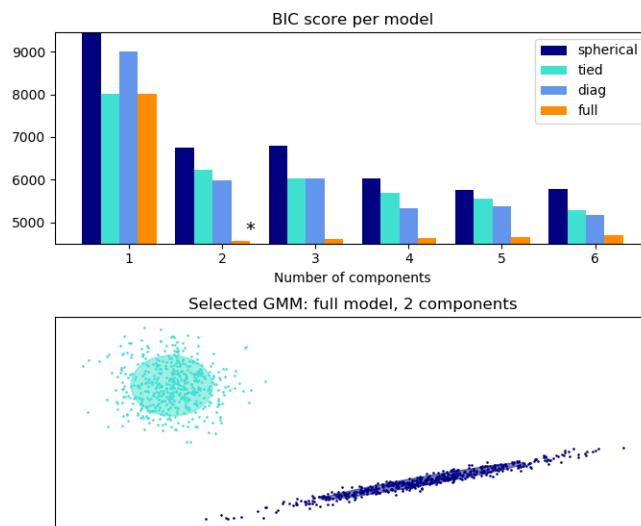
Singularities When one has insufficiently many points per mixture, estimating the covariance matrices becomes difficult, and the algorithm is known to diverge and find solutions with infinite likelihood unless one regularizes the covariances artificially.

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.

Selecting the number of components in a classical Gaussian Mixture Model

The BIC criterion can be used to select the number of components in a Gaussian Mixture 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 and assuming

that the data was actually generated i.i.d. from a mixture of Gaussian distribution). Note that using a [Variational Bayesian Gaussian mixture](#) 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 Gaussian mixture.

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-founded 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.

Variational Bayesian Gaussian Mixture

The [`BayesianGaussianMixture`](#) object implements a variant of the Gaussian mixture model with variational inference algorithms. The API is similar as the one defined by [`GaussianMixture`](#).

Estimation algorithm: variational inference

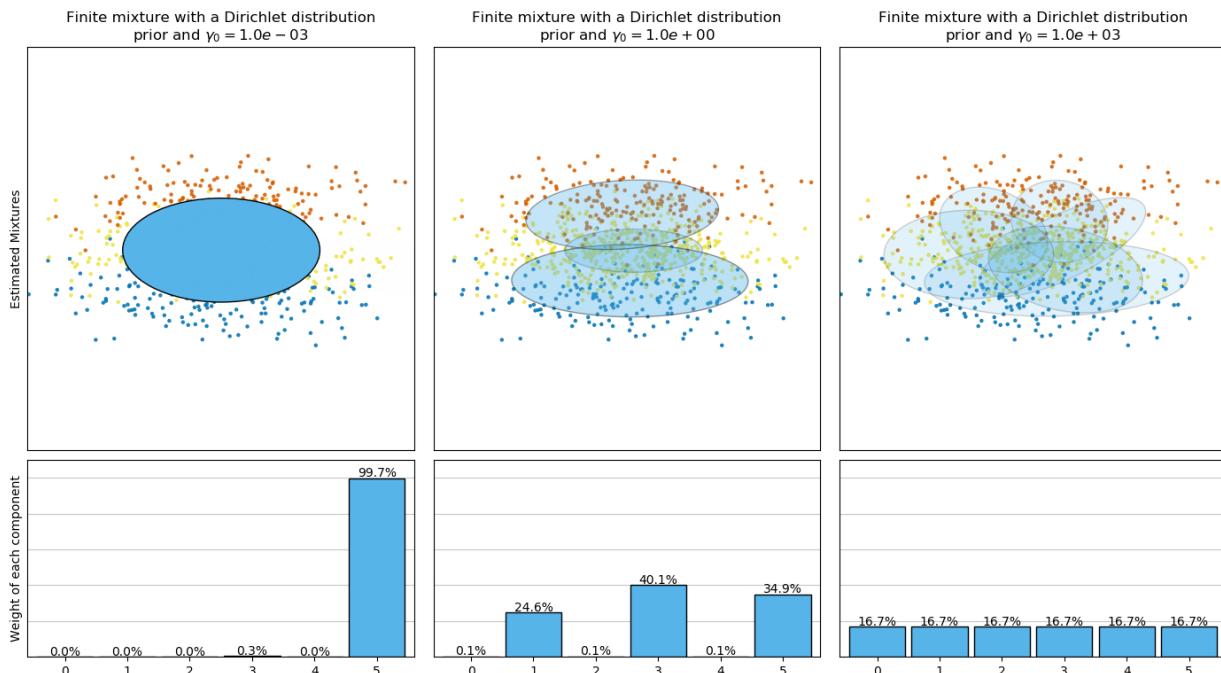
Variational inference is an extension of expectation-maximization that maximizes a lower bound on model evidence (including priors) instead of data likelihood. The principle behind variational methods is the same as expectation-maximization (that is both are iterative algorithms that alternate between finding the probabilities for each point to

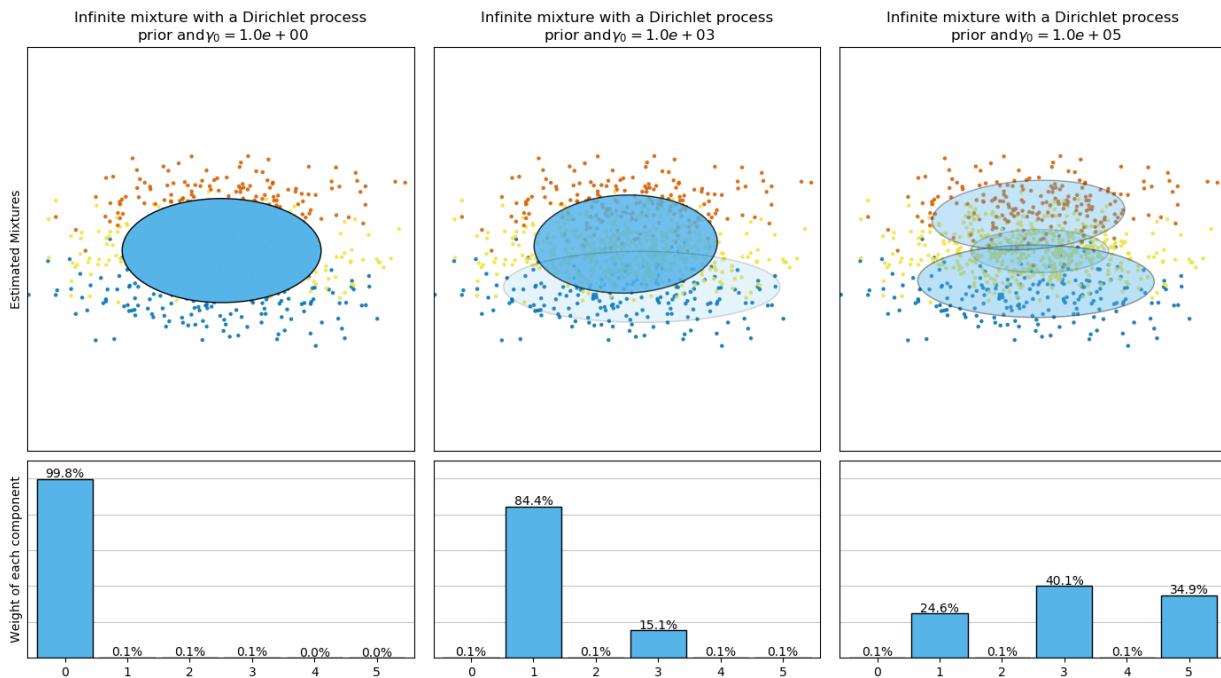
be generated by each mixture and fitting the mixture to these assigned points), but variational methods add regularization by integrating information from prior distributions. This avoids the singularities often found in expectation-maximization solutions but introduces some subtle biases to the model. 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 `weight_concentration_prior`. Specifying a low value for the concentration prior will make the model put most of the weight on few components set the remaining components weights very close to zero. High values of the concentration prior will allow a larger number of components to be active in the mixture.

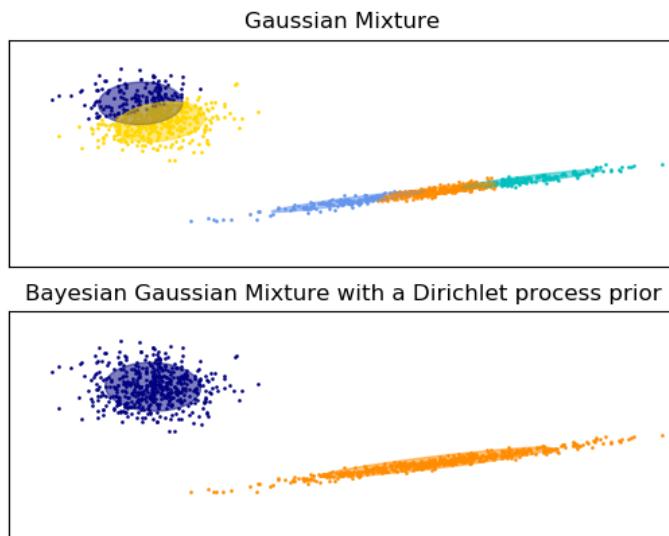
The parameters implementation of the `BayesianGaussianMixture` class proposes two types of prior for the weights distribution: a finite mixture model with Dirichlet distribution and an infinite mixture model with the Dirichlet Process. In practice Dirichlet Process inference algorithm is approximated and uses a truncated distribution with a fixed maximum number of components (called the Stick-breaking representation). The number of components actually used almost always depends on the data.

The next figure compares the results obtained for the different type of the weight concentration prior (parameter `weight_concentration_prior_type`) for different values of `weight_concentration_prior`. Here, we can see the value of the `weight_concentration_prior` parameter has a strong impact on the effective number of active components obtained. We can also notice that large values for the concentration weight prior lead to more uniform weights when the type of prior is ‘dirichlet_distribution’ while this is not necessarily the case for the ‘dirichlet_process’ type (used by default).



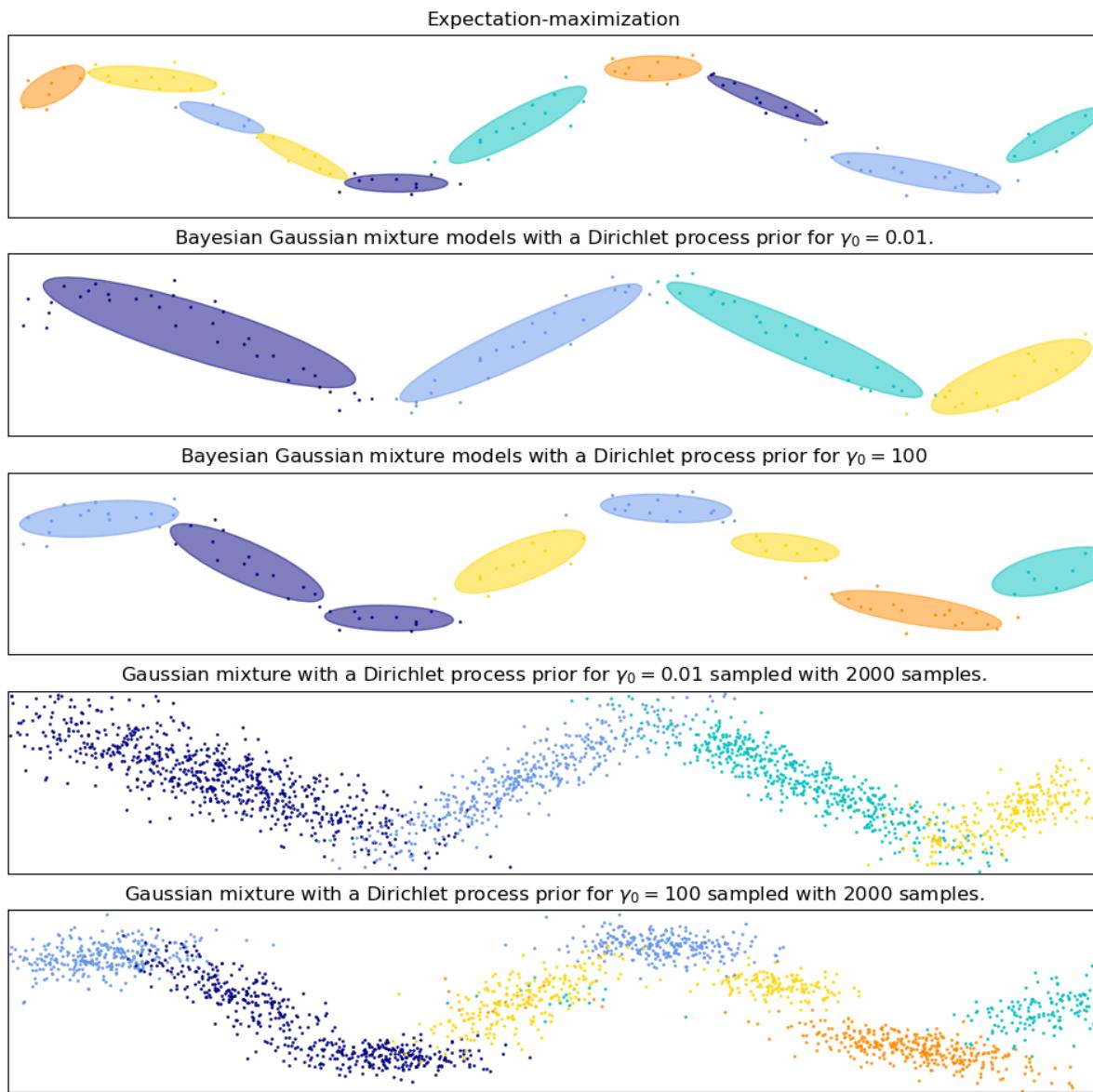


The examples below compare Gaussian mixture models with a fixed number of components, to the variational Gaussian mixture models with a Dirichlet process prior. Here, a classical Gaussian mixture is fitted with 5 components on a dataset composed of 2 clusters. We can see that the variational Gaussian mixture with a Dirichlet process prior is able to limit itself to only 2 components whereas the Gaussian mixture fits the data with a fixed number of components that has to be set a priori by the user. In this case the user has selected `n_components=5` which does not match the true generative distribution of this toy dataset. Note that with very little observations, the variational Gaussian mixture models with a Dirichlet process prior can take a conservative stand, and fit only one component.



On the following figure we are fitting a dataset not well-depicted by a Gaussian mixture. Adjusting the `weight_concentration_prior` parameter of the `BayesianGaussianMixture` controls the number of

components used to fit this data. We also present on the last two plots a random sampling generated from the two resulting mixtures.



Examples:

- See [Gaussian Mixture Model Ellipsoids](#) for an example on plotting the confidence ellipsoids for both `GaussianMixture` and `BayesianGaussianMixture`.
- [Gaussian Mixture Model Sine Curve](#) shows using `GaussianMixture` and `BayesianGaussianMixture` to fit a sine wave.
- See [Concentration Prior Type Analysis of Variation Bayesian Gaussian Mixture](#) for an example plotting the confidence ellipsoids for the `BayesianGaussianMixture` with dif-

ferent `weight_concentration_prior_type` for different values of the parameter `weight_concentration_prior`.

Pros and cons of variational inference with BayesianGaussianMixture

Pros

Automatic selection when `weight_concentration_prior` is small enough and `n_components` is larger than what is found necessary by the model, the Variational Bayesian mixture model has a natural tendency to set some mixture weights values close to zero. This makes it possible to let the model choose a suitable number of effective components automatically. Only an upper bound of this number needs to be provided. Note however that the “ideal” number of active components is very application specific and is typically ill-defined in a data exploration setting.

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 variational inference with a Dirichlet process prior (`weight_concentration_prior_type='dirichlet_process'`) won’t change much with changes to the parameters, leading to more stability and less tuning.

Regularization due to the incorporation of prior information, variational solutions have less pathological special cases than expectation-maximization solutions.

Cons

Speed the extra parametrization necessary for variational inference make inference slower, although not by much.

Hyperparameters this algorithm needs an extra hyperparameter that might need experimental tuning via cross-validation.

Bias there are many implicit biases in the inference algorithms (and also in the Dirichlet process if used), and whenever there is a mismatch between these biases and the data it might be possible to fit better models using a finite mixture.

The Dirichlet Process

Here we describe variational inference algorithms on Dirichlet process mixture. 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.

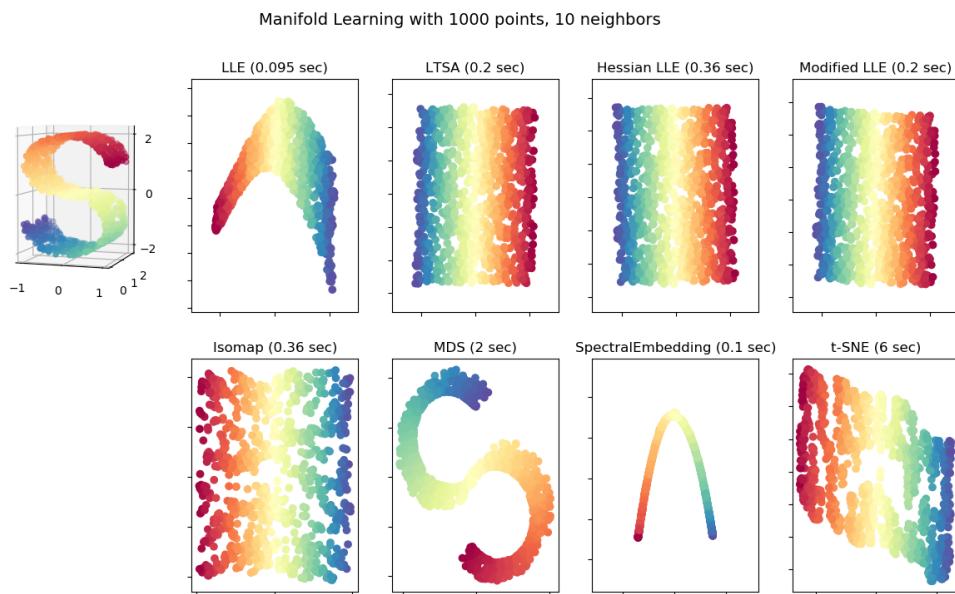
An important question is how can the Dirichlet process use an infinite, unbounded number of clusters and still be consistent. While a full explanation doesn’t fit this manual, one can think of its [stick breaking process](#) analogy to help understanding it. The stick breaking process is a generative story for the Dirichlet process. We start with a unit-length stick and in each step we break off a portion of the remaining stick. Each time, we associate the length of the piece of the stick to the proportion of points that falls into a group of the mixture. At the end, to represent the infinite mixture, we associate the last remaining piece of the stick to the proportion of points that don’t fall into all the other groups. The length of each piece is a random variable with probability proportional to the concentration parameter. Smaller value of the concentration will divide the unit-length into larger pieces of the stick (defining more concentrated distribution). Larger concentration values will create smaller pieces of the stick (increasing the number of components with non zero weights).

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).

3.2.2 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]

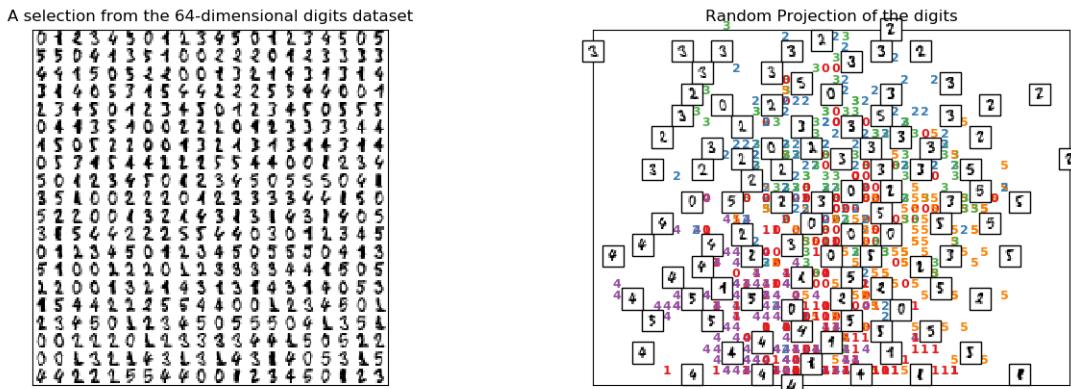


Manifold learning is an approach to non-linear dimensionality reduction. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high.

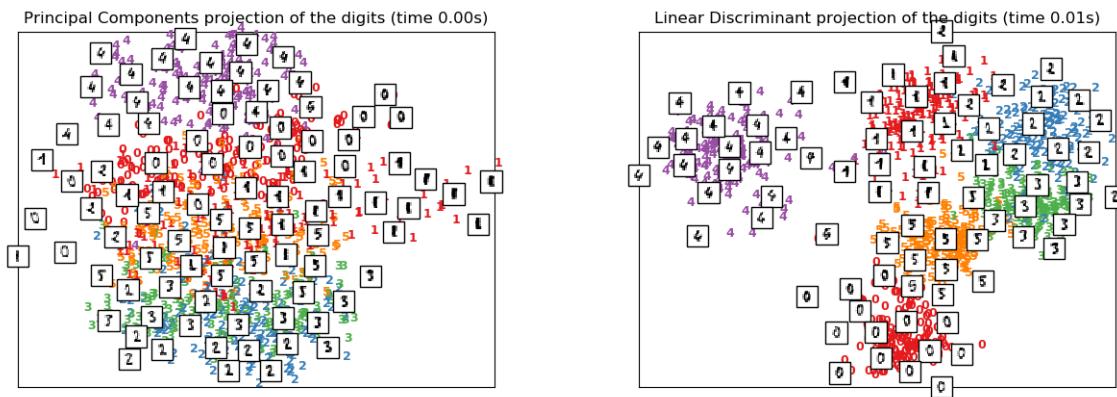
Introduction

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:

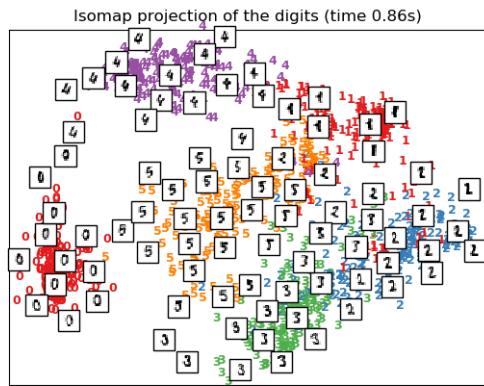
- See [Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...](#) for an example of dimensionality reduction on handwritten digits.
- See [Comparison of Manifold Learning methods](#) for an example of dimensionality reduction on a toy “S-curve” dataset.

The manifold learning implementations available in scikit-learn are summarized below

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](#).



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

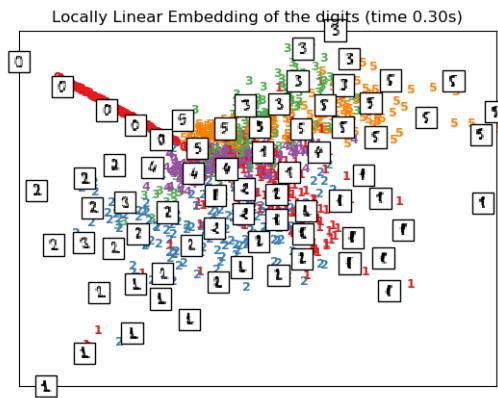
References:

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

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`.



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:

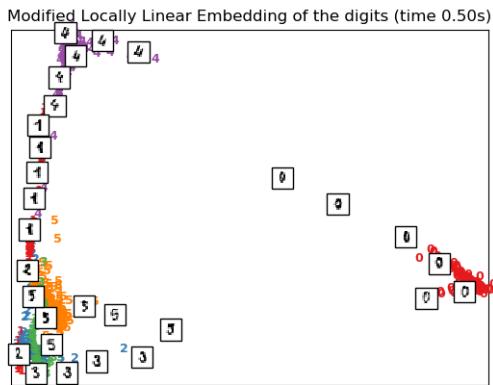
- “Nonlinear dimensionality reduction by locally linear embedding” Roweis, S. & Saul, L. Science 290:2323 (2000)

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 $r \rightarrow 0$, 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`.



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 $O[D \log(k)N \log(N)] + O[DNk^3] + O[N(k-D)k^2] + O[dN^2]$.

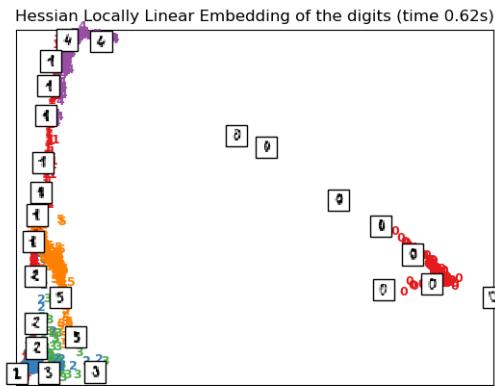
- N : number of training data points
- D : input dimension
- k : number of nearest neighbors
- d : output dimension

References:

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

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`.



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)

Spectral Embedding

Spectral Embedding is an approach to calculating a non-linear embedding. Scikit-learn implements Laplacian Eigenmaps, which 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`.

Complexity

The Spectral Embedding (Laplacian Eigenmaps) 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

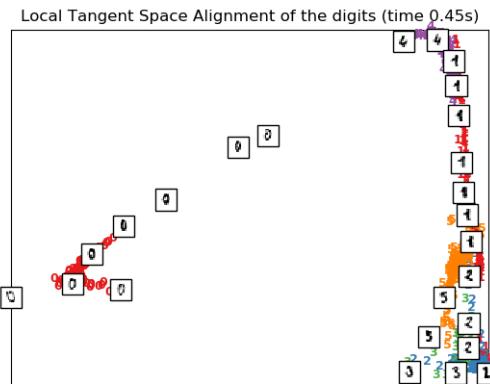
Local Tangent Space Alignment

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'`.

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)

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.

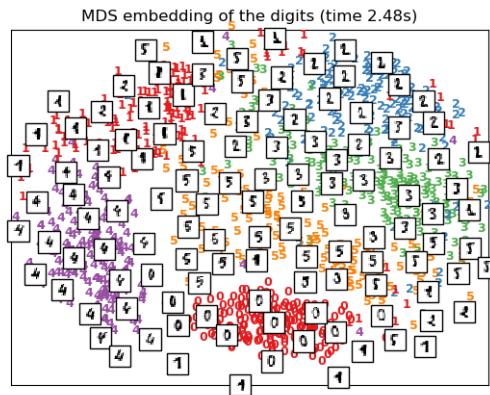
In general, is a technique used for analyzing similarity or dissimilarity data. [MDS](#) attempts to model similarity or dissimilarity data as distances in a geometric spaces. The data can be ratings of similarity between objects, interaction frequencies of molecules, or trade indices between countries.

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 $\sum_{i < j} d_{ij}(X) - \hat{d}_{ij}(X)$

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.

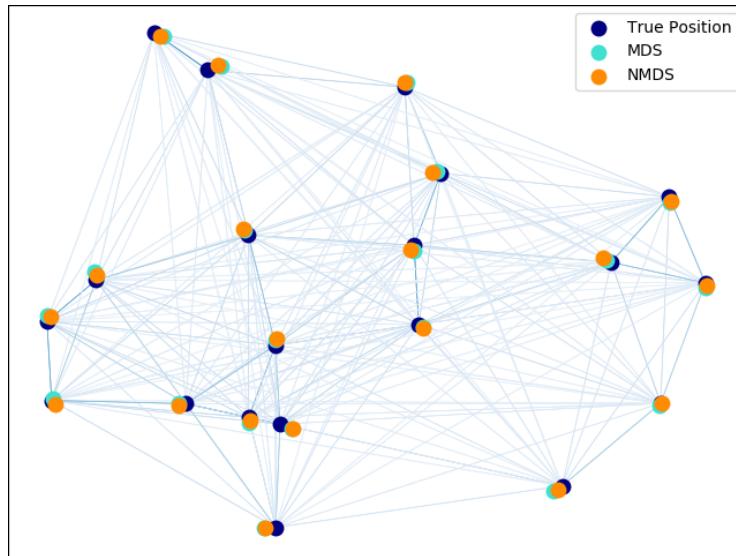


Most commonly, disparities are set to $\hat{d}_{ij} = bS_{ij}$.

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)

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. This allows t-SNE to be particularly sensitive to local structure and has a few other advantages over existing techniques:

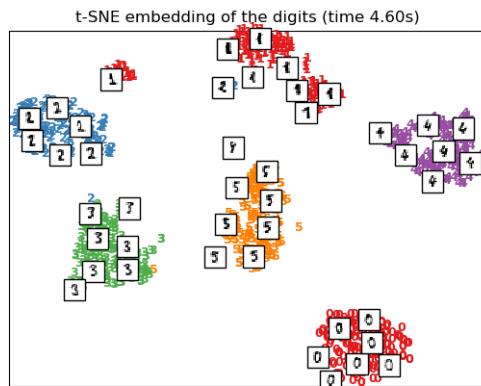
- Revealing the structure at many scales on a single map
- Revealing data that lie in multiple, different, manifolds or clusters
- Reducing the tendency to crowd points together at the center

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.

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 disadvantages to using t-SNE are roughly:

- t-SNE is computationally expensive, and can take several hours on million-sample datasets where PCA will finish in seconds or minutes
- The Barnes-Hut t-SNE method is limited to two or three dimensional embeddings.
- The algorithm is stochastic and multiple restarts with different seeds can yield different embeddings. However, it is perfectly legitimate to pick the embedding with the least error.
- Global structure is not explicitly preserved. This problem is mitigated by initializing points with PCA (using `init='pca'`).



Optimizing t-SNE

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 five parameters that control the optimization of t-SNE and therefore possibly the quality of the resulting embedding:

- perplexity
- early exaggeration factor
- learning rate
- maximum number of iterations
- angle (not used in the exact method)

The perplexity is defined as $k = 2^{(S)}$ where S is the Shannon entropy of the conditional probability distribution. The perplexity of a k -sided die is k , so that k is effectively the number of nearest neighbors t-SNE considers when generating the conditional probabilities. Larger perplexities lead to more nearest neighbors and less sensitive to small structure. Conversely a lower perplexity considers a smaller number of neighbors, and thus ignores more global information in favour of the local neighborhood. As dataset sizes get larger more points will be required to get a reasonable sample of the local neighborhood, and hence larger perplexities may be required. Similarly noisier datasets will require larger perplexity values to encompass enough local neighbors to see beyond the background noise.

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). The last parameter, angle, is a tradeoff between performance and accuracy. Larger angles imply that we can approximate larger regions by a single point, leading to better speed but less accurate results.

[“How to Use t-SNE Effectively”](#) provides a good discussion of the effects of the various parameters, as well as interactive plots to explore the effects of different parameters.

Barnes-Hut t-SNE

The Barnes-Hut 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 $O[dN\log(N)]$, where d is the number of output dimensions and N is the number of samples. The Barnes-Hut method improves on the exact method where t-SNE complexity is $O[dN^2]$, but has several other notable differences:

- The Barnes-Hut implementation only works when the target dimensionality is 3 or less. The 2D case is typical when building visualizations.
- Barnes-Hut only works with dense input data. Sparse data matrices can only be embedded with the exact method or can be approximated by a dense low rank projection for instance using `sklearn.decomposition.TruncatedSVD`
- Barnes-Hut is an approximation of the exact method. The approximation is parameterized with the angle parameter, therefore the angle parameter is unused when method="exact"
- Barnes-Hut is significantly more scalable. Barnes-Hut can be used to embed hundred of thousands of data points while the exact method can handle thousands of samples before becoming computationally intractable

For visualization purpose (which is the main use case of t-SNE), using the Barnes-Hut method is strongly recommended. The exact t-SNE method is useful for checking the theoretically properties of the embedding possibly in higher dimensional space but limit to small datasets due to computational constraints.

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 imply that the data cannot be correctly classified by a supervised model. It might be the case that 2 dimensions are not low enough 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.
- “Accelerating t-SNE using Tree-Based Algorithms.” L.J.P. van der Maaten. Journal of Machine Learning Research 15(Oct):3221-3245, 2014.

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.

3.2.3 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 can take different kinds of matrix as input. All the methods accept standard data matrices of shape [n_samples, n_features]. These can be obtained from the classes in the `sklearn.feature_extraction` module. For `AffinityPropagation`, `SpectralClustering` and `DBSCAN` one can also input similarity matrices of shape [n_samples, n_samples]. These can be obtained from the functions in the `sklearn.metrics.pairwise` module.

Overview of clustering methods

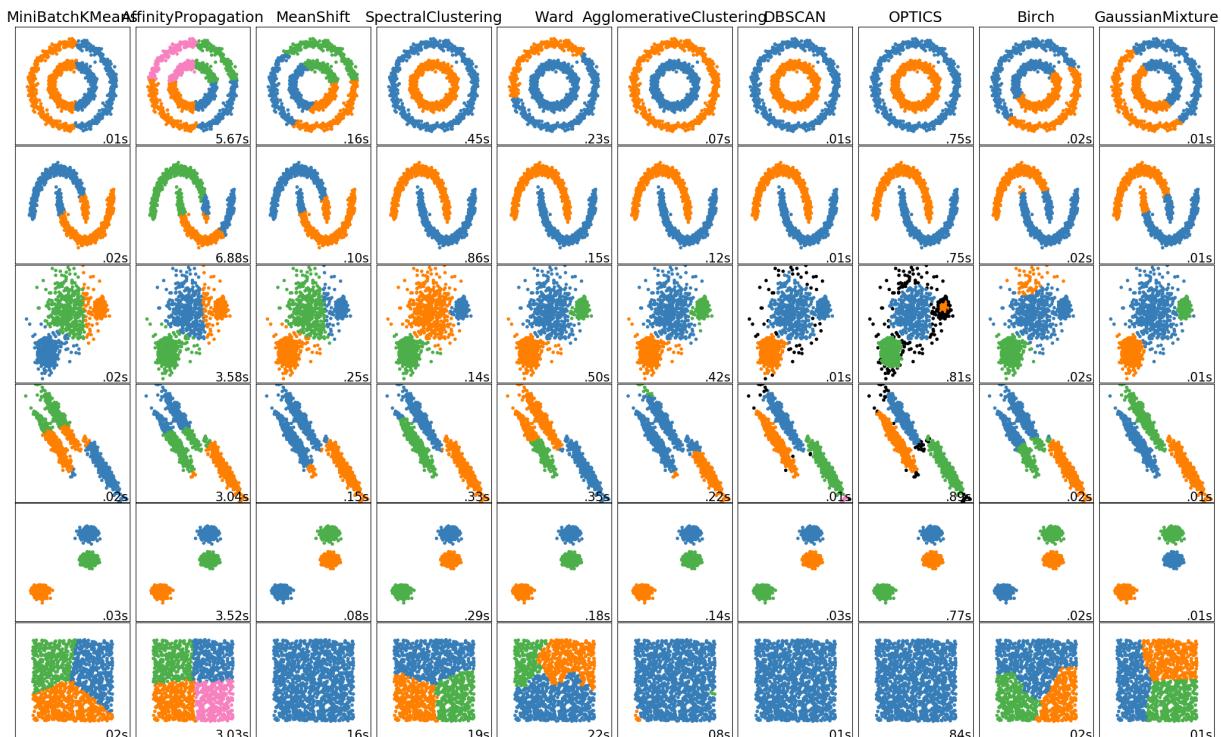


Fig. 3.4: A comparison of the clustering algorithms in scikit-learn

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 or distance threshold	Large n_samples and n_clusters	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters or distance threshold, 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
OPTICS	minimum cluster membership	Very large n_samples, large n_clusters	Non-flat geometry, uneven cluster sizes, variable cluster density	Distances between points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large n_clusters and n_samples	Large dataset, outlier removal, data reduction.	Euclidean distance between points

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.

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* or within-cluster sum-of-squares (see below). 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,

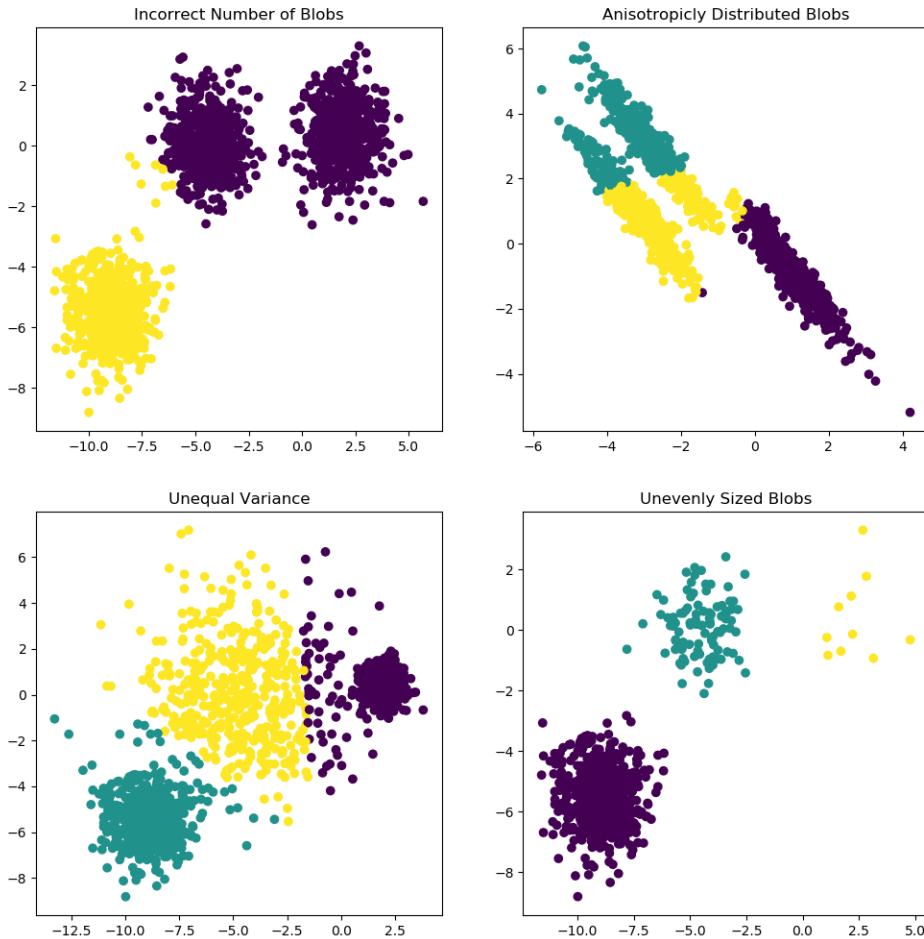
points from X , although they live in the same space.

The K-means algorithm aims to choose centroids that minimise the **inertia**, or **within-cluster sum-of-squares criterion**:

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

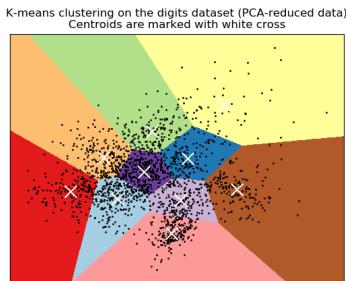
Inertia can be recognized as a measure of how internally coherent clusters are. It suffers from various drawbacks:

- 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 *Principal component analysis (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='k-means++'` 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.

The algorithm supports sample weights, which can be given by a parameter `sample_weight`. This allows to assign more weight to some samples when computing cluster centers and values of inertia. For example, assigning a weight of 2 to a sample is equivalent to adding a duplicate of that sample to the dataset X .

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:

- [*Demonstration of k-means assumptions*](#): Demonstrating when k-means performs intuitively and when it does not
- [*A demo of K-Means clustering on the handwritten digits data*](#): Clustering handwritten digits

References:

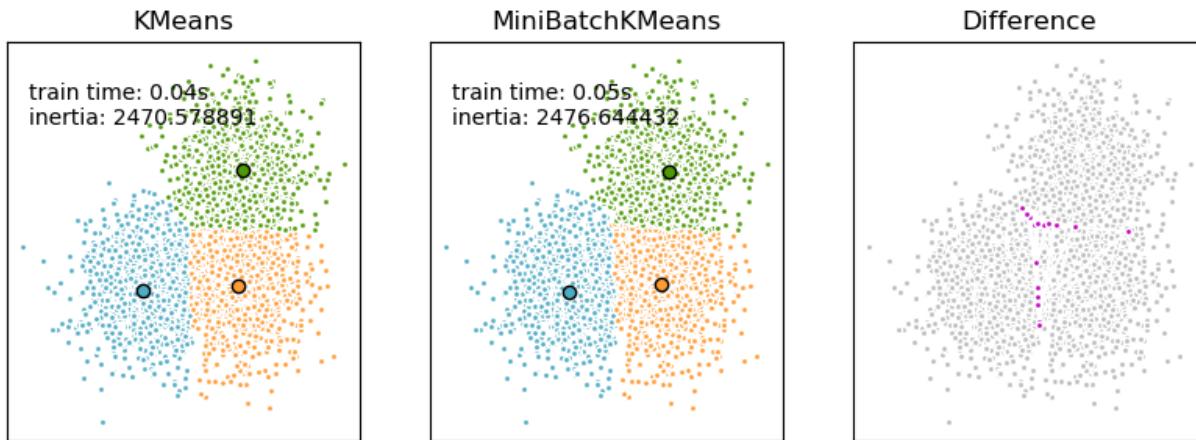
- “[k-means++: The advantages of careful seeding](#)” Arthur, David, and Sergei Vassilvitskii, *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics (2007)

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:**

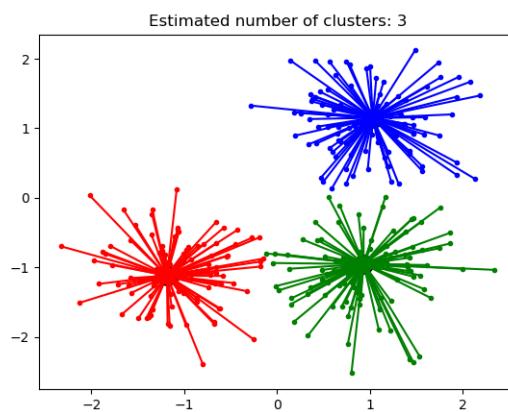
- [Comparison of the K-Means and MiniBatchKMeans clustering algorithms](#): Comparison of KMeans and MiniBatchKMeans
- [Clustering text documents using k-means](#): Document clustering using sparse MiniBatchKMeans
- [Online learning of a dictionary of parts of faces](#)

References:

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

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* which damps the responsibility and availability messages to avoid numerical oscillations when updating these messages.

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:

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

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:

$$a(i, k) \leftarrow \min[0, r(k, k) + \sum_{i' \text{ s.t. } i' \notin \{i, k\}} r(i', k)]$$

To begin with, all values for r and a are set to zero, and the calculation of each iterates until convergence. As discussed above, in order to avoid numerical oscillations when updating the messages, the damping factor λ is introduced to iteration process:

$$r_{t+1}(i, k) = \lambda \cdot r_t(i, k) + (1 - \lambda) \cdot r_{t+1}(i, k)$$

$$a_{t+1}(i, k) = \lambda \cdot a_t(i, k) + (1 - \lambda) \cdot a_{t+1}(i, k)$$

where t indicates the iteration times.

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:

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

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:

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

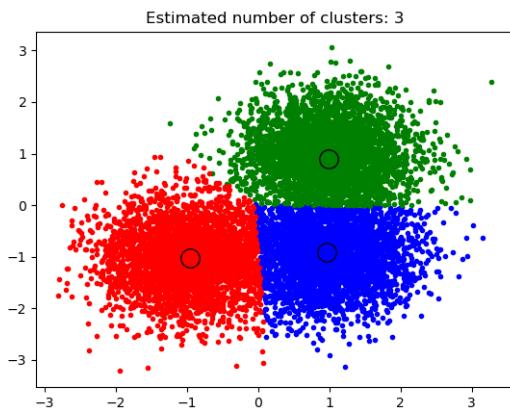
The algorithm automatically sets the number of clusters, instead of relying on a parameter `bandwidth`, which dictates the size of the region to search through. This parameter can be set manually, but can be estimated using the provided `estimate_bandwidth` function, which is called if the bandwidth 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.



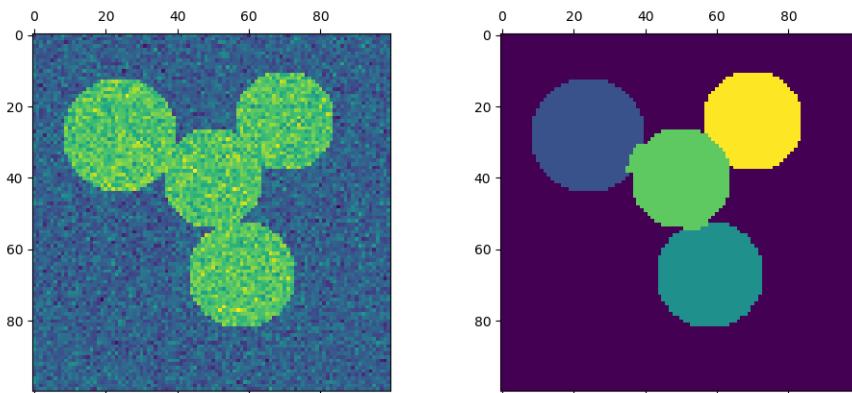
References:

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

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())
```

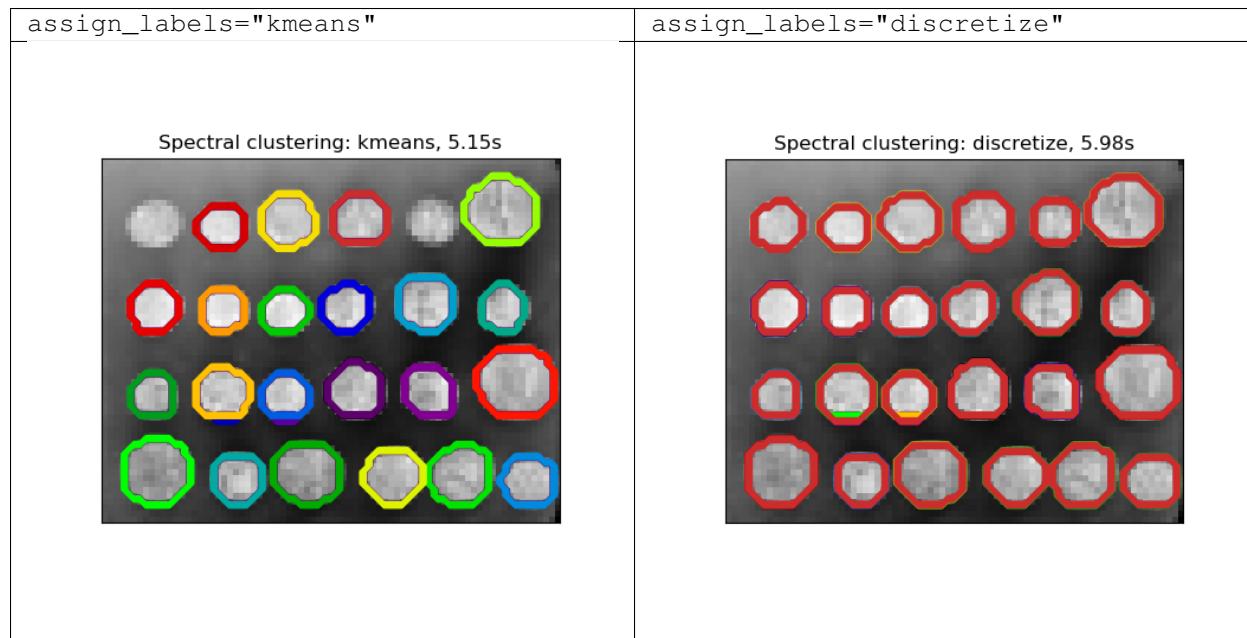
See the examples for such an application.

Examples:

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

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.



Spectral Clustering Graphs

Spectral Clustering can also be used to cluster graphs by their spectral embeddings. In this case, the affinity matrix is the adjacency matrix of the graph, and `SpectralClustering` is initialized with `affinity='precomputed'`:

```
>>> from sklearn.cluster import SpectralClustering
>>> sc = SpectralClustering(3, affinity='precomputed', n_init=100,
...                         assign_labels='discretize')
>>> sc.fit_predict(adjacency_matrix)
```

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

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.
- **Single linkage** minimizes the distance between the closest observations of pairs of clusters.

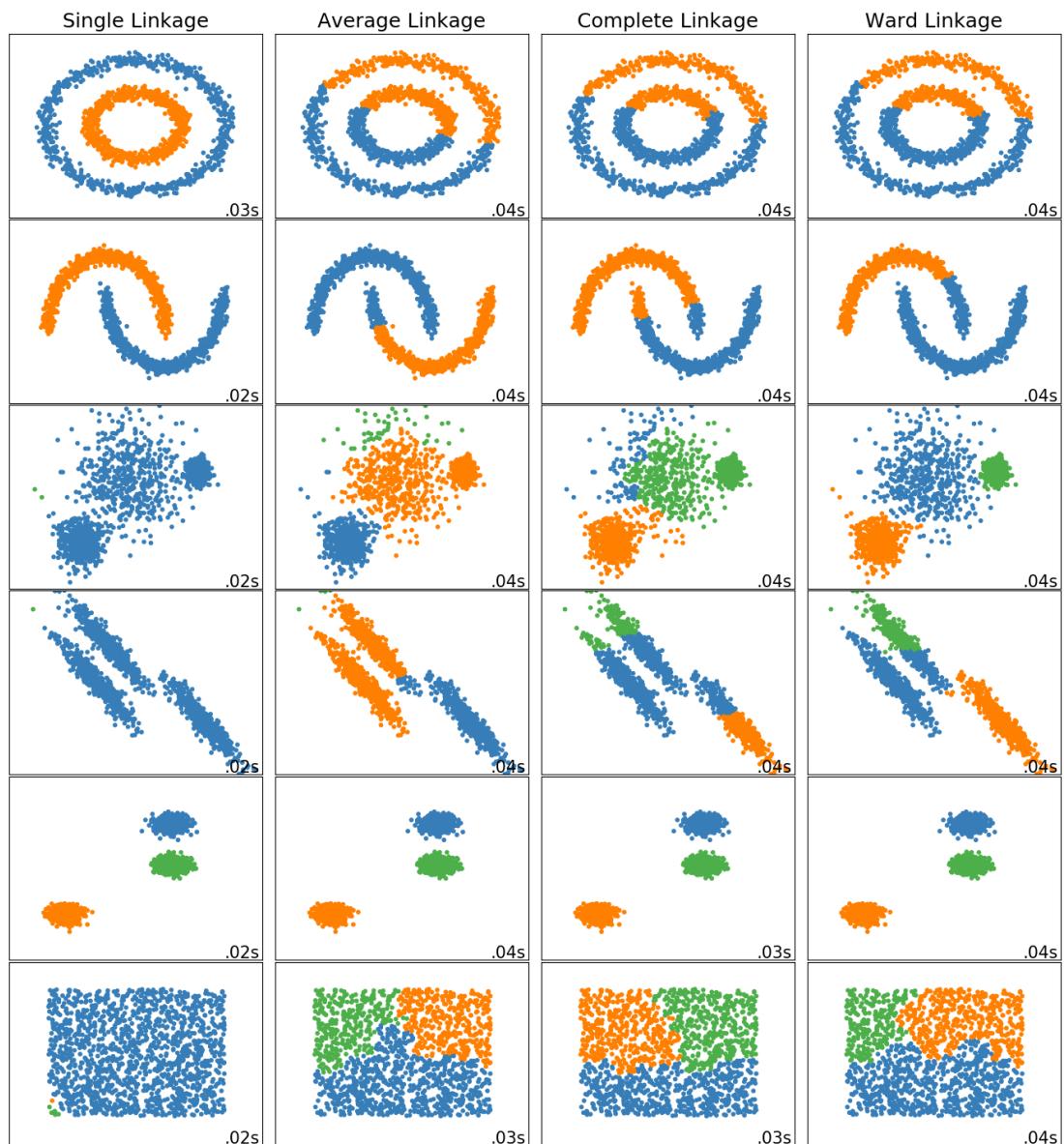
`AgglomerativeClustering` can also scale to large number of samples when it is used jointly with a connectivity matrix, but is computationally expensive when no connectivity constraints are added between samples: it considers at each step all the possible merges.

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 dimensionality reduction](#).

Different linkage type: Ward, complete, average, and single linkage

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



Ag-

glomerative cluster has a “rich get richer” behavior that leads to uneven cluster sizes. In this regard, single 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. Single linkage, while not robust to noisy data, can be computed very efficiently and can therefore be useful to provide hierarchical clustering of larger datasets. Single linkage can also perform well on non-globular data.

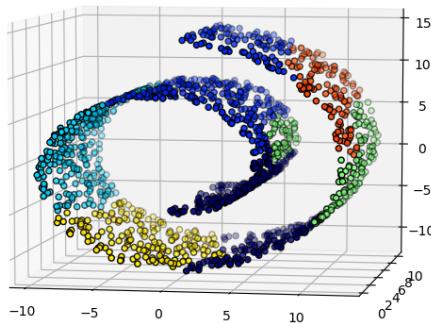
Examples:

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

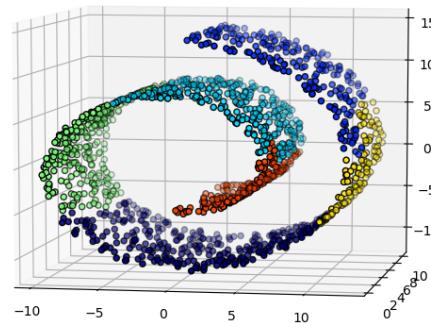
Adding connectivity constraints

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.

Without connectivity constraints (time 0.08s)



With connectivity constraints (time 0.07s)



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.

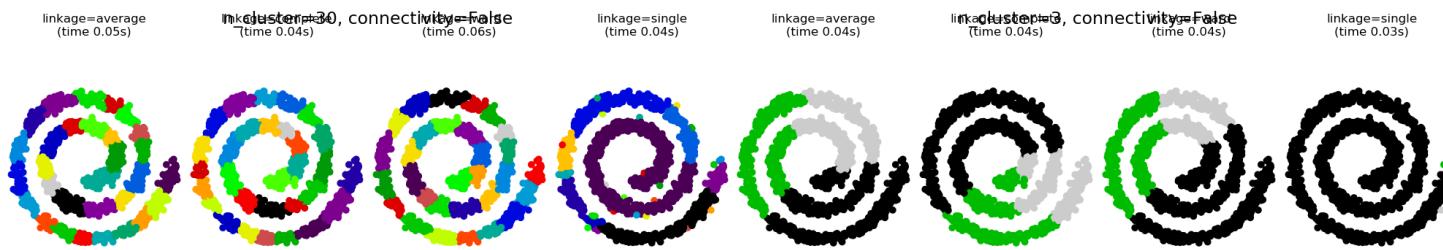
The connectivity constraints are imposed via an connectivity matrix: a scipy sparse matrix that has elements only at the intersection of a row and a column with indices of the dataset that should be connected. This matrix can be constructed from a-priori information: for instance, you may wish to cluster 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 [coin](#) example.

Examples:

- [A demo of structured Ward hierarchical clustering on an image of coins](#): Ward clustering to split the image of coins 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 single, average and complete linkage

Connectivity constraints and single, 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](#)). Single linkage is the most brittle linkage option with regard to this issue.

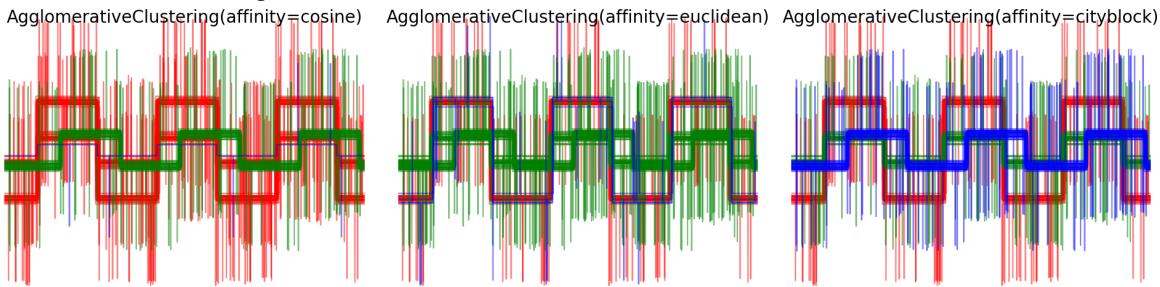


Varying the metric

Single, 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: i.e. many of the features are zero, as in text mining using occurrences 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*

DBSCAN

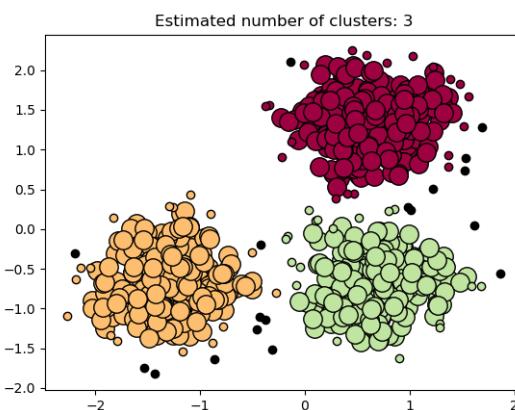
The [DBSCAN](#) algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to k-means which assumes that clusters are convex shaped. The central component to the DBSCAN is the concept of *core 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 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. Any sample that is not a core sample, and is at least `eps` in distance from any core sample, is considered an outlier by the algorithm.

While the parameter `min_samples` primarily controls how tolerant the algorithm is towards noise (on noisy and large data sets it may be desirable to increase this parameter), the parameter `eps` is *crucial to choose appropriately* for the data set and distance function and usually cannot be left at the default value. It controls the local neighborhood of the points. When chosen too small, most data will not be clustered at all (and labeled as `-1` for “noise”). When chosen too large, it causes close clusters to be merged into one cluster, and eventually the entire data set to be returned as a single cluster. Some heuristics for choosing this parameter have been discussed in literature, for example based on a knee in the nearest neighbor distances plot (as discussed in the references below).

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 DBSCAN algorithm is deterministic, always generating the same clusters when given the same data in the same order. However, the results can differ when data is provided in a different order. First, even though the core samples will always be assigned to the same clusters, the labels of those clusters will depend on the order in which those samples are encountered in the data. Second and more importantly, the clusters to which non-core samples are assigned can differ depending on the data order. This would happen when a non-core sample has 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 in a pass through the data, and so the results will depend on the data ordering.

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`.

Memory consumption for large sample sizes

This implementation is by default not memory efficient because it constructs a full pairwise similarity matrix in the case where kd-trees or ball-trees cannot be used (e.g., with sparse matrices). This matrix will consume n^2 floats. A couple of mechanisms for getting around this are:

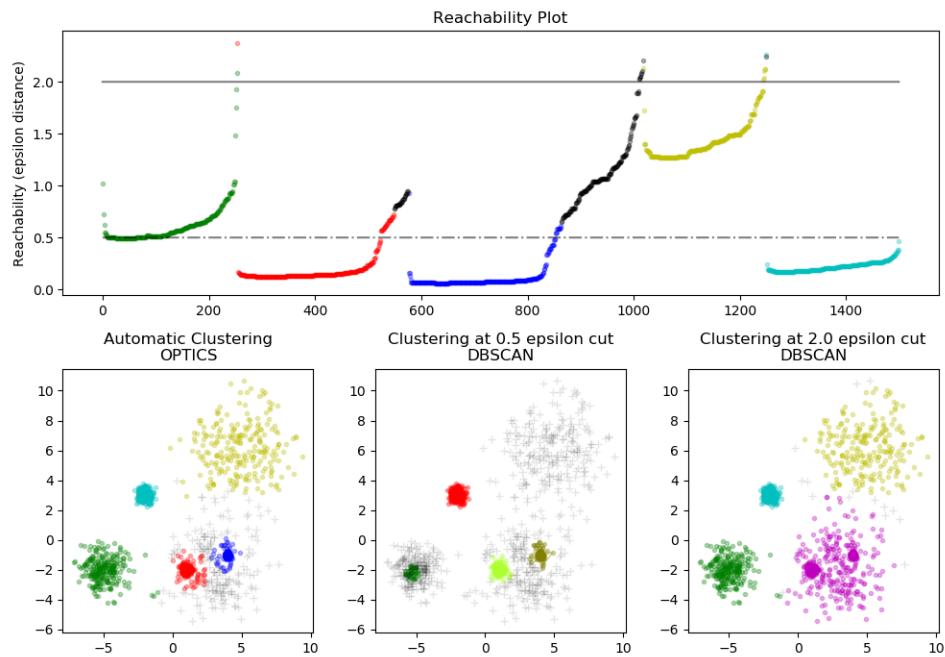
- Use [OPTICS](#) clustering in conjunction with the `extract_dbSCAN` method. OPTICS clustering also calculates the full pairwise matrix, but only keeps one row in memory at a time (memory complexity n).
- A sparse radius neighborhood graph (where missing entries are presumed to be out of `eps`) can be precomputed in a memory-efficient way and `dbSCAN` can be run over this with `metric='precomputed'`. See `sklearn.neighbors.NearestNeighbors.radius_neighbors_graph`.
- The dataset can be compressed, either by removing exact duplicates if these occur in your data, or by using BIRCH. Then you only have a relatively small number of representatives for a large number of points. You can then provide a `sample_weight` when fitting DBSCAN.

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
- “DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017). In ACM Transactions on Database Systems (TODS), 42(3), 19.

OPTICS

The [OPTICS](#) algorithm shares many similarities with the [DBSCAN](#) algorithm, and can be considered a generalization of DBSCAN that relaxes the `eps` requirement from a single value to a value range. The key difference between DBSCAN and OPTICS is that the OPTICS algorithm builds a *reachability* graph, which assigns each sample both a `reachability_distance`, and a spot within the cluster `ordering_attribute`; these two attributes are assigned when the model is fitted, and are used to determine cluster membership. If OPTICS is run with the default value of `inf` set for `max_eps`, then DBSCAN style cluster extraction can be performed repeatedly in linear time for any given `eps` value using the `cluster_optics_dbSCAN` method. Setting `max_eps` to a lower value will result in shorter run times, and can be thought of as the maximum neighborhood radius from each point to find other potential reachable points.



The *reachability* distances generated by OPTICS allow for variable density extraction of clusters within a single data set. As shown in the above plot, combining *reachability* distances and data set `ordering_` produces a *reachability plot*, where point density is represented on the Y-axis, and points are ordered such that nearby points are adjacent. ‘Cutting’ the reachability plot at a single value produces DBSCAN like results; all points above the ‘cut’ are classified as noise, and each time that there is a break when reading from left to right signifies a new cluster. The default cluster extraction with OPTICS looks at the steep slopes within the graph to find clusters, and the user can define what counts as a steep slope using the parameter `xi`. There are also other possibilities for analysis on the graph itself, such as generating hierarchical representations of the data through reachability-plot dendograms, and the hierarchy of clusters detected by the algorithm can be accessed through the `cluster_hierarchy_` parameter. The plot above has been color-coded so that cluster colors in planar space match the linear segment clusters of the reachability plot. Note that the blue and red clusters are adjacent in the reachability plot, and can be hierarchically represented as children of a larger parent cluster.

Examples:

- *Demo of OPTICS clustering algorithm*

Comparison with DBSCAN

The results from OPTICS `cluster_optics_dbSCAN` method and DBSCAN are very similar, but not always identical; specifically, labeling of periphery and noise points. This is in part because the first samples of each dense area processed by OPTICS have a large reachability value while being close to other points in their area, and will thus sometimes be marked as noise rather than periphery. This affects adjacent points when they are considered as candidates for being marked as either periphery or noise.

Note that for any single value of `eps`, DBSCAN will tend to have a shorter run time than OPTICS; however, for repeated runs at varying `eps` values, a single run of OPTICS may require less cumulative runtime than DBSCAN. It is also important to note that OPTICS’ output is close to DBSCAN’s only if `eps` and `max_eps` are close.

Computational Complexity

Spatial indexing trees are used to avoid calculating the full distance matrix, and allow for efficient memory usage on large sets of samples. Different distance metrics can be supplied via the `metric` keyword.

For large datasets, similar (but not identical) results can be obtained via [HDBSCAN](#). The HDBSCAN implementation is multithreaded, and has better algorithmic runtime complexity than OPTICS, at the cost of worse memory scaling. For extremely large datasets that exhaust system memory using HDBSCAN, OPTICS will maintain n (as opposed to n^2) memory scaling; however, tuning of the `max_eps` parameter will likely need to be used to give a solution in a reasonable amount of wall time.

References:

- “OPTICS: ordering points to identify the clustering structure.” Ankerst, Mihael, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. In ACM Sigmod Record, vol. 28, no. 2, pp. 49-60. ACM, 1999.

Birch

The [Birch](#) builds a tree called the Characteristic Feature Tree (CFT) for the given data. The data is essentially lossy compressed to a set of Characteristic Feature nodes (CF Nodes). The CF Nodes have a number of subclusters called Characteristic Feature subclusters (CF Subclusters) and these CF Subclusters located in the non-terminal CF Nodes can have CF Nodes as children.

The CF Subclusters hold the necessary information for clustering which prevents the need to hold the entire input data in memory. This information includes:

- Number of samples in a subcluster.
- Linear Sum - A n-dimensional vector holding the sum of all samples
- Squared Sum - Sum of the squared L2 norm of all samples.
- Centroids - To avoid recalculation linear sum / $n_{samples}$.
- Squared norm of the centroids.

The Birch algorithm has two parameters, the threshold and the branching factor. The branching factor limits the number of subclusters in a node and the threshold limits the distance between the entering sample and the existing subclusters.

This algorithm can be viewed as an instance or data reduction method, since it reduces the input data to a set of subclusters which are obtained directly from the leaves of the CFT. This reduced data can be further processed by feeding it into a global clusterer. This global clusterer can be set by `n_clusters`. If `n_clusters` is set to None, the subclusters from the leaves are directly read off, otherwise a global clustering step labels these subclusters into global clusters (labels) and the samples are mapped to the global label of the nearest subcluster.

Algorithm description:

- A new sample is inserted into the root of the CF Tree which is a CF Node. It is then merged with the subcluster of the root, that has the smallest radius after merging, constrained by the threshold and branching factor conditions. If the subcluster has any child node, then this is done repeatedly till it reaches a leaf. After finding the nearest subcluster in the leaf, the properties of this subcluster and the parent subclusters are recursively updated.
- If the radius of the subcluster obtained by merging the new sample and the nearest subcluster is greater than the square of the threshold and if the number of subclusters is greater than the branching factor, then a space is temporarily allocated to this new sample. The two farthest subclusters are taken and the subclusters are divided into two groups on the basis of the distance between these subclusters.

- If this split node has a parent subcluster and there is room for a new subcluster, then the parent is split into two. If there is no room, then this node is again split into two and the process is continued recursively, till it reaches the root.

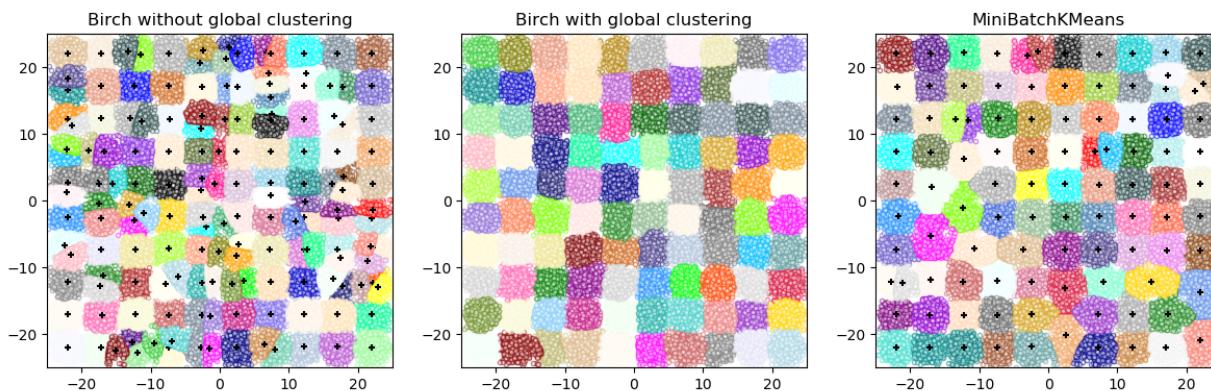
Birch or MiniBatchKMeans?

- Birch does not scale very well to high dimensional data. As a rule of thumb if `n_features` is greater than twenty, it is generally better to use MiniBatchKMeans.
- If the number of instances of data needs to be reduced, or if one wants a large number of subclusters either as a preprocessing step or otherwise, Birch is more useful than MiniBatchKMeans.

How to use `partial_fit`?

To avoid the computation of global clustering, for every call of `partial_fit` the user is advised

1. To set `n_clusters=None` initially
2. Train all data by multiple calls to `partial_fit`.
3. Set `n_clusters` to a required value using `brc.set_params(n_clusters=n_clusters)`.
4. Call `partial_fit` finally with no arguments, i.e. `brc.partial_fit()` which performs the global clustering.



References:

- Tian Zhang, Raghu Ramakrishnan, Maron Livny BIRCH: An efficient data clustering method for large databases. <https://www.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf>
- Roberto Perdisci Jbirch - Java implementation of BIRCH clustering algorithm <https://code.google.com/archive/p/jbirch>

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 than members of different classes according to some similarity metric.

Adjusted Rand index

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...
```

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.

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.

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:

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

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 $E[\text{RI}]$ of random labelings by defining the adjusted Rand index as follows:

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

References

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

Mutual Information based scores

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...
```

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).
- **Upper bound of 1**: Values close to zero indicate two label assignments that are largely independent, while values close to one indicate significant agreement. Further, an AMI of exactly 1 indicates that the two label assignments are equal (with or without permutation).

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.

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:

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

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 .

It also can be expressed in set cardinality formulation:

$$\text{MI}(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} \frac{|U_i \cap V_j|}{N} \log \left(\frac{N|U_i \cap V_j|}{|U_i||V_j|} \right)$$

The normalized mutual information is defined as

$$\text{NMI}(U, V) = \frac{\text{MI}(U, V)}{\text{mean}(H(U), H(V))}$$

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 [VEB2009]. 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).

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

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

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

For normalized mutual information and adjusted mutual information, the normalizing value is typically some *generalized* mean of the entropies of each clustering. Various generalized means exist, and no firm rules exist for preferring one over the others. The decision is largely a field-by-field basis; for instance, in community detection, the arithmetic mean is most common. Each normalizing method provides “qualitatively similar behaviours” [YAT2016]. In our implementation, this is controlled by the `average_method` parameter.

Vinh et al. (2010) named variants of NMI and AMI by their averaging method [VEB2010]. Their ‘sqrt’ and ‘sum’ averages are the geometric and arithmetic means; we use these more broadly common names.

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.
- Wikipedia entry for the (normalized) Mutual Information
- Wikipedia entry for the Adjusted Mutual Information

Homogeneity, completeness and V-measure

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...
```

This function's formula is as follows::

$$\dots \text{math}:: v = \frac{(1 + \beta) \times \text{homogeneity} \times \text{completeness}}{\beta \times \text{homogeneity} + \text{completeness}}$$

`beta` defaults to a value of 1.0, but for using a value less than 1 for beta:

```
>>> metrics.v_measure_score(labels_true, labels_pred, beta=0.6)
0.54...
```

more weight will be attributed to homogeneity, and using a value greater than 1:

```
>>> metrics.v_measure_score(labels_true, labels_pred, beta=1.8)
0.48...
```

more weight will be attributed to completeness.

The V-measure is actually equivalent to the mutual information (NMI) discussed above, with the aggregation function being the arithmetic mean [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)
```

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.

Drawbacks

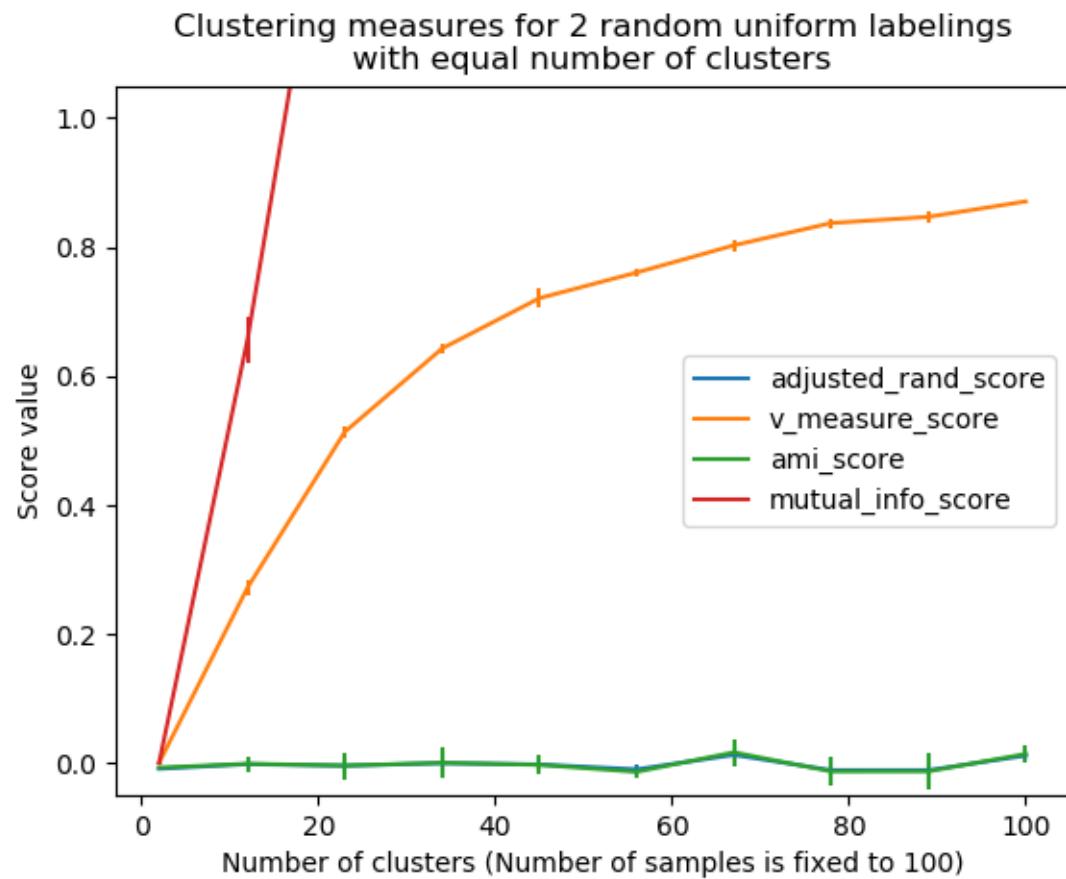
- The previously introduced metrics are **not normalized with regards to random labeling**: this means that depending on the number of samples, clusters and ground truth classes, a completely random labeling will not always yield the same values for homogeneity, completeness and hence v-measure. In particular **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.



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

- V-Measure: A conditional entropy-based external cluster evaluation measure Andrew Rosenberg and Julia Hirschberg, 2007

Fowlkes-Mallows scores

The Fowlkes-Mallows index (`sklearn.metrics.fowlkes_mallows_score`) can be used when the ground truth class assignments of the samples is known. The Fowlkes-Mallows score FMI is defined as the geometric mean of the pairwise precision and recall:

$$\text{FMI} = \frac{\text{TP}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})}}$$

Where TP is the number of **True Positive** (i.e. the number of pair of points that belong to the same clusters in both the true labels and the predicted labels), FP is the number of **False Positive** (i.e. the number of pair of points that belong to the same clusters in the true labels and not in the predicted labels) and FN is the number of **False Negative** (i.e the number of pair of points that belongs in the same clusters in the predicted labels and not in the true labels).

The score ranges from 0 to 1. A high value indicates a good similarity between two clusters.

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

```
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.47140...
```

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.fowlkes_mallows_score(labels_true, labels_pred)
0.47140...
```

Perfect labeling is scored 1.0:

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

Bad (e.g. independent labelings) have zero scores:

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.0
```

Advantages

- **Random (uniform) label assignments have a FMI 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).
- **Upper-bounded at 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 FMI 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.

Drawbacks

- Contrary to inertia, **FMI-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).

References

- E. B. Fowlkes and C. L. Mallows, 1983. “A method for comparing two hierarchical clusterings”. Journal of the American Statistical Association. <http://wildfire.stat.ucla.edu/pdflibrary/fowlkes.pdf>
- Wikipedia entry for the Fowlkes-Mallows Index

Silhouette Coefficient

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 Silhouette 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.

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.

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.

Examples:

- *Selecting the number of clusters with silhouette analysis on KMeans clustering* : In this example the silhouette analysis is used to choose an optimal value for n_clusters.

Calinski-Harabasz Index

If the ground truth labels are not known, the Calinski-Harabasz index (`sklearn.metrics.calinski_harabasz_score`) - also known as the Variance Ratio Criterion - can be used to evaluate the model, where a higher Calinski-Harabasz score relates to a model with better defined clusters.

For k clusters, the Calinski-Harabasz score s is given as the ratio of the between-clusters dispersion mean and the within-cluster dispersion:

$$s(k) = \frac{\text{Tr}(B_k)}{\text{Tr}(W_k)} \times \frac{N - k}{k - 1}$$

where B_K is the between group dispersion matrix and W_K is the within-cluster dispersion matrix defined by:

$$W_k = \sum_{q=1}^k \sum_{x \in C_q} (x - c_q)(x - c_q)^T$$

$$B_k = \sum_q n_q (c_q - c)(c_q - c)^T$$

with N be the number of points in our data, C_q be the set of points in cluster q , c_q be the center of cluster q , c be the center of E , n_q be the number of points in cluster q .

```
>>> 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 Calinski-Harabasz index 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.calinski_harabasz_score(X, labels)
561.62...
```

Advantages

- The score is higher when clusters are dense and well separated, which relates to a standard concept of a cluster.
- The score is fast to compute

Drawbacks

- The Calinski-Harabasz index is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained through DBSCAN.

References

- Caliński, T., & Harabasz, J. (1974). “A dendrite method for cluster analysis”. Communications in Statistics-theory and Methods 3: 1-27. doi:10.1080/03610926.2011.560741.

Davies-Bouldin Index

If the ground truth labels are not known, the Davies-Bouldin index (`sklearn.metrics.davies_bouldin_score`) can be used to evaluate the model, where a lower Davies-Bouldin index relates to a model with better separation between the clusters.

The index is defined as the average similarity between each cluster C_i for $i = 1, \dots, k$ and its most similar one C_j . In the context of this index, similarity is defined as a measure R_{ij} that trades off:

- s_i , the average distance between each point of cluster i and the centroid of that cluster – also know as cluster diameter.
- d_{ij} , the distance between cluster centroids i and j .

A simple choice to construct R_{ij} so that it is nonnegative and symmetric is:

$$R_{ij} = \frac{s_i + s_j}{d_{ij}}$$

Then the Davies-Bouldin index is defined as:

$$DB = \frac{1}{k} \sum_{i=1}^k \max_{i \neq j} R_{ij}$$

Zero is the lowest possible score. Values closer to zero indicate a better partition.

In normal usage, the Davies-Bouldin index is applied to the results of a cluster analysis as follows:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
>>> from sklearn.cluster import KMeans
>>> from sklearn.metrics import davies_bouldin_score
>>> kmeans = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans.labels_
>>> davies_bouldin_score(X, labels)
0.6619...
```

Advantages

- The computation of Davies-Bouldin is simpler than that of Silhouette scores.
- The index is computed only quantities and features inherent to the dataset.

Drawbacks

- The Davies-Boulding index is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained from DBSCAN.
- The usage of centroid distance limits the distance metric to Euclidean space.
- A good value reported by this method does not imply the best information retrieval.

References

- Davies, David L.; Bouldin, Donald W. (1979). “A Cluster Separation Measure” IEEE Transactions on Pattern Analysis and Machine Intelligence. PAMI-1 (2): 224-227. doi:10.1109/TPAMI.1979.4766909.
- Halkidi, Maria; Batistakis, Yannis; Vazirgiannis, Michalis (2001). “On Clustering Validation Techniques” Journal of Intelligent Information Systems, 17(2-3), 107-145. doi:10.1023/A:1012801612483.
- Wikipedia entry for Davies-Bouldin index.

Contingency Matrix

Contingency matrix (`sklearn.metrics.cluster.contingency_matrix`) reports the intersection cardinality for every true/predicted cluster pair. The contingency matrix provides sufficient statistics for all clustering metrics where the samples are independent and identically distributed and one doesn’t need to account for some instances not being clustered.

Here is an example:

```
>>> from sklearn.metrics.cluster import contingency_matrix
>>> x = ["a", "a", "a", "b", "b", "b"]
>>> y = [0, 0, 1, 1, 2, 2]
>>> contingency_matrix(x, y)
array([[2, 1, 0],
       [0, 1, 2]])
```

The first row of output array indicates that there are three samples whose true cluster is “a”. Of them, two are in predicted cluster 0, one is in 1, and none is in 2. And the second row indicates that there are three samples whose true cluster is “b”. Of them, none is in predicted cluster 0, one is in 1 and two are in 2.

A *confusion matrix* for classification is a square contingency matrix where the order of rows and columns correspond to a list of classes.

Advantages

- Allows to examine the spread of each true cluster across predicted clusters and vice versa.
- The contingency table calculated is typically utilized in the calculation of a similarity statistic (like the others listed in this document) between the two clusterings.

Drawbacks

- Contingency matrix is easy to interpret for a small number of clusters, but becomes very hard to interpret for a large number of clusters.

- It doesn't give a single metric to use as an objective for clustering optimisation.

References

- Wikipedia entry for contingency matrix

3.2.4 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:

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:

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.

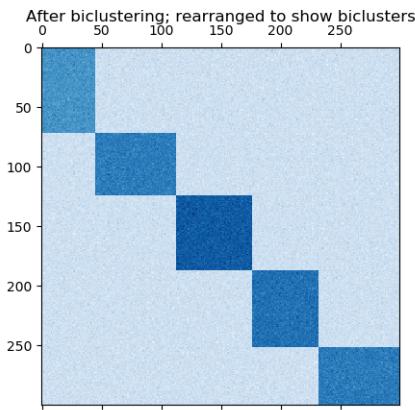


Fig. 3.5: An example of biclusters formed by partitioning rows and columns.

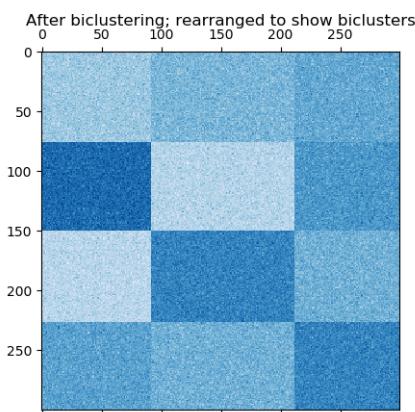


Fig. 3.6: An example of checkerboard biclusters.

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.

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 columns of U are $u_2, \dots, u_{\ell+1}$, 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](#).

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 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.

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. **Bistochasticization**: 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 $\overline{L_{\cdot i}}$, row mean $\overline{L_{\cdot j}}$, and overall mean $\overline{L_{..}}$ of L are computed. The final matrix is computed according to the formula

$$K_{ij} = L_{ij} - \overline{L_{\cdot i}} - \overline{L_{\cdot j}} + \overline{L_{..}}$$

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 bistochasticization 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.

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 do 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.

3.2.5 Decomposing signals in components (matrix factorization problems)

Principal component analysis (PCA)

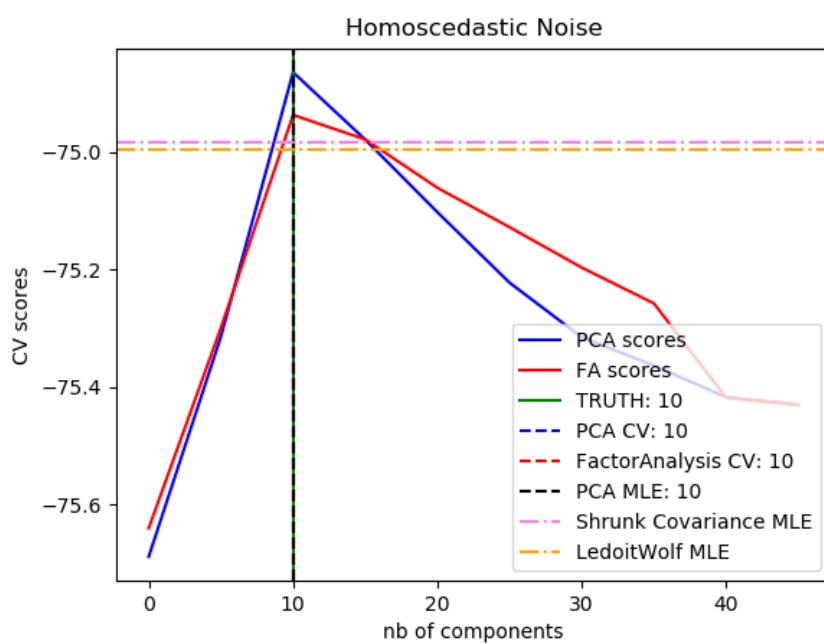
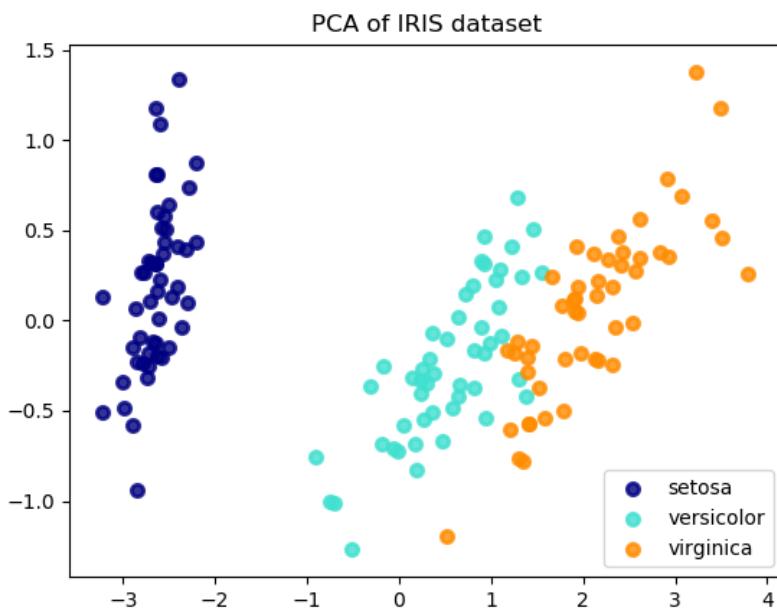
Exact PCA and probabilistic interpretation

PCA is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance. In 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.

PCA centers but does not scale the input data for each feature before applying the SVD. The optional parameter parameter `whiten=True` makes 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.

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)

Incremental PCA

The `PCA` object is very useful, but has certain limitations for large datasets. The biggest limitation is that `PCA` only supports batch processing, which means all of the data to be processed must fit in main memory. The `IncrementalPCA` object uses a different form of processing and allows for partial computations which almost exactly match the results of `PCA` while processing the data in a minibatch fashion. `IncrementalPCA` makes it possible to implement out-of-core Principal Component Analysis either by:

- Using its `partial_fit` method on chunks of data fetched sequentially from the local hard drive or a network database.
- Calling its `fit` method on a memory mapped file using `numpy.memmap`.

`IncrementalPCA` only stores estimates of component and noise variances, in order update `explained_variance_ratio_` incrementally. This is why memory usage depends on the number of samples per batch, rather than the number of samples to be processed in the dataset.

As in `PCA`, `IncrementalPCA` centers but does not scale the input data for each feature before applying the SVD.

Examples:

- Incremental PCA

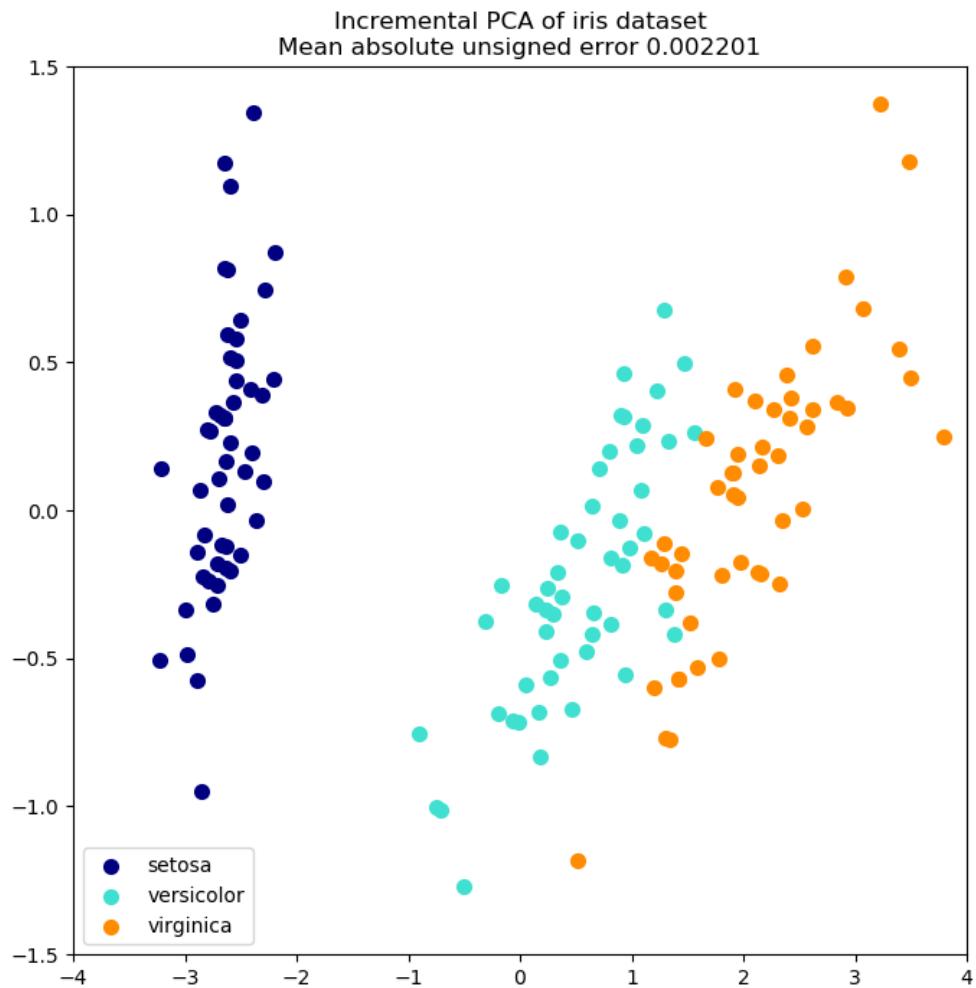
PCA using randomized SVD

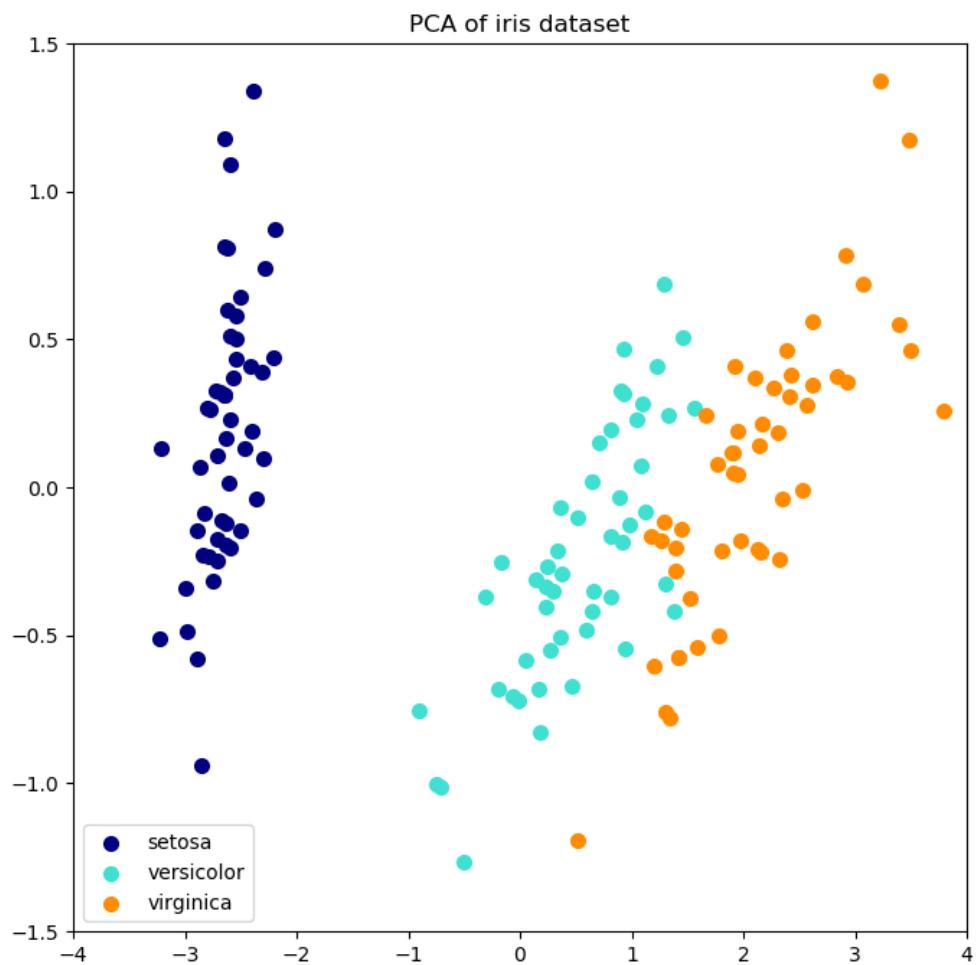
It is often interesting to project data to a lower-dimensional space that preserves most of the variance, by dropping the singular vector of components associated with lower singular values.

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 `PCA` used with the optional parameter `svd_solver='randomized'` 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 is less than 1s:





First centered Olivetti faces



genfaces - PCA using randomized SVD - Train time 0.0



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

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

Note: the implementation of `inverse_transform` in `PCA` with `svd_solver='randomized'` is not the exact inverse transform of `transform` even when `whiten=False` (default).

Examples:

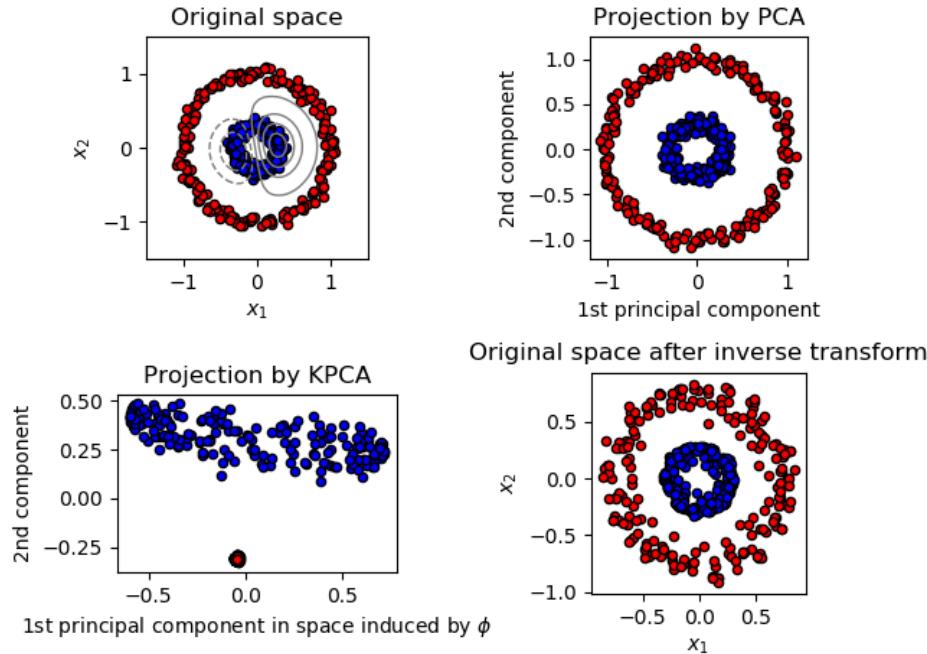
- *Faces recognition example using eigenfaces and SVMs*
- *Faces dataset decompositions*

References:

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

Kernel PCA

KernelPCA is an extension of PCA which achieves non-linear dimensionality reduction through the use of kernels (see [Pairwise metrics, Affinities and Kernels](#)). It has many applications including denoising, compression and structured prediction (kernel dependency estimation). *KernelPCA* supports both transform and inverse_transform.



Examples:

- [Kernel PCA](#)

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.

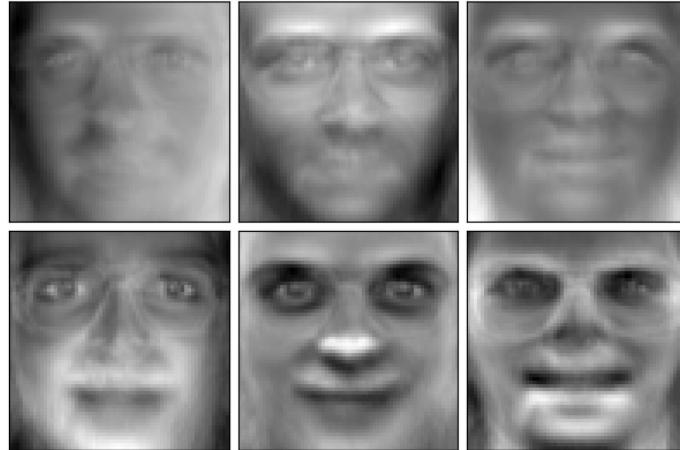
Principal component analysis ([PCA](#)) has the disadvantage that the components extracted by this method have exclusively dense expressions, i.e. they have non-zero coefficients when expressed as linear combinations of the original variables. This can make interpretation difficult. In many cases, the 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.

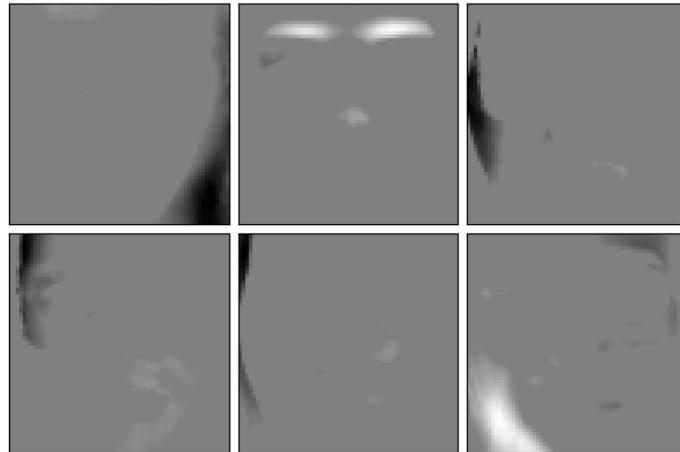
The following example illustrates 16 components extracted using sparse PCA from the Olivetti faces dataset. It can be seen how the regularization term induces many zeros. Furthermore, the natural structure of the data causes the non-zero coefficients to be vertically adjacent. The model does not enforce 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.

genfaces - PCA using randomized SVD - Train time 0.0



Sparse comp. - MiniBatchSparsePCA - Train time 1.1s



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:

$$(U^*, V^*) = \arg \min_{U, V} \frac{1}{2} \|X - UV\|_2^2 + \alpha \|V\|_1$$

subject to $\|U_k\|_2 = 1$ 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:**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 :

$$X' = X V_k$$

Note: Most treatments of LSA in the natural language processing (NLP) and information retrieval (IR) literature swap the axes of the matrix X so that it has shape `n_features` \times `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

Dictionary Learning

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 $2 * n_{\text{components}}$ and is constructed using the following rule: First, the regular code of length $n_{\text{components}}$ is computed. Then, the first $n_{\text{components}}$ 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*

Generic dictionary learning

Dictionary learning (`DictionaryLearning`) is a matrix factorization problem that amounts to finding a (usually overcomplete) dictionary that will perform well at sparsely encoding the fitted data.

Representing data as sparse combinations of atoms from an overcomplete dictionary is suggested to be the way the mammalian 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.

$$(U^*, V^*) = \arg \min_{U,V} \frac{1}{2} \|X - UV\|_2^2 + \alpha \|U\|_1$$

subject to $\|V_k\|_2 = 1$ for all $0 \leq k < n_{\text{atoms}}$

genfaces - PCA using randomized SVD - Train time 0.0



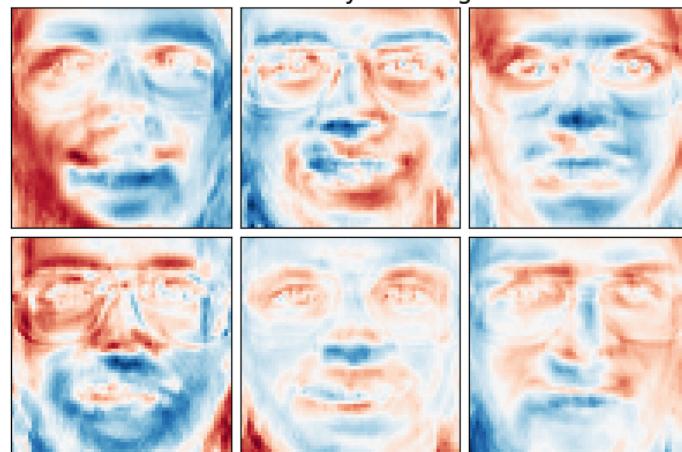
MiniBatchDictionaryLearning - Train time 1.0s



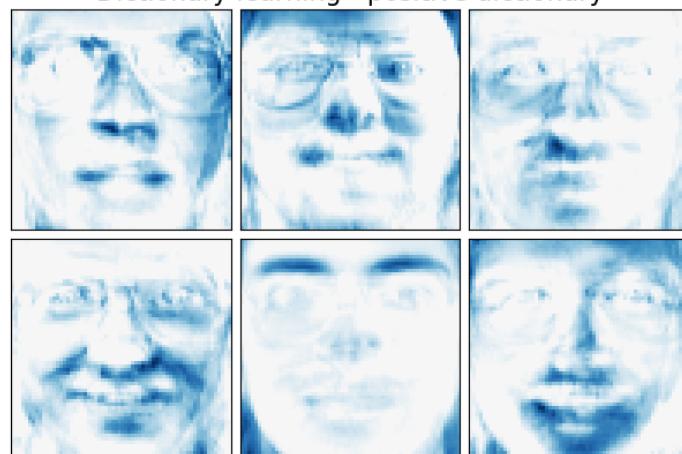
After using such a procedure to fit the dictionary, the transform is simply a sparse coding step that shares the same implementation with all dictionary learning objects (see [Sparse coding with a precomputed dictionary](#)).

It is also possible to constrain the dictionary and/or code to be positive to match constraints that may be present in the data. Below are the faces with different positivity constraints applied. Red indicates negative values, blue indicates positive values, and white represents zeros.

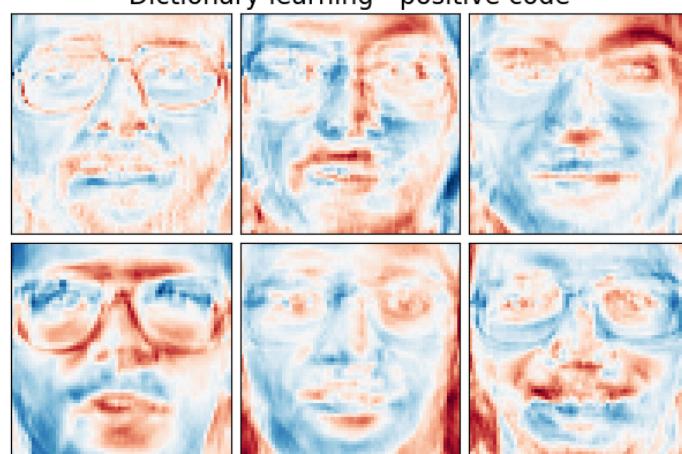
Dictionary learning



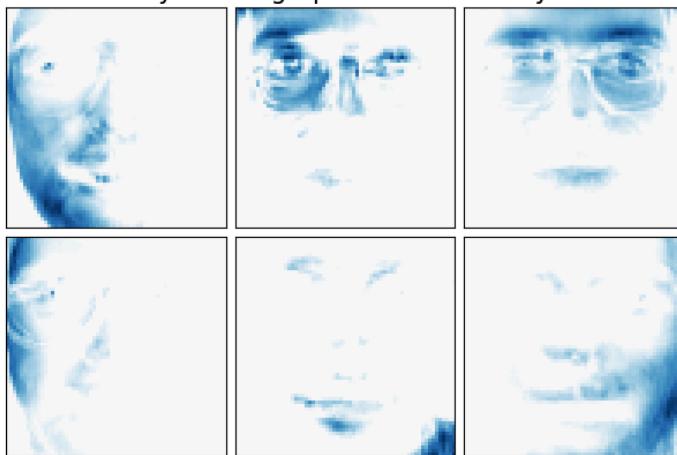
Dictionary learning - positive dictionary



Dictionary learning - positive code



Dictionary learning - positive dictionary & code



The following image shows how a dictionary learned from 4x4 pixel image patches extracted from part of the image of a raccoon face looks like.

Dictionary learned from face patches
Train time 7.2s on 22692 patches



Examples:

- *Image denoising using dictionary learning*

References:

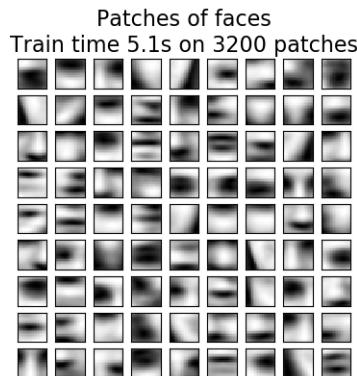
- “Online dictionary learning for sparse coding” J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009

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*

Factor Analysis

In unsupervised learning we only have a dataset $X = \{x_1, x_2, \dots, x_n\}$. How can this dataset be described mathematically? A very simple continuous latent variable model for X is

$$x_i = Wh_i + \mu + \epsilon$$

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}):

$$\mathbf{X} = \mathbf{W}\mathbf{H} + \mathbf{M} + \mathbf{E}$$

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](#).

- $\Psi = \text{diag}(\psi_1, \psi_2, \dots, \psi_n)$: This model is called *FactorAnalysis*, a classical statistical model. The matrix W is sometimes called the “factor loading matrix”.

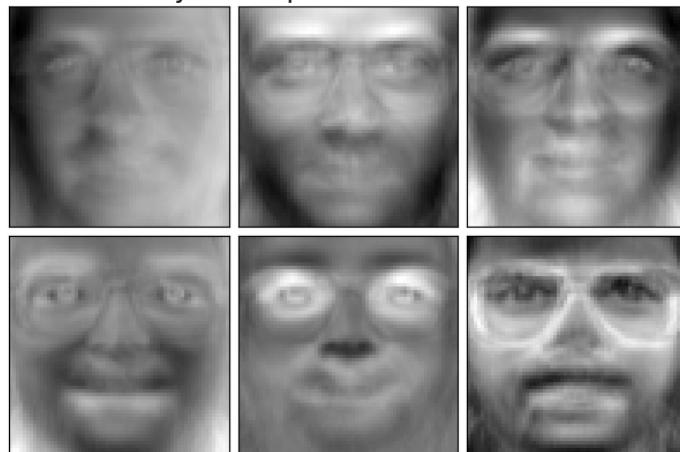
Both models essentially estimate a Gaussian with a low-rank covariance matrix. Because both models are probabilistic they can be integrated in more complex models, e.g. Mixture of Factor Analysers. One gets very different models (e.g. *FastICA*) if non-Gaussian priors on the latent variables are assumed.

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):

genfaces - PCA using randomized SVD - Train time 0.0



Factor Analysis components - FA - Train time 0.2s



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:

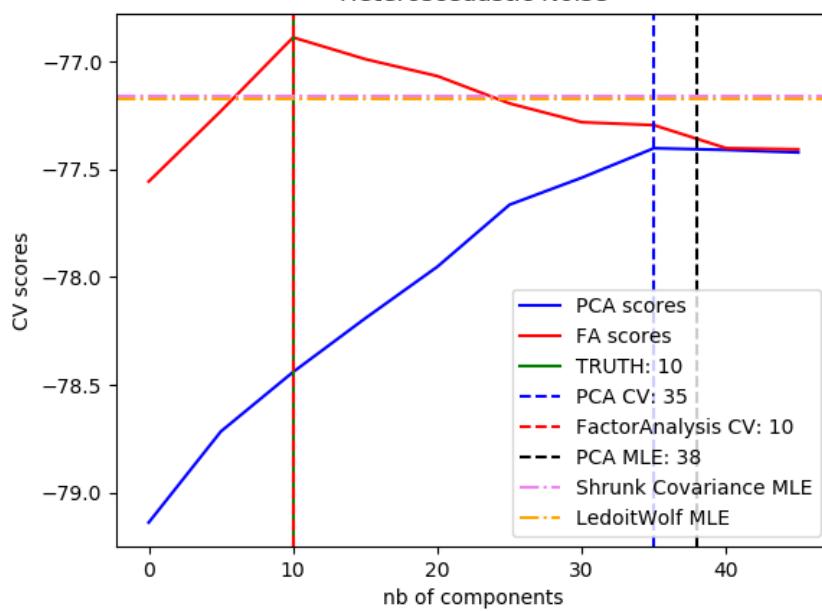
Examples:

- *Model selection with Probabilistic PCA and Factor Analysis (FA)*

Pixelwise variance



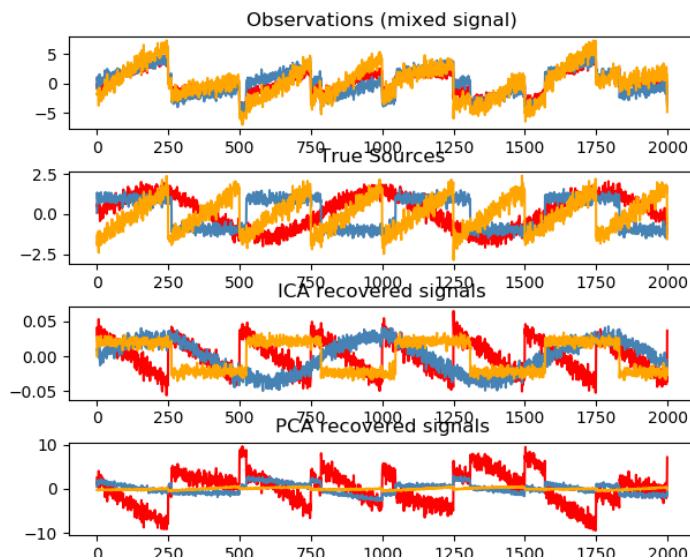
Heteroscedastic Noise



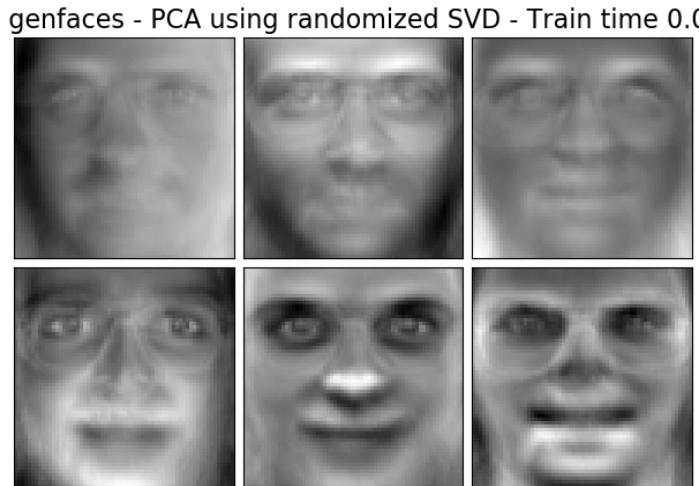
Independent component analysis (ICA)

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:



Independent components - FastICA - Train time 0.3s

**Examples:**

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

Non-negative matrix factorization (NMF or NNMF)**NMF with the Frobenius norm**

[NMF](#)¹ 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 W and H of non-negative elements, by optimizing the distance d between X and the matrix product WH . The most widely used distance function is the squared Frobenius norm, which is an obvious extension of the Euclidean norm to matrices:

$$d_{\text{Fro}}(X, Y) = \frac{1}{2} \|X - Y\|_{\text{Fro}}^2 = \frac{1}{2} \sum_{i,j} (X_{ij} - Y_{ij})^2$$

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, 2004]² 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.

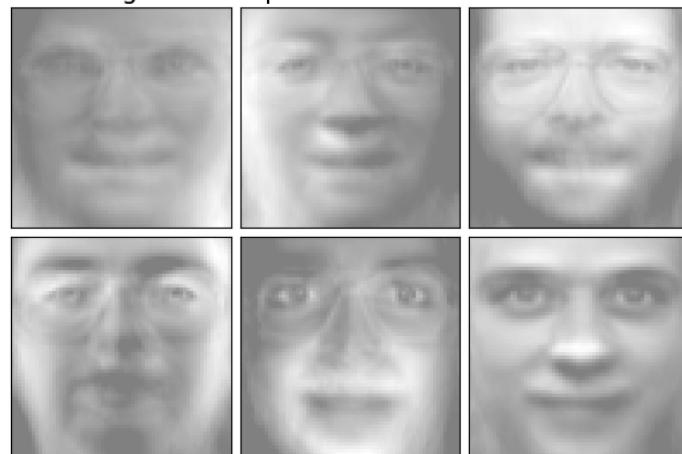
¹ “Learning the parts of objects by non-negative matrix factorization” D. Lee, S. Seung, 1999

² “Non-negative Matrix Factorization with Sparseness Constraints” P. Hoyer, 2004

genfaces - PCA using randomized SVD - Train time 0.0



Non-negative components - NMF - Train time 0.1s



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. NNDSVD⁴ 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.

Note that the Multiplicative Update ('mu') solver cannot update zeros present in the initialization, so it leads to poorer results when used jointly with the basic NNDSVD algorithm which introduces a lot of zeros; in this case, NNDSVDA or NNDSVDAR should be preferred.

[NMF](#) can also be initialized with correctly scaled random non-negative matrices by setting `init="random"`. An integer seed or a `RandomState` can also be passed to `random_state` to control reproducibility.

In [NMF](#), L1 and L2 priors can be added to the loss function in order to regularize the model. The L2 prior uses the Frobenius norm, while the L1 prior uses an elementwise L1 norm. As in `ElasticNet`, we control the combination of L1 and L2 with the `l1_ratio` (ρ) parameter, and the intensity of the regularization with the `alpha` (α) parameter. Then the priors terms are:

$$\alpha\rho\|W\|_1 + \alpha\rho\|H\|_1 + \frac{\alpha(1-\rho)}{2}\|W\|_{\text{Fro}}^2 + \frac{\alpha(1-\rho)}{2}\|H\|_{\text{Fro}}^2$$

⁴ "SVD based initialization: A head start for nonnegative matrix factorization" C. Boutsidis, E. Gallopoulos, 2008

and the regularized objective function is:

$$d_{\text{Fro}}(X, WH) + \alpha\rho||W||_1 + \alpha\rho||H||_1 + \frac{\alpha(1-\rho)}{2}||W||_{\text{Fro}}^2 + \frac{\alpha(1-\rho)}{2}||H||_{\text{Fro}}^2$$

`NMF` regularizes both `W` and `H`. The public function `non_negative_factorization` allows a finer control through the `regularization` attribute, and may regularize only `W`, only `H`, or both.

NMF with a beta-divergence

As described previously, the most widely used distance function is the squared Frobenius norm, which is an obvious extension of the Euclidean norm to matrices:

$$d_{\text{Fro}}(X, Y) = \frac{1}{2}||X - Y||_{\text{Fro}}^2 = \frac{1}{2} \sum_{i,j} (X_{ij} - Y_{ij})^2$$

Other distance functions can be used in NMF as, for example, the (generalized) Kullback-Leibler (KL) divergence, also referred as I-divergence:

$$d_{KL}(X, Y) = \sum_{i,j} (X_{ij} \log(\frac{X_{ij}}{Y_{ij}}) - X_{ij} + Y_{ij})$$

Or, the Itakura-Saito (IS) divergence:

$$d_{IS}(X, Y) = \sum_{i,j} (\frac{X_{ij}}{Y_{ij}} - \log(\frac{X_{ij}}{Y_{ij}}) - 1)$$

These three distances are special cases of the beta-divergence family, with $\beta = 2, 1, 0$ respectively⁶. The beta-divergence are defined by :

$$d_\beta(X, Y) = \sum_{i,j} \frac{1}{\beta(\beta-1)} (X_{ij}^\beta + (\beta-1)Y_{ij}^\beta - \beta X_{ij}Y_{ij}^{\beta-1})$$

Note that this definition is not valid if $\beta \in (0; 1)$, yet it can be continuously extended to the definitions of d_{KL} and d_{IS} respectively.

`NMF` implements two solvers, using Coordinate Descent ('cd')⁵, and Multiplicative Update ('mu')⁶. The 'mu' solver can optimize every beta-divergence, including of course the Frobenius norm ($\beta = 2$), the (generalized) Kullback-Leibler divergence ($\beta = 1$) and the Itakura-Saito divergence ($\beta = 0$). Note that for $\beta \in (1; 2)$, the 'mu' solver is significantly faster than for other values of β . Note also that with a negative (or 0, i.e. 'itakura-saito') β , the input matrix cannot contain zero values.

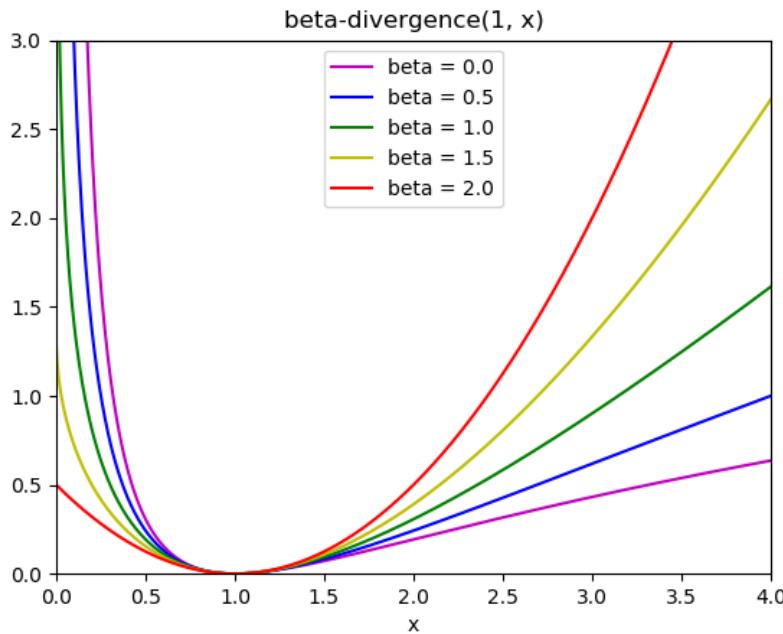
The 'cd' solver can only optimize the Frobenius norm. Due to the underlying non-convexity of NMF, the different solvers may converge to different minima, even when optimizing the same distance function.

NMF is best used with the `fit_transform` method, which returns the matrix `W`. The matrix `H` is stored into the fitted model in the `components_` attribute; the method `transform` will decompose a new matrix `X_new` based on these stored components:

```
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [3, 1.2], [4, 1], [5, 0.8], [6, 1]])
>>> from sklearn.decomposition import NMF
>>> model = NMF(n_components=2, init='random', random_state=0)
```

⁶ "Algorithms for nonnegative matrix factorization with the beta-divergence" C. Févotte, J. Idier, 2011

⁵ "Fast local algorithms for large scale nonnegative matrix and tensor factorizations." A. Cichocki, A. Phan, 2009



```
>>> W = model.fit_transform(X)
>>> H = model.components_
>>> X_new = np.array([[1, 0], [1, 6.1], [1, 0], [1, 4], [3.2, 1], [0, 4]])
>>> W_new = model.transform(X_new)
```

Examples:

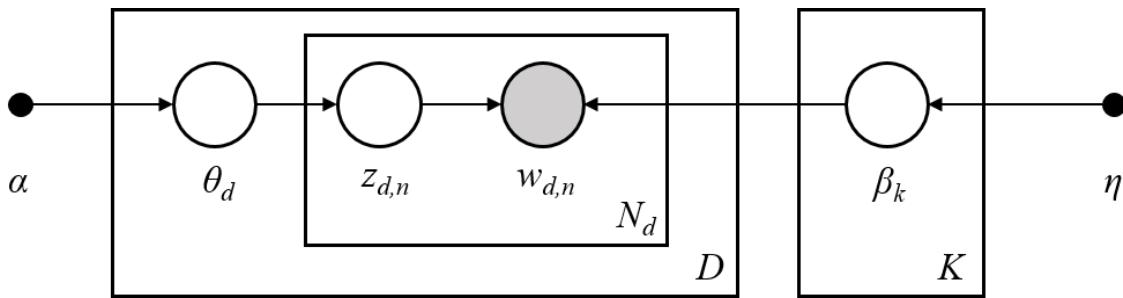
- *Faces dataset decompositions*
- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*
- *Beta-divergence loss functions*

References:

Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation is a generative probabilistic model for collections of discrete dataset such as text corpora. It is also a topic model that is used for discovering abstract topics from a collection of documents.

The graphical model of LDA is a three-level generative model:



Note on notations presented in the graphical model above, which can be found in Hoffman et al. (2013):

- The corpus is a collection of D documents.
- A document is a sequence of N words.
- There are K topics in the corpus.
- The boxes represent repeated sampling.

In the graphical model, each node is a random variable and has a role in the generative process. A shaded node indicates an observed variable and an unshaded node indicates a hidden (latent) variable. In this case, words in the corpus are the only data that we observe. The latent variables determine the random mixture of topics in the corpus and the distribution of words in the documents. The goal of LDA is to use the observed words to infer the hidden topic structure.

When modeling text corpora, the model assumes the following generative process for a corpus with D documents and K topics, with K corresponding to `n_components` in the API:

1. For each topic $k \in K$, draw $\beta_k \sim \text{Dirichlet}(\eta)$. This provides a distribution over the words, i.e. the probability of a word appearing in topic k . η corresponds to `topic_word_prior`.
2. For each document $d \in D$, draw the topic proportions $\theta_d \sim \text{Dirichlet}(\alpha)$. α corresponds to `doc_topic_prior`.
3. For each word i in document d :
 1. Draw the topic assignment $z_{di} \sim \text{Multinomial}(\theta_d)$
 2. Draw the observed word $w_{ij} \sim \text{Multinomial}(\beta_{z_{di}})$

For parameter estimation, the posterior distribution is:

$$p(z, \theta, \beta | w, \alpha, \eta) = \frac{p(z, \theta, \beta | \alpha, \eta)}{p(w | \alpha, \eta)}$$

Since the posterior is intractable, variational Bayesian method uses a simpler distribution $q(z, \theta, \beta | \lambda, \phi, \gamma)$ to approximate it, and those variational parameters λ, ϕ, γ are optimized to maximize the Evidence Lower Bound (ELBO):

$$\log P(w | \alpha, \eta) \geq L(w, \phi, \gamma, \lambda) \stackrel{\Delta}{=} E_q[\log p(w, z, \theta, \beta | \alpha, \eta)] - E_q[\log q(z, \theta, \beta)]$$

Maximizing ELBO is equivalent to minimizing the Kullback-Leibler(KL) divergence between $q(z, \theta, \beta)$ and the true posterior $p(z, \theta, \beta | w, \alpha, \eta)$.

`LatentDirichletAllocation` implements the online variational Bayes algorithm and supports both online and batch update methods. While the batch method updates variational variables after each full pass through the data, the online method updates variational variables from mini-batch data points.

Note: Although the online method is guaranteed to converge to a local optimum point, the quality of the optimum point and the speed of convergence may depend on mini-batch size and attributes related to learning rate setting.

When `LatentDirichletAllocation` is applied on a “document-term” matrix, the matrix will be decomposed into a “topic-term” matrix and a “document-topic” matrix. While “topic-term” matrix is stored as `components_` in the model, “document-topic” matrix can be calculated from `transform` method.

`LatentDirichletAllocation` also implements `partial_fit` method. This is used when data can be fetched sequentially.

Examples:

- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*

References:

- “Latent Dirichlet Allocation” D. Blei, A. Ng, M. Jordan, 2003
- “Online Learning for Latent Dirichlet Allocation” M. Hoffman, D. Blei, F. Bach, 2010
- “Stochastic Variational Inference” M. Hoffman, D. Blei, C. Wang, J. Paisley, 2013

See also `Dimensionality reduction` for dimensionality reduction with Neighborhood Components Analysis.

3.2.6 Covariance estimation

Many statistical problems require 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) have a large influence on the estimation’s quality. The `sklearn.covariance` package provides tools for accurately estimating a population’s covariance matrix under various settings.

We assume that the observations are independent and identically distributed (i.i.d.).

Empirical covariance

The covariance matrix of a data set is known to be well approximated by 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’s 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 results depend on whether the data are centered, so one may want to use the `assume_centered` parameter accurately. More precisely, if `assume_centered=False`, then the test set is supposed to have the same mean vector as the training set. If not, 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.

Shrunk Covariance

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 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, results depend on whether the data are centered, 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 eigenvalues 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.

Ledoit-Wolf shrinkage

In their 2004 paper¹, O. Ledoit and M. Wolf propose a formula 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.

Note: Case when population covariance matrix is isotropic

It is important to note that when the number of samples is much larger than the number of features, one would expect that no shrinkage would be necessary. The intuition behind this is that if the population covariance is full rank, when the number of sample grows, the sample covariance will also become positive definite. As a result, no shrinkage would necessary and the method should automatically do this.

This, however, is not the case in the Ledoit-Wolf procedure when the population covariance happens to be a multiple of the identity matrix. In this case, the Ledoit-Wolf shrinkage estimate approaches 1 as the number of samples increases. This indicates that the optimal estimate of the covariance matrix in the Ledoit-Wolf sense is multiple of the identity. Since the population covariance is already a multiple of the identity matrix, the Ledoit-Wolf solution is indeed a reasonable estimate.

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.

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.

References:**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.

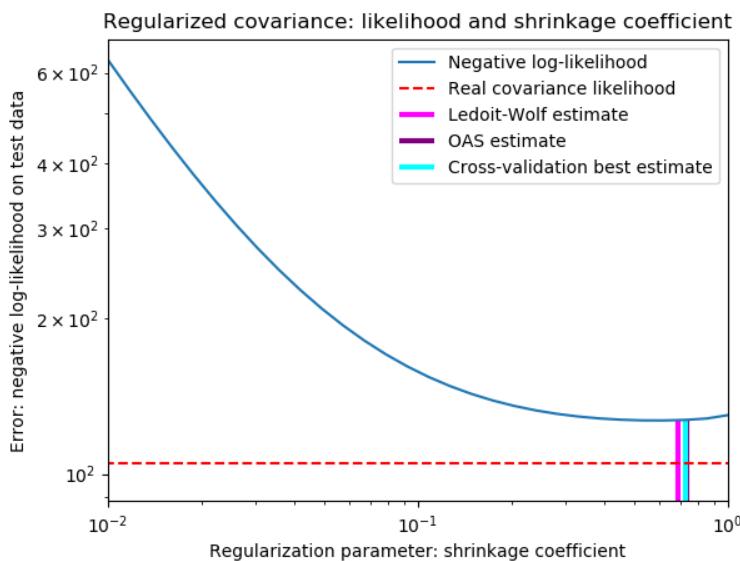


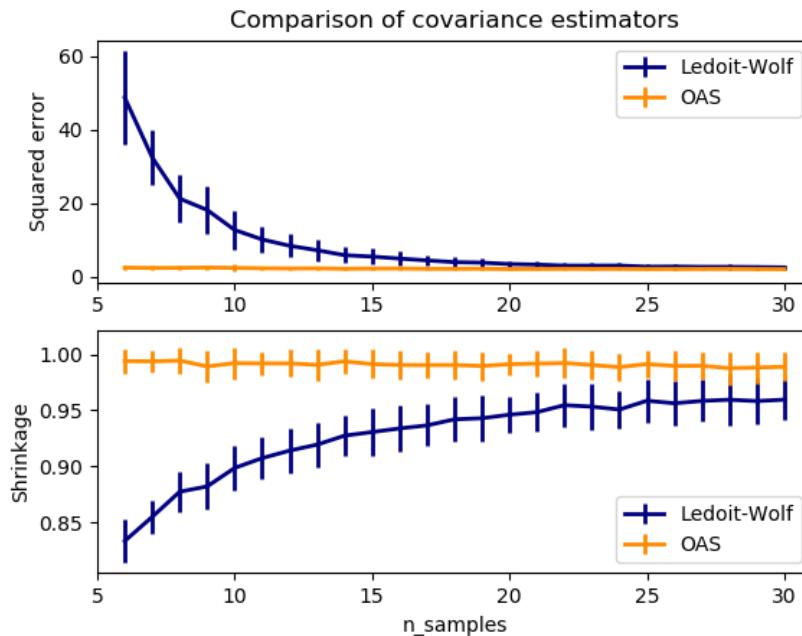
Fig. 3.7: Bias-variance trade-off when setting the shrinkage: comparing the choices of Ledoit-Wolf and OAS estimators

References:**Examples:**

- See [Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood](#) for an example on how to fit an `OAS` object to data.

² Chen et al., "Shrinkage Algorithms for MMSE Covariance Estimation", IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

- See [Ledoit-Wolf vs OAS estimation](#) to visualize the Mean Squared Error difference between a `LedoitWolf` and an `OAS` estimator of the covariance.



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: the estimation of the covariance matrix is better conditioned by learning independence relations from the data. 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 `GraphicalLasso` estimator uses an ℓ_1 penalty to enforce sparsity on the precision matrix: the higher its `alpha` parameter, the more sparse the precision matrix. The corresponding `GraphicalLassoCV` object uses cross-validation to automatically set the `alpha` parameter.

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 `GraphicalLasso`
- If the underlying graph has nodes with much more connections than the average node, the algorithm will miss some of these connections.

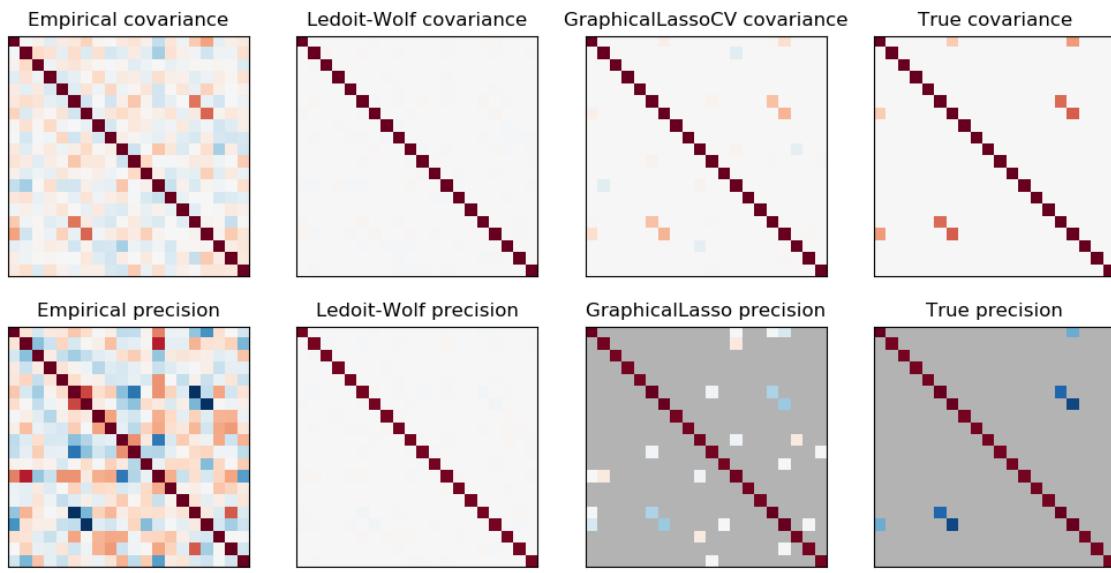


Fig. 3.8: A comparison of maximum likelihood, shrinkage and sparse estimates of the covariance and precision matrix in the very small samples settings.

- 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 `GraphicalLassoCV` 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:

$$\hat{K} = \operatorname{argmin}_K (\operatorname{tr} SK - \log \det K + \alpha \|K\|_1)$$

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

Robust Covariance Estimation

Real data sets are often subject to measurement or recording errors. Regular but uncommon observations may also appear for a variety of reasons. Observations which are very uncommon are called outliers. The empirical covariance estimator and the shrunk covariance estimators presented above are very sensitive to the presence of outliers 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 Determinant³.

Minimum Covariance Determinant

The Minimum Covariance Determinant estimator is a robust estimator of a data set's covariance introduced by P.J. Rousseeuw in³. 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 to a reweighted estimate of the covariance matrix of the data set (“reweighting step”).

Rousseeuw and Van Driessen⁴ 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.

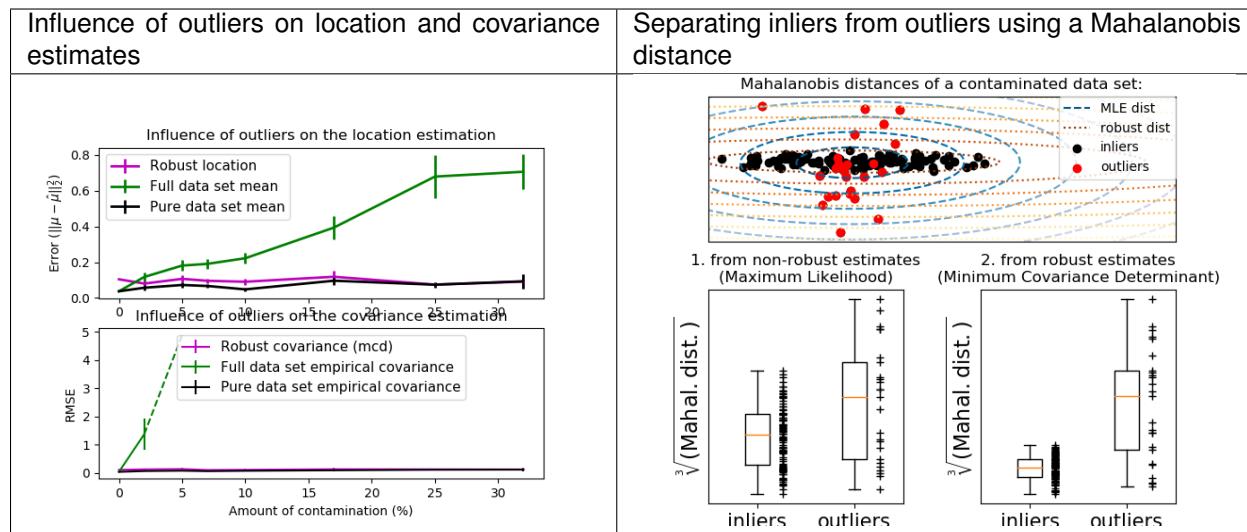
References:

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).

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.



3.2.7 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 distinctions must be made:

outlier detection The training data contains outliers which are defined as observations that are far from the others. Outlier detection estimators thus try to fit the regions where the training data is the most concentrated, ignoring the deviant observations.

novelty detection The training data is not polluted by outliers and we are interested in detecting whether a **new** observation is an outlier. In this context an outlier is also called a novelty.

Outlier detection and novelty detection are both used for anomaly detection, where one is interested in detecting abnormal or unusual observations. Outlier detection is then also known as unsupervised anomaly detection and novelty detection as semi-supervised anomaly detection. In the context of outlier detection, the outliers/anomalies cannot form a dense cluster as available estimators assume that the outliers/anomalies are located in low density regions. On the contrary, in the context of novelty detection, novelties/anomalies can form a dense cluster as long as they are in a low density region of the training data, considered as normal in this context.

The scikit-learn project provides a set of machine learning tools that can be used both for novelty or outlier 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. The `predict` method makes use of a threshold on the raw scoring function computed by the estimator. This scoring function is accessible through the `score_samples` method, while the threshold can be controlled by the `contamination` parameter.

The `decision_function` method is also defined from the scoring function, in such a way that negative values are outliers and non-negative ones are inliers:

```
estimator.decision_function(X_test)
```

Note that `neighbors.LocalOutlierFactor` does not support `predict`, `decision_function` and `score_samples` methods by default but only a `fit_predict` method, as this estimator was originally meant to be applied for outlier detection. The scores of abnormality of the training samples are accessible through the `negative_outlier_factor_` attribute.

If you really want to use `neighbors.LocalOutlierFactor` for novelty detection, i.e. predict labels or compute the score of abnormality of new unseen data, you can instantiate the estimator with the `novelty` parameter set to `True` before fitting the estimator. In this case, `fit_predict` is not available.

Warning: Novelty detection with Local Outlier Factor

When `novelty` is set to `True` be aware that you must only use `predict`, `decision_function` and `score_samples` on new unseen data and not on the training samples as this would lead to wrong results. The scores of abnormality of the training samples are always accessible through the `negative_outlier_factor_` attribute.

The behavior of `neighbors.LocalOutlierFactor` is summarized in the following table.

Method	Outlier detection	Novelty detection
<code>fit_predict</code>	OK	Not available
<code>predict</code>	Not available	Use only on new data
<code>decision_function</code>	Not available	Use only on new data
<code>score_samples</code>	Use <code>negative_outlier_factor_</code>	Use only on new data

Overview of outlier detection methods

A comparison of the outlier detection algorithms in scikit-learn. Local Outlier Factor (LOF) does not show a decision boundary in black as it has no `predict` method to be applied on new data when it is used for outlier detection.

`ensemble.IsolationForest` and `neighbors.LocalOutlierFactor` perform reasonably well on the data sets considered here. The `svm.OneClassSVM` is known to be sensitive to outliers and thus does not perform very well for outlier detection. Finally, `covariance.EllipticEnvelope` assumes the data is Gaussian and learns an ellipse. For more details on the different estimators refer to the example [Comparing anomaly detection algorithms for outlier detection on toy datasets](#) and the sections hereunder.

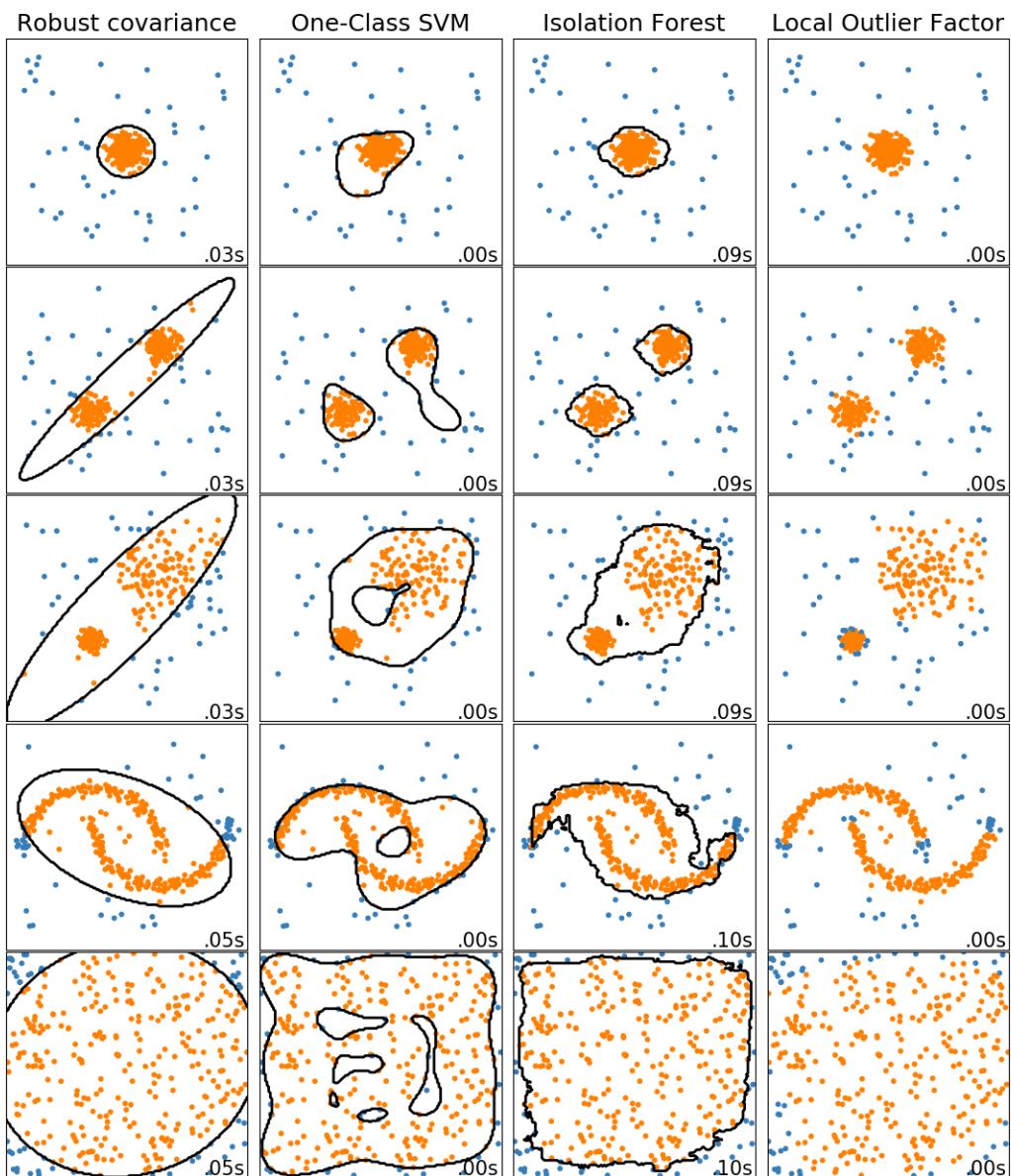
Examples:

- See [Comparing anomaly detection algorithms for outlier detection on toy datasets](#) for a comparison of the `svm.OneClassSVM`, the `ensemble.IsolationForest`, the `neighbors.LocalOutlierFactor` and `covariance.EllipticEnvelope`.

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.

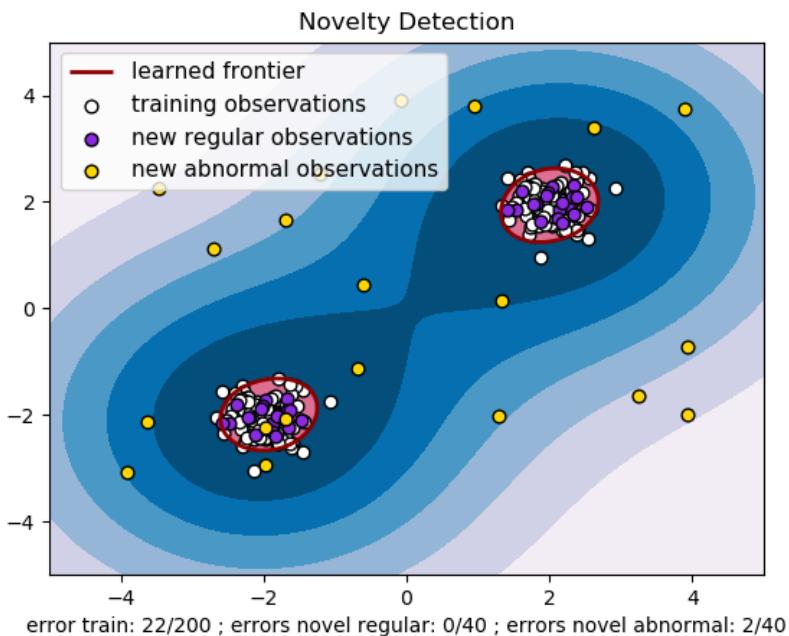
The One-Class SVM has been introduced by Schölkopf et al. for that purpose and implemented in the [Support Vector Machines](#) module in the `svm.OneClassSVM` object. It requires the choice of a kernel and a scalar parameter to define a frontier. The RBF kernel is usually chosen although there exists 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.
- [Species distribution modeling](#)



Outlier Detection

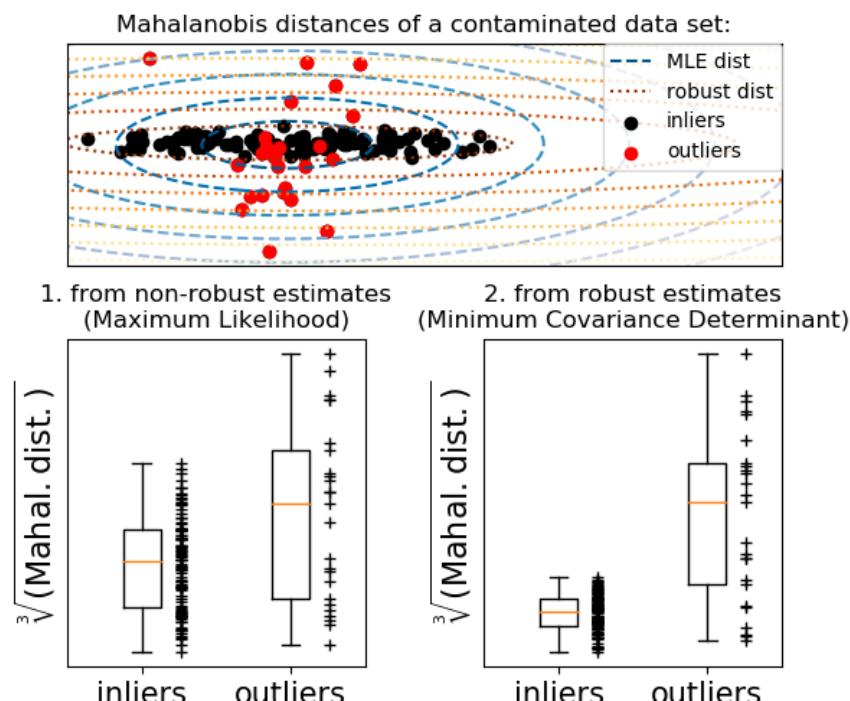
Outlier detection is similar to novelty detection in the sense that the goal is to separate a core of regular observations from some polluting ones, called *outliers*. Yet, in the case of outlier detection, we don't have a clean data set representing the population of regular observations that can be used to train any tool.

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. without 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:

- Rousseeuw, P.J., Van Driessen, K. “A fast algorithm for the minimum covariance determinant estimator” *Technometrics* 41(3), 212 (1999)

Isolation Forest

One efficient way of performing outlier detection in high-dimensional datasets is to use random forests. The `ensemble.IsolationForest` ‘isolates’ observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

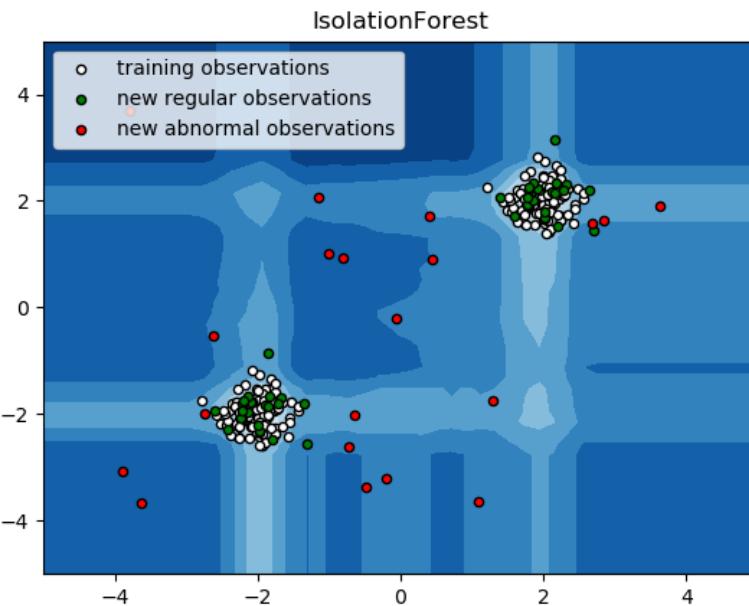
Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node.

This path length, averaged over a forest of such random trees, is a measure of normality and our decision function.

Random partitioning produces noticeably shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.

The implementation of `ensemble.IsolationForest` is based on an ensemble of `tree.ExtraTreeRegressor`. Following Isolation Forest original paper, the maximum depth of each tree is set to $\lceil \log_2(n) \rceil$ where n is the number of samples used to build the tree (see (Liu et al., 2008) for more details).

This algorithm is illustrated below. The `ensemble.IsolationForest` supports `warm_start=True` which



allows you to add more trees to an already fitted model:

```
>>> from sklearn.ensemble import IsolationForest
>>> import numpy as np
>>> X = np.array([-1, -1], [-2, -1], [-3, -2], [0, 0], [-20, 50], [3, 5])
>>> clf = IsolationForest(n_estimators=10, warm_start=True)
>>> clf.fit(X) # fit 10 trees
>>> clf.set_params(n_estimators=20) # add 10 more trees
>>> clf.fit(X) # fit the added trees
```

Examples:

- See [IsolationForest example](#) for an illustration of the use of IsolationForest.

- See [Comparing anomaly detection algorithms for outlier detection on toy datasets](#) for a comparison of `ensemble.IsolationForest` with `neighbors.LocalOutlierFactor`, `svm.OneClassSVM` (tuned to perform like an outlier detection method) and a covariance-based outlier detection with `covariance.EllipticEnvelope`.

References:

- Liu, Fei Tony, Ting, Kai Ming and Zhou, Zhi-Hua. “Isolation forest.” Data Mining, 2008. ICDM’08. Eighth IEEE International Conference on.

Local Outlier Factor

Another efficient way to perform outlier detection on moderately high dimensional datasets is to use the Local Outlier Factor (LOF) algorithm.

The `neighbors.LocalOutlierFactor` (LOF) algorithm computes a score (called local outlier factor) reflecting the degree of abnormality of the observations. It measures the local density deviation of a given data point with respect to its neighbors. The idea is to detect the samples that have a substantially lower density than their neighbors.

In practice the local density is obtained from the k-nearest neighbors. The LOF score of an observation is equal to the ratio of the average local density of his k-nearest neighbors, and its own local density: a normal instance is expected to have a local density similar to that of its neighbors, while abnormal data are expected to have much smaller local density.

The number k of neighbors considered, (alias parameter `n_neighbors`) is typically chosen 1) greater than the minimum number of objects a cluster has to contain, so that other objects can be local outliers relative to this cluster, and 2) smaller than the maximum number of close by objects that can potentially be local outliers. In practice, such informations are generally not available, and taking `n_neighbors=20` appears to work well in general. When the proportion of outliers is high (i.e. greater than 10 %, as in the example below), `n_neighbors` should be greater (`n_neighbors=35` in the example below).

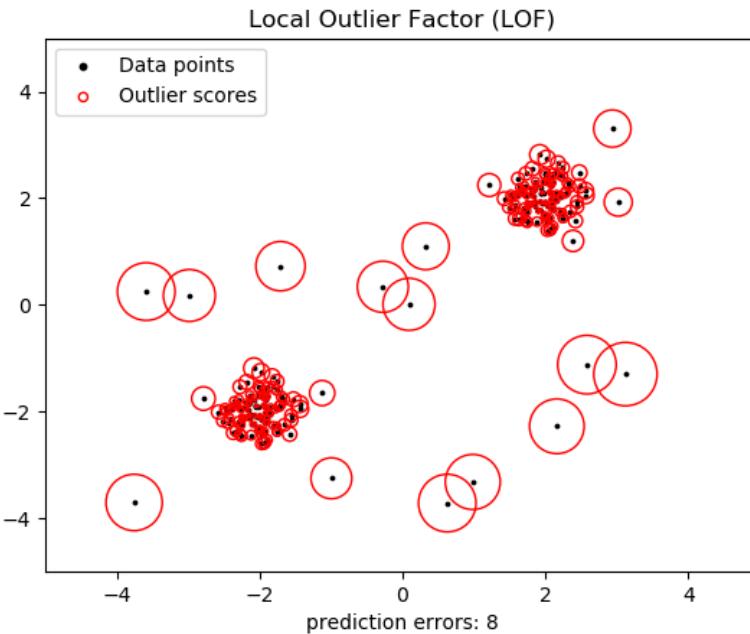
The strength of the LOF algorithm is that it takes both local and global properties of datasets into consideration: it can perform well even in datasets where abnormal samples have different underlying densities. The question is not, how isolated the sample is, but how isolated it is with respect to the surrounding neighborhood.

When applying LOF for outlier detection, there are no `predict`, `decision_function` and `score_samples` methods but only a `fit_predict` method. The scores of abnormality of the training samples are accessible through the `negative_outlier_factor_` attribute. Note that `predict`, `decision_function` and `score_samples` can be used on new unseen data when LOF is applied for novelty detection, i.e. when the `novelty` parameter is set to True. See [Novelty detection with Local Outlier Factor](#).

This strategy is illustrated below.

Examples:

- See [Outlier detection with Local Outlier Factor \(LOF\)](#) for an illustration of the use of `neighbors.LocalOutlierFactor`.
- See [Comparing anomaly detection algorithms for outlier detection on toy datasets](#) for a comparison with other anomaly detection methods.



References:

- Breunig, Kriegel, Ng, and Sander (2000) LOF: identifying density-based local outliers. Proc. ACM SIGMOD

Novelty detection with Local Outlier Factor

To use `neighbors.LocalOutlierFactor` for novelty detection, i.e. predict labels or compute the score of abnormality of new unseen data, you need to instantiate the estimator with the `novelty` parameter set to `True` before fitting the estimator:

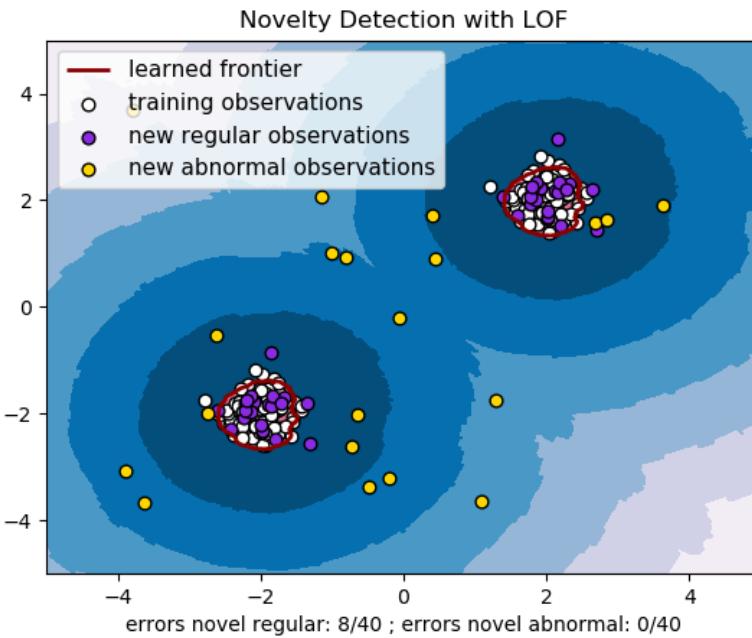
```
lof = LocalOutlierFactor(novelty=True)
lof.fit(X_train)
```

Note that `fit_predict` is not available in this case.

Warning: Novelty detection with Local Outlier Factor¹

When `novelty` is set to `True` be aware that you must only use `predict`, `decision_function` and `score_samples` on new unseen data and not on the training samples as this would lead to wrong results. The scores of abnormality of the training samples are always accessible through the `negative_outlier_factor_` attribute.

Novelty detection with Local Outlier Factor is illustrated below.



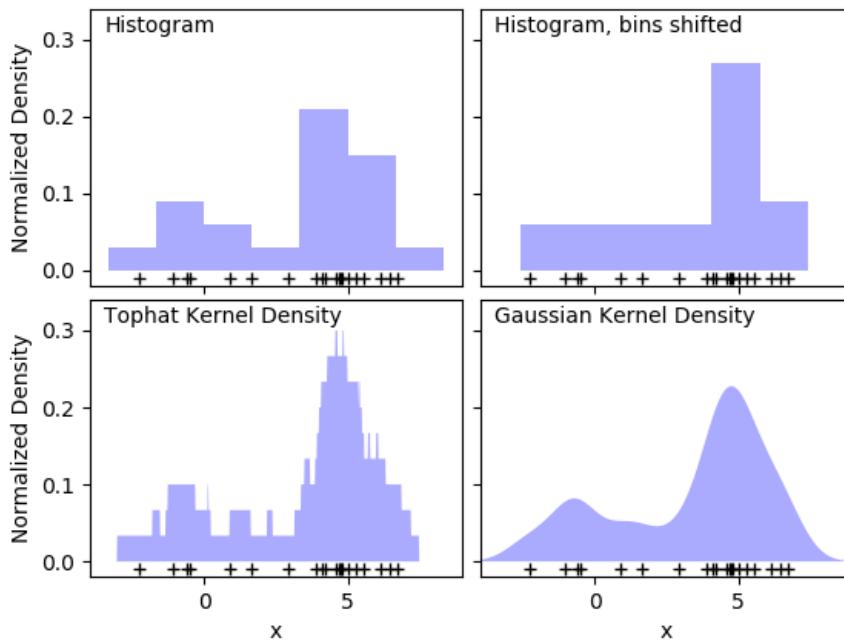
3.2.8 Density Estimation

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.GaussianMixture](#)), 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.

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.

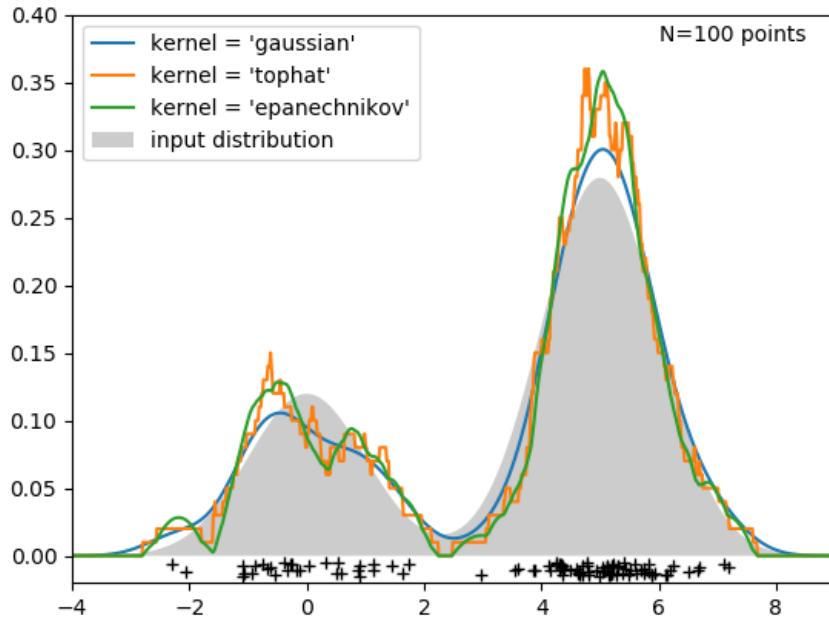
Intuitively, one can also think of a histogram as a stack of blocks, one block per point. By stacking the blocks in the appropriate grid space, we recover the histogram. But what if, instead of stacking the blocks on a regular grid, we center each block on the point it represents, and sum the total height 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.

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:

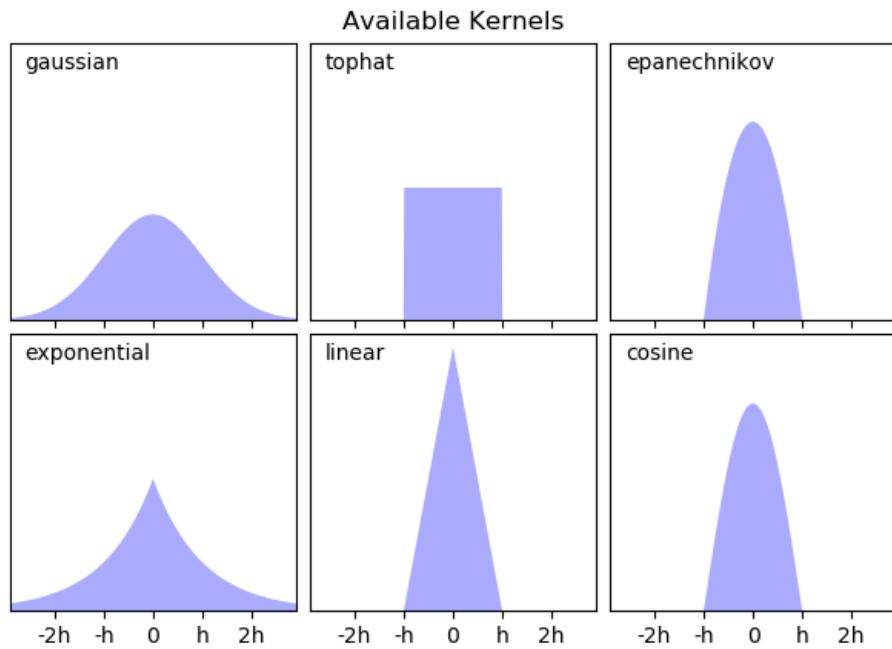
```
>>> from sklearn.neighbors.kde import KernelDensity
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> kde = KernelDensity(kernel='gaussian', bandwidth=0.2).fit(X)
>>> kde.score_samples(X)
array([-0.41075698, -0.41075698, -0.41076071, -0.41075698, -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'`)

$$K(x; h) \propto \exp\left(-\frac{x^2}{2h^2}\right)$$
- Tophat kernel (`kernel = 'tophat'`)

$$K(x; h) \propto 1 \text{ if } x < h$$
- Epanechnikov kernel (`kernel = 'epanechnikov'`)

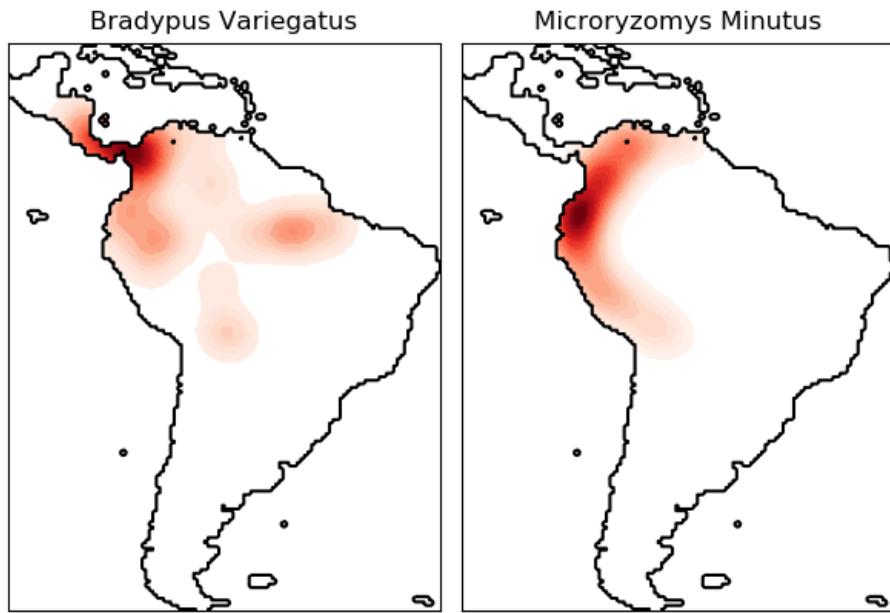
$$K(x; h) \propto 1 - \frac{x^2}{h^2}$$
- Exponential kernel (`kernel = 'exponential'`)

$$K(x; h) \propto \exp(-x/h)$$
- Linear kernel (`kernel = 'linear'`)

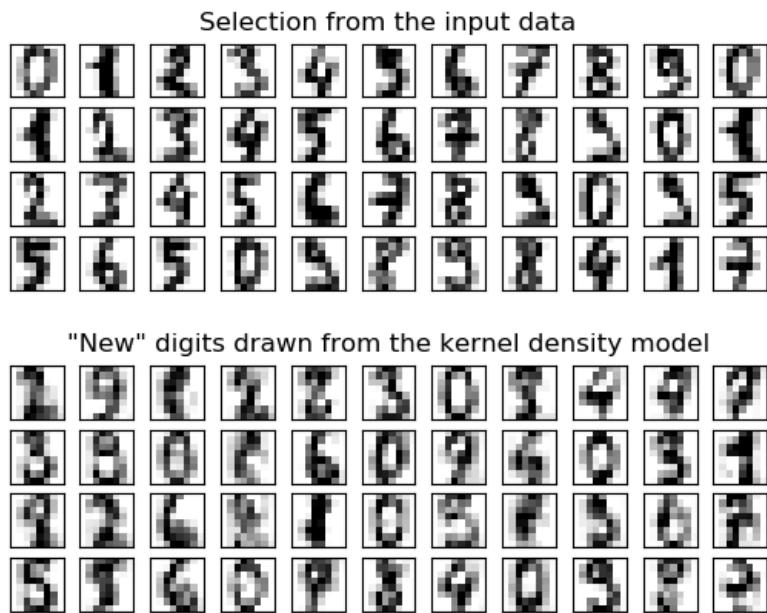
$$K(x; h) \propto 1 - x/h \text{ if } x < h$$
- Cosine kernel (`kernel = 'cosine'`)

$$K(x; h) \propto \cos\left(\frac{\pi x}{2h}\right) \text{ if } x < h$$

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

3.2.9 Neural network models (unsupervised)

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.

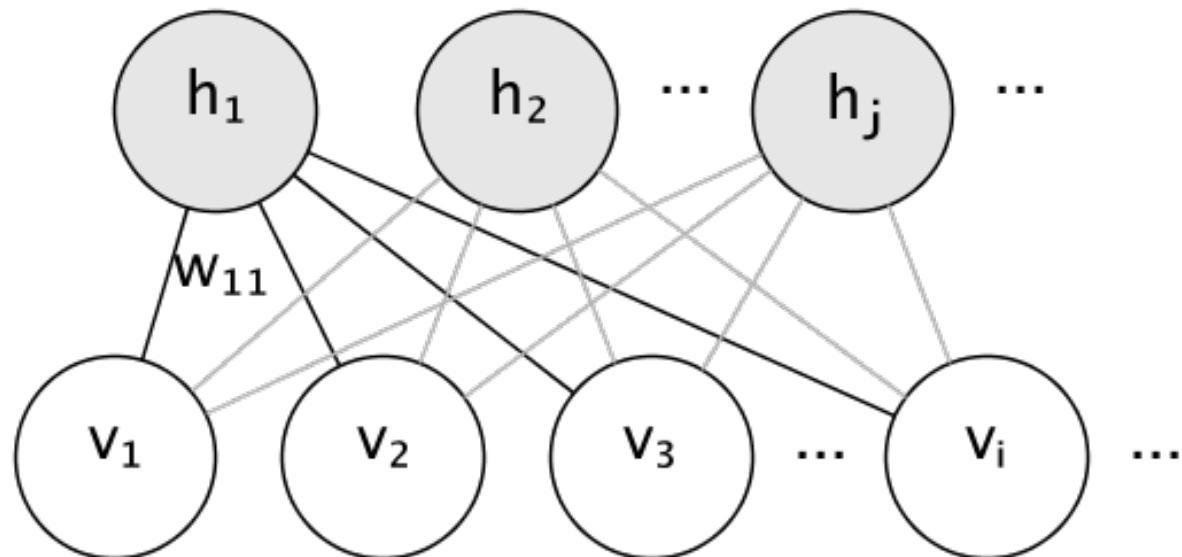
Examples:

- *Restricted Boltzmann Machine features for digit classification*

Graphical model and parametrization

The graphical model of an RBM is a fully-connected bipartite graph.

100 components extracted by RBM



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, omitted 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:

$$\begin{aligned} h_i &\perp h_j \mid \mathbf{v} \\ v_i &\perp v_j \mid \mathbf{h} \end{aligned}$$

The bipartite structure allows for the use of efficient block Gibbs sampling for inference.

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:

$$\begin{aligned} P(v_i = 1 \mid \mathbf{h}) &= \sigma\left(\sum_j w_{ij} h_j + b_i\right) \\ P(h_i = 1 \mid \mathbf{v}) &= \sigma\left(\sum_i w_{ij} v_i + c_j\right) \end{aligned}$$

where σ is the logistic sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

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 referred 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.

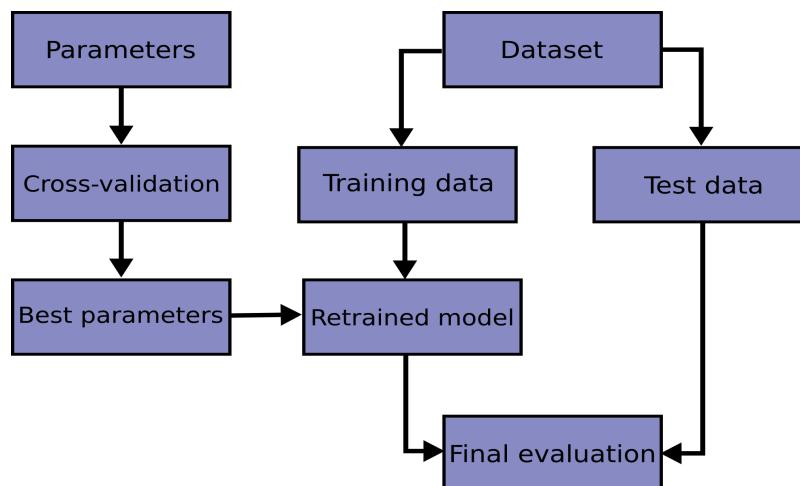
References:

- “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

3.3 Model selection and evaluation

3.3.1 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. Here is a flowchart of typical cross validation workflow in model training. The best parameters can be determined by *grid search* techniques.



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.model_selection import train_test_split
>>> 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 = 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...
```

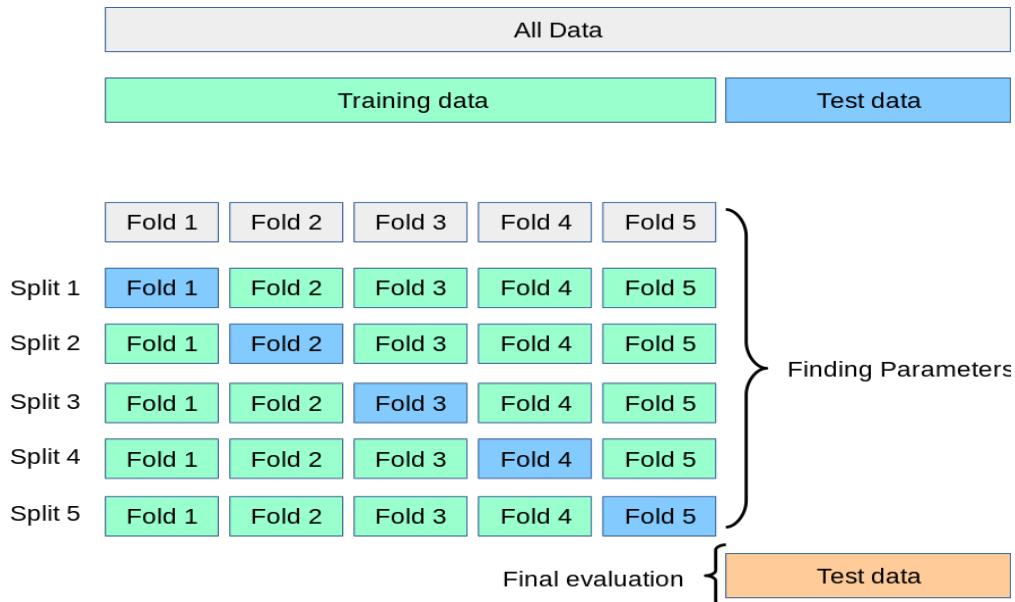
When evaluating different settings (“hyperparameters”) for estimators, such as the C setting that must be manually set for an SVM, there is still a risk of overfitting *on the test set* because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set 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 is the case when fixing an arbitrary validation set), which is a major advantage in problems such as inverse inference where the number of samples is very small.



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):

```
>>> from sklearn.model_selection import cross_val_score
>>> clf = svm.SVC(kernel='linear', C=1)
>>> scores = 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 95% confidence interval 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_val_score(
...     clf, iris.data, iris.target, cv=5, scoring='f1_macro')
>>> 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, the latter being used if the estimator derives from `ClassifierMixin`.

It is also possible to use other cross validation strategies by passing a cross validation iterator instead, for instance:

```
>>> from sklearn.model_selection import ShuffleSplit
>>> n_samples = iris.data.shape[0]
>>> cv = ShuffleSplit(n_splits=5, test_size=0.3, random_state=0)
>>> cross_val_score(clf, iris.data, iris.target, cv=cv)
array([0.977..., 0.977..., 1.  ..., 0.955..., 1.        ])
```

Another option is to use an iterable yielding (train, test) splits as arrays of indices, for example:

```
>>> def custom_cv_2folds(X):
...     n = X.shape[0]
...     i = 1
...     while i <= 2:
...         idx = np.arange(n * (i - 1) / 2, n * i / 2, dtype=int)
...         yield idx, idx
...         i += 1
...
>>> custom_cv = custom_cv_2folds(iris.data)
>>> cross_val_score(clf, iris.data, iris.target, cv=custom_cv)
array([1.        , 0.973...])
```

Data transformation with held out data

Just as it is important to test a predictor on data held-out from training, preprocessing (such as standardization, feature selection, etc.) and similar *data transformations* similarly should be learnt from a training set and applied to held-out data for prediction:

```
>>> from sklearn import preprocessing
>>> X_train, X_test, y_train, y_test = train_test_split(
...     iris.data, iris.target, test_size=0.4, random_state=0)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train_transformed = scaler.transform(X_train)
>>> clf = svm.SVC(C=1).fit(X_train_transformed, y_train)
>>> X_test_transformed = scaler.transform(X_test)
>>> clf.score(X_test_transformed, y_test)
0.9333...
```

A *Pipeline* makes it easier to compose estimators, providing this behavior under cross-validation:

```
>>> from sklearn.pipeline import make_pipeline
>>> clf = make_pipeline(preprocessing.StandardScaler(), svm.SVC(C=1))
>>> cross_val_score(clf, iris.data, iris.target, cv=cv)
...
array([0.977..., 0.933..., 0.955..., 0.933..., 0.977...])
```

See [Pipelines and composite estimators](#).

The `cross_validate` function and multiple metric evaluation

The `cross_validate` function differs from `cross_val_score` in two ways:

- It allows specifying multiple metrics for evaluation.
- It returns a dict containing fit-times, score-times (and optionally training scores as well as fitted estimators) in addition to the test score.

For single metric evaluation, where the scoring parameter is a string, callable or None, the keys will be -
['test_score', 'fit_time', 'score_time']

And for multiple metric evaluation, the return value is a dict with the following keys -
['test_<scorer1_name>', 'test_<scorer2_name>', 'test_<scorer...>', 'fit_time', 'score_time']

`return_train_score` is set to `False` by default to save computation time. To evaluate the scores on the training set as well you need to be set to `True`.

You may also retain the estimator fitted on each training set by setting `return_estimator=True`.

The multiple metrics can be specified either as a list, tuple or set of predefined scorer names:

```
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics import recall_score
>>> scoring = ['precision_macro', 'recall_macro']
>>> clf = svm.SVC(kernel='linear', C=1, random_state=0)
>>> scores = cross_validate(clf, iris.data, iris.target, scoring=scoring,
...                         cv=5)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_precision_macro', 'test_recall_macro']
>>> scores['test_recall_macro']
array([0.96..., 1. ..., 0.96..., 0.96..., 1. ...])
```

Or as a dict mapping scorer name to a predefined or custom scoring function:

```
>>> from sklearn.metrics.scorer import make_scorer
>>> scoring = {'prec_macro': 'precision_macro',
...             'rec_macro': make_scorer(recall_score, average='macro')}
>>> scores = cross_validate(clf, iris.data, iris.target, scoring=scoring,
...                         cv=5, return_train_score=True)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_prec_macro', 'test_rec_macro',
 'train_prec_macro', 'train_rec_macro']
>>> scores['train_rec_macro']
array([0.97..., 0.97..., 0.99..., 0.98..., 0.98...])
```

Here is an example of `cross_validate` using a single metric:

```
>>> scores = cross_validate(clf, iris.data, iris.target,
...                         scoring='precision_macro', cv=5,
...                         return_estimator=True)
>>> sorted(scores.keys())
['estimator', 'fit_time', 'score_time', 'test_score']
```

Obtaining predictions by cross-validation

The function `cross_val_predict` has a similar interface to `cross_val_score`, but returns, for each element in the input, the prediction that was obtained for that element when it was in the test set. Only cross-validation strategies that assign all elements to a test set exactly once can be used (otherwise, an exception is raised).

Warning: Note on inappropriate usage of `cross_val_predict`

The result of `cross_val_predict` may be different from those obtained using `cross_val_score` as the elements are grouped in different ways. The function `cross_val_score` takes an average over cross-validation folds, whereas `cross_val_predict` simply returns the labels (or probabilities) from several distinct models undistinguished. Thus, `cross_val_predict` is not an appropriate measure of generalisation error.

The function `cross_val_predict` is appropriate for:

- Visualization of predictions obtained from different models.
- Model blending: When predictions of one supervised estimator are used to train another estimator in ensemble methods.

The available cross validation iterators are introduced in the following section.

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,*
- *Plotting Cross-Validated Predictions,*
- *Nested versus non-nested cross-validation.*

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.

Cross-validation iterators for i.i.d. data

Assuming that some data is Independent and Identically Distributed (i.i.d.) is making the assumption that all samples stem from the same generative process and that the generative process is assumed to have no memory of past generated samples.

The following cross-validators can be used in such cases.

NOTE

While i.i.d. data is a common assumption in machine learning theory, it rarely holds in practice. If one knows that the samples have been generated using a time-dependent process, it's safer to use a [time-series aware cross-validation scheme](#). Similarly if we know that the generative process has a group structure (samples from collected from different subjects, experiments, measurement devices) it's safer to use [group-wise cross-validation](#).

K-fold

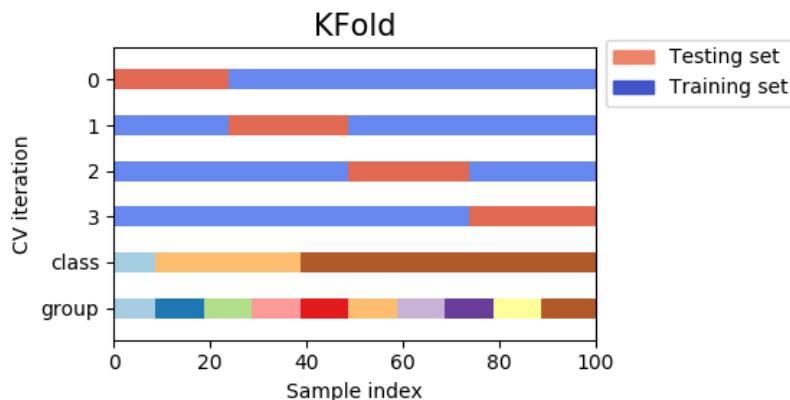
`KFold` divides all the samples in k groups of samples, called folds (if $k = n$, this is equivalent to the *Leave One Out* strategy), of equal sizes (if possible). The prediction function is learned using $k - 1$ folds, and the fold left out is used for test.

Example of 2-fold cross-validation on a dataset with 4 samples:

```
>>> import numpy as np
>>> from sklearn.model_selection import KFold

>>> X = ["a", "b", "c", "d"]
>>> kf = KFold(n_splits=2)
>>> for train, test in kf.split(X):
...     print("%s %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```

Here is a visualization of the cross-validation behavior. Note that `KFold` is not affected by classes or groups.



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]
```

Repeated K-Fold

RepeatedKFold repeats K-Fold n times. It can be used when one requires to run *KFold* n times, producing different splits in each repetition.

Example of 2-fold K-Fold repeated 2 times:

```
>>> import numpy as np
>>> from sklearn.model_selection import RepeatedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> random_state = 12883823
>>> rkf = RepeatedKFold(n_splits=2, n_repeats=2, random_state=random_state)
>>> for train, test in rkf.split(X):
...     print("%s %s" % (train, test))
...
[2 3] [0 1]
[0 1] [2 3]
[0 2] [1 3]
[1 3] [0 2]
```

Similarly, *RepeatedStratifiedKFold* repeats Stratified K-Fold n times with different randomization in each repetition.

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.model_selection import LeaveOneOut
>>> X = [1, 2, 3, 4]
>>> loo = LeaveOneOut()
>>> for train, test in loo.split(X):
...     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.

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.model_selection import LeavePOut

>>> X = np.ones(4)
>>> lpo = LeavePOut(p=2)
>>> for train, test in lpo.split(X):
...     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]
```

Random permutations cross-validation a.k.a. Shuffle & Split

ShuffleSplit

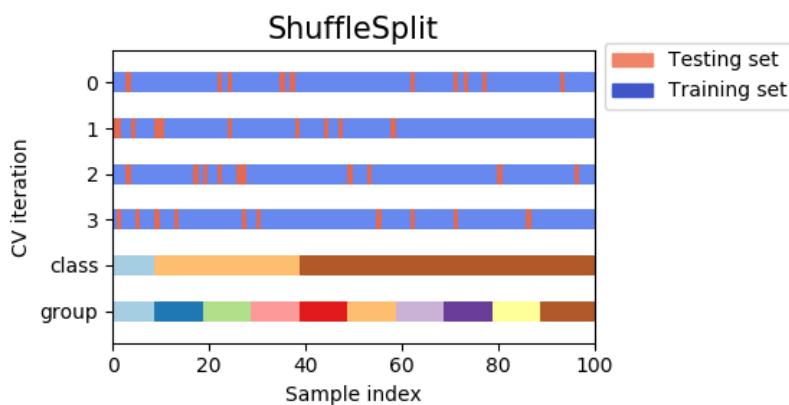
The *ShuffleSplit* iterator will generate a user defined number of independent train / test dataset splits. Samples are first shuffled and then split into a pair of train and test sets.

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:

```
>>> from sklearn.model_selection import ShuffleSplit
>>> X = np.arange(10)
>>> ss = ShuffleSplit(n_splits=5, test_size=0.25,
...     random_state=0)
>>> for train_index, test_index in ss.split(X):
...     print("%s %s" % (train_index, test_index))
[9 1 6 7 3 0 5] [2 8 4]
[2 9 8 0 6 7 4] [3 5 1]
[4 5 1 0 6 9 7] [2 3 8]
[2 7 5 8 0 3 4] [6 1 9]
[4 1 0 6 8 9 3] [5 2 7]
```

Here is a visualization of the cross-validation behavior. Note that `ShuffleSplit` is not affected by classes or groups.



`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 on each side of the train / test split.

Cross-validation iterators with stratification based on class labels.

Some classification problems can exhibit a large imbalance in the distribution of the target classes: for instance there could be several times more negative samples than positive samples. In such cases it is recommended to use stratified sampling as implemented in `StratifiedKFold` and `StratifiedShuffleSplit` to ensure that relative class frequencies is approximately preserved in each train and validation fold.

Stratified k-fold

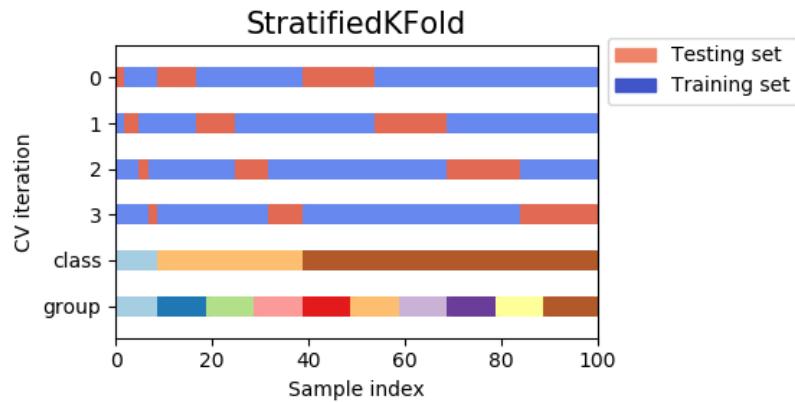
`StratifiedKFold` is a variation of *k-fold* which returns *stratified* folds: each set contains approximately the same percentage of samples of each target class as the complete set.

Example of stratified 3-fold cross-validation on a dataset with 10 samples from two slightly unbalanced classes:

```
>>> from sklearn.model_selection import StratifiedKFold
>>> X = np.ones(10)
>>> y = [0, 0, 0, 0, 1, 1, 1, 1, 1]
>>> skf = StratifiedKFold(n_splits=3)
>>> for train, test in skf.split(X, y):
```

```
...     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]
```

Here is a visualization of the cross-validation behavior.

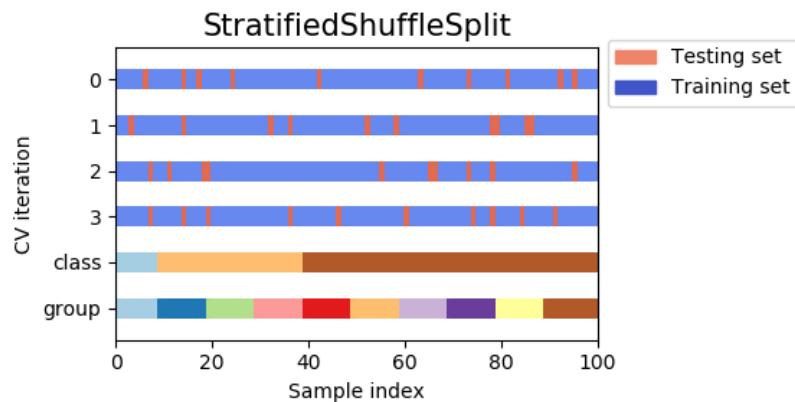


`RepeatedStratifiedKFold` can be used to repeat Stratified K-Fold n times with different randomization in each repetition.

Stratified Shuffle Split

`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.

Here is a visualization of the cross-validation behavior.



Cross-validation iterators for grouped data.

The i.i.d. assumption is broken if the underlying generative process yield groups of dependent samples.

Such a grouping of data is domain specific. An example would be when there is medical data collected from multiple patients, with multiple samples taken from each patient. And such data is likely to be dependent on the individual group. In our example, the patient id for each sample will be its group identifier.

In this case we would like to know if a model trained on a particular set of groups generalizes well to the unseen groups. To measure this, we need to ensure that all the samples in the validation fold come from groups that are not represented at all in the paired training fold.

The following cross-validation splitters can be used to do that. The grouping identifier for the samples is specified via the `groups` parameter.

Group k-fold

`GroupKFold` is a variation of k-fold which ensures that the same group is not represented in both testing and training sets. For example if the data is obtained from different subjects with several samples per-subject and if the model is flexible enough to learn from highly person specific features it could fail to generalize to new subjects. `GroupKFold` makes it possible to detect this kind of overfitting situations.

Imagine you have three subjects, each with an associated number from 1 to 3:

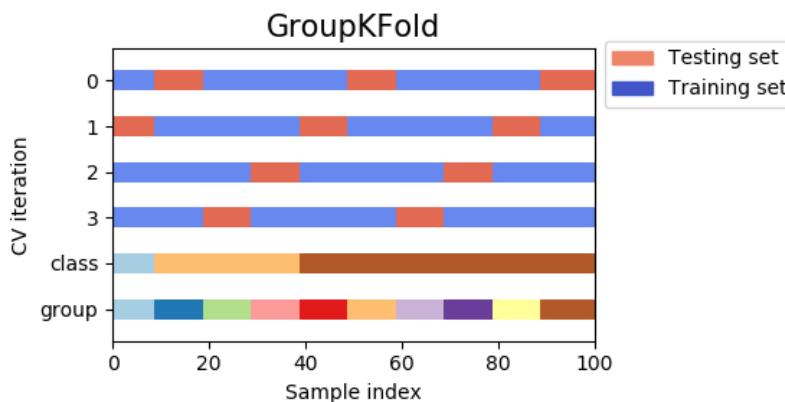
```
>>> from sklearn.model_selection import GroupKFold

>>> X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 8.8, 9, 10]
>>> y = ["a", "b", "b", "b", "c", "c", "c", "d", "d", "d"]
>>> groups = [1, 1, 1, 2, 2, 2, 3, 3, 3, 3]

>>> gkf = GroupKFold(n_splits=3)
>>> for train, test in gkf.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
[0 1 2 3 4 5] [6 7 8 9]
[0 1 2 6 7 8 9] [3 4 5]
[3 4 5 6 7 8 9] [0 1 2]
```

Each subject is in a different testing fold, and the same subject is never in both testing and training. Notice that the folds do not have exactly the same size due to the imbalance in the data.

Here is a visualization of the cross-validation behavior.



Leave One Group Out

LeaveOneGroupOut is a cross-validation scheme which holds out the samples according to a third-party provided array of integer groups. This group 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 group.

For example, in the cases of multiple experiments, *LeaveOneGroupOut* 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.model_selection import LeaveOneGroupOut

>>> X = [1, 5, 10, 50, 60, 70, 80]
>>> y = [0, 1, 1, 2, 2, 2, 2]
>>> groups = [1, 1, 2, 2, 3, 3, 3]
>>> logo = LeaveOneGroupOut()
>>> for train, test in logo.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
[2 3 4 5 6] [0 1]
[0 1 4 5 6] [2 3]
[0 1 2 3] [4 5 6]
```

Another common application is to use time information: for instance the groups could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

Leave P Groups Out

LeavePGroupsOut is similar as *LeaveOneGroupOut*, but removes samples related to P groups for each training/test set.

Example of Leave-2-Group Out:

```
>>> from sklearn.model_selection import LeavePGroupsOut

>>> X = np.arange(6)
>>> y = [1, 1, 1, 2, 2, 2]
>>> groups = [1, 1, 2, 2, 3, 3]
>>> lpg = LeavePGroupsOut(n_groups=2)
>>> for train, test in lpg.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
[4 5] [0 1 2 3]
[2 3] [0 1 4 5]
[0 1] [2 3 4 5]
```

Group Shuffle Split

The *GroupShuffleSplit* iterator behaves as a combination of *ShuffleSplit* and *LeavePGroupsOut*, and generates a sequence of randomized partitions in which a subset of groups are held out for each split.

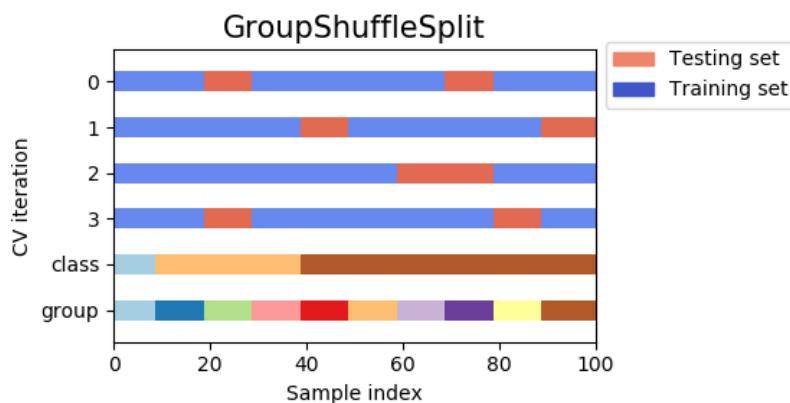
Here is a usage example:

```
>>> from sklearn.model_selection import GroupShuffleSplit

>>> X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 0.001]
>>> y = ["a", "b", "b", "b", "c", "c", "c", "a"]
```

```
>>> groups = [1, 1, 2, 2, 3, 3, 4, 4]
>>> gss = GroupShuffleSplit(n_splits=4, test_size=0.5, random_state=0)
>>> for train, test in gss.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
...
[0 1 2 3] [4 5 6 7]
[2 3 6 7] [0 1 4 5]
[2 3 4 5] [0 1 6 7]
[4 5 6 7] [0 1 2 3]
```

Here is a visualization of the cross-validation behavior.



This class is useful when the behavior of `LeavePGroupsOut` is desired, but the number of groups is large enough that generating all possible partitions with P groups withheld would be prohibitively expensive. In such a scenario, `GroupShuffleSplit` provides a random sample (with replacement) of the train / test splits generated by `LeavePGroupsOut`.

Predefined Fold-Splits / Validation-Sets

For some datasets, a pre-defined split of the data into training- and validation fold or into several cross-validation folds already exists. Using `PredefinedSplit` it is possible to use these folds e.g. when searching for hyperparameters.

For example, when using a validation set, set the `test_fold` to 0 for all samples that are part of the validation set, and to -1 for all other samples.

Cross validation of time series data

Time series data is characterised by the correlation between observations that are near in time (*autocorrelation*). However, classical cross-validation techniques such as `KFold` and `ShuffleSplit` assume the samples are independent and identically distributed, and would result in unreasonable correlation between training and testing instances (yielding poor estimates of generalisation error) on time series data. Therefore, it is very important to evaluate our model for time series data on the “future” observations least like those that are used to train the model. To achieve this, one solution is provided by `TimeSeriesSplit`.

Time Series Split

`TimeSeriesSplit` is a variation of *k-fold* which returns first k folds as train set and the $(k + 1)$ th fold as test set. Note that unlike standard cross-validation methods, successive training sets are supersets of those that come before them. Also, it adds all surplus data to the first training partition, which is always used to train the model.

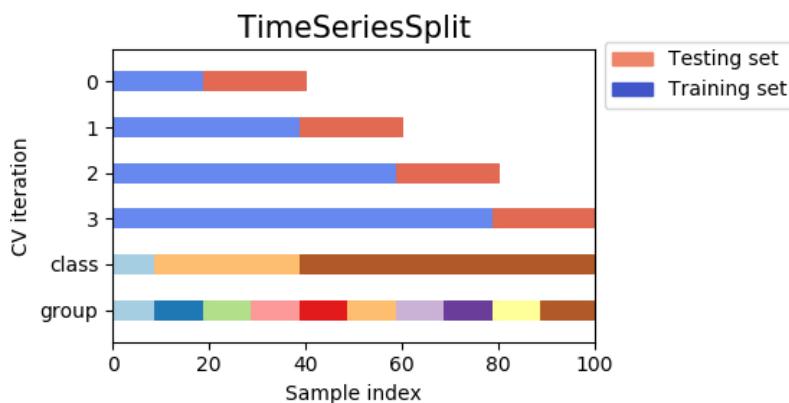
This class can be used to cross-validate time series data samples that are observed at fixed time intervals.

Example of 3-split time series cross-validation on a dataset with 6 samples:

```
>>> from sklearn.model_selection import TimeSeriesSplit

>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> tscv = TimeSeriesSplit(n_splits=3)
>>> print(tscv)
TimeSeriesSplit(max_train_size=None, n_splits=3)
>>> for train, test in tscv.split(X):
...     print("%s %s" % (train, test))
[0 1 2] [3]
[0 1 2 3] [4]
[0 1 2 3 4] [5]
```

Here is a visualization of the cross-validation behavior.



A note on shuffling

If the data ordering is not arbitrary (e.g. samples with the same class 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 get identical results for each split, set `random_state` to an integer.

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 of the next section: [Tuning the hyper-parameters of an estimator](#).

3.3.2 Tuning the hyper-parameters of an estimator

Hyper-parameters are parameters that are not directly learnt within estimators. In scikit-learn they are passed as arguments to the constructor of the estimator classes. Typical examples include `C`, `kernel` and `gamma` for Support Vector Classifier, `alpha` for Lasso, etc.

It is possible and recommended to search the hyper-parameter space for the best [cross validation](#) score.

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()
```

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](#).

Some models allow for specialized, efficient parameter search strategies, [outlined below](#). 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. After describing these tools we detail [best practice](#) applicable to both approaches.

Note that it is common that a small subset of those parameters can have a large impact on the predictive or computation performance of the model while others can be left to their default values. It is recommended to read the docstring of the estimator class to get a finer understanding of their expected behavior, possibly by reading the enclosed reference to the literature.

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.

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.
- See [Nested versus non-nested cross-validation](#) for an example of Grid Search within a cross validation loop on the iris dataset. This is the best practice for evaluating the performance of a model with grid search.
- See [Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV](#) for an example of `GridSearchCV` being used to evaluate multiple metrics simultaneously.
- See [Balance model complexity and cross-validated score](#) for an example of using `refit=callable` interface in `GridSearchCV`. The example shows how this interface adds certain amount of flexibility in identifying the “best” estimator. This interface can also be used in multiple metrics evaluation.

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':['balanced', 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` prior to version `scipy 0.16` 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`. However, beginning scikit-learn 0.18, the `sklearn`.

`model_selection`

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)

Tips for parameter search

Specifying an objective metric

By default, parameter search 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`, `RandomizedSearchCV` and many of the specialized cross-validation tools described below. See [The scoring parameter: defining model evaluation rules](#) for more details.

Specifying multiple metrics for evaluation

`GridSearchCV` and `RandomizedSearchCV` allow specifying multiple metrics for the `scoring` parameter.

Multimetric scoring can either be specified as a list of strings of predefined scores names or a dict mapping the scorer name to the scorer function and/or the predefined scorer name(s). See [Using multiple metric evaluation](#) for more details.

When specifying multiple metrics, the `refit` parameter must be set to the metric (string) for which the `best_params_` will be found and used to build the `best_estimator_` on the whole dataset. If the search should not be refit, set `refit=False`. Leaving `refit` to the default value `None` will result in an error when using multiple metrics.

See [Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV](#) for an example usage.

Composite estimators and parameter spaces

[Pipeline: chaining estimators](#) describes building composite estimators whose parameter space can be searched with these tools.

Model selection: development and evaluation

Model selection by evaluating various parameter settings 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 `train_test_split` utility function.

Parallelism

`GridSearchCV` and `RandomizedSearchCV` evaluate each parameter setting independently. Computations can be run in parallel if your OS supports it, by using the keyword `n_jobs=-1`. See function signature for more details.

Robustness to failure

Some parameter settings may result in a failure to `fit` one or more folds of the data. By default, this will cause the entire search to fail, even if some parameter settings could be fully evaluated. Setting `error_score=0` (or `=np.NaN`) will make the procedure robust to such failure, issuing a warning and setting the score for that fold to 0 (or `NaN`), but completing the search.

Alternatives to brute force parameter search

Model specific cross-validation

Some models can fit data for a range of values 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:

<code>linear_model.ElasticNetCV([l1_ratio, eps, ...])</code>	Elastic Net model with iterative fitting along a regularization path.
<code>linear_model.LarsCV([fit_intercept, ...])</code>	Cross-validated Least Angle Regression model.
<code>linear_model.LassoCV([eps, n_alphas, ...])</code>	Lasso linear model with iterative fitting along a regularization path.
<code>linear_model.LassoLarsCV([fit_intercept, ...])</code>	Cross-validated Lasso, using the LARS algorithm.
<code>linear_model.LogisticRegressionCV([Cs, ...])</code>	Logistic Regression CV (aka logit, MaxEnt) classifier.
<code>linear_model.MultiTaskElasticNetCV([...])</code>	Multi-task L1/L2 ElasticNet with built-in cross-validation.
<code>linear_model.MultiTaskLassoCV([eps, ...])</code>	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.
<code>linear_model.OrthogonalMatchingPursuitCV([Cross-validated Orthogonal Matching Pursuit model (OMP).])</code>	Cross-validated Orthogonal Matching Pursuit model (OMP).
<code>linear_model.RidgeCV([alphas, ...])</code>	Ridge regression with built-in cross-validation.

Continued on next page

Table 3.1 – continued from previous page

<code>linear_model.RidgeClassifierCV([alphas,</code>	Ridge classifier with built-in cross-validation.
<code>...])</code>	

`sklearn.linear_model.ElasticNetCV`

```
class sklearn.linear_model.ElasticNetCV(l1_ratio=0.5,    eps=0.001,    n_alphas=100,    al-
phas=None,    fit_intercept=True,    normalize=False,
precompute='auto',    max_iter=1000,    tol=0.0001,
cv='warn',    copy_X=True,    verbose=0,    n_jobs=None,
positive=False,    random_state=None,    selec-
tion='cyclic')
```

Elastic Net model with iterative fitting along a regularization path.

See glossary entry for [cross-validation estimator](#).

Read more in the [User Guide](#).

Parameters

l1_ratio [float or array of floats, optional] float between 0 and 1 passed to ElasticNet (scaling between l1 and l2 penalties). For `l1_ratio = 0` the penalty is an L2 penalty. For `l1_ratio = 1` it is an L1 penalty. For $0 < \text{l1_ratio} < 1$, the penalty is a combination of L1 and L2. This parameter can be a list, in which case the different values are tested by cross-validation and the one giving the best prediction score is used. Note that a good choice of list of values for `l1_ratio` is often to put more values close to 1 (i.e. Lasso) and less close to 0 (i.e. Ridge), as in `[.1, .5, .7, .9, .95, .99, 1]`

eps [float, optional] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path, used for each `l1_ratio`.

alphas [numpy array, optional] List of alphas where to compute the models. If None alphas are set automatically

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

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use [`sklearn.preprocessing.StandardScaler`](#) before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | ‘auto’ | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to ‘auto’ let us decide. The Gram matrix can also be passed as argument.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,

- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer [User Guide](#) for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

copy_X [boolean, optional, default True] If `True`, `X` will be copied; else, it may be overwritten.

verbose [bool or integer] Amount of verbosity.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See [Glossary](#) for more details.

positive [bool, optional] When set to `True`, forces the coefficients to be positive.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, `random_state` is the seed used by the random number generator; If RandomState instance, `random_state` is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection == 'random'`.

selection [str, default ‘cyclic’] If set to ‘random’, a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to ‘random’) often leads to significantly faster convergence especially when `tol` is higher than 1e-4.

Attributes

alpha_ [float] The amount of penalization chosen by cross validation

l1_ratio_ [float] The compromise between l1 and l2 penalization chosen by cross validation

coef_ [array, shape (n_features,) | (n_targets, n_features)] Parameter vector (w in the cost function formula),

intercept_ [float | array, shape (n_targets, n_features)] Independent term in the decision function.

mse_path_ [array, shape (n_l1_ratio, n_alpha, n_folds)] Mean square error for the test set on each fold, varying `l1_ratio` and `alpha`.

alphas_ [numpy array, shape (n_alphas,) or (n_l1_ratio, n_alphas)] The grid of alphas used for fitting, for each `l1_ratio`.

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance for the optimal alpha.

See also:

[`enet_path`](#)

[`ElasticNet`](#)

Notes

For an example, see [examples/linear_model/plot_lasso_selection.py](#).

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

The parameter l1_ratio corresponds to alpha in the glmnet R package while alpha corresponds to the lambda parameter in glmnet. More specifically, the optimization objective is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

If you are interested in controlling the L1 and L2 penalty separately, keep in mind that this is equivalent to:

```
a * L1 + b * L2
```

for:

```
alpha = a + b and l1_ratio = a / (a + b).
```

Examples

```
>>> from sklearn.linear_model import ElasticNetCV
>>> from sklearn.datasets import make_regression

>>> X, y = make_regression(n_features=2, random_state=0)
>>> regr = ElasticNetCV(cv=5, random_state=0)
>>> regr.fit(X, y)
ElasticNetCV(alphas=None, copy_X=True, cv=5, eps=0.001, fit_intercept=True,
             l1_ratio=0.5, max_iter=1000, n_alphas=100, n_jobs=None,
             normalize=False, positive=False, precompute='auto', random_state=0,
             selection='cyclic', tol=0.0001, verbose=0)
>>> print(regr.alpha_)
0.199...
>>> print(regr.intercept_)
0.398...
>>> print(regr.predict([[0, 0]]))
[0.398...]
```

Methods

<code>fit(self, X, y)</code>	Fit linear model with coordinate descent
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>path(X, y[, l1_ratio, eps, n_alphas, ...])</code>	Compute elastic net path with coordinate descent
<code>predict(self, X)</code>	Predict using the linear model
<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

```
__init__(self, l1_ratio=0.5, eps=0.001, n_alphas=100, alphas=None, fit_intercept=True, normalize=False, precompute='auto', max_iter=1000, tol=0.0001, cv='warn', copy_X=True, verbose=0, n_jobs=None, positive=False, random_state=None, selection='cyclic')
```

```
fit(self, X, y)
Fit linear model with coordinate descent
```

Fit is on grid of alphas and best alpha estimated by cross-validation.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If y is mono-output, X can be sparse.

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values

get_params (self, deep=True)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

static path (X, y, l1_ratio=0.5, eps=0.001, n_alphas=100, alphas=None, precompute='auto', Xy=None, copy_X=True, coef_init=None, verbose=False, return_n_iter=False, positive=False, check_input=True, **params)

Compute elastic net path with coordinate descent

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

$$\|W\|_{21} = \sum_i \sqrt{\sum_j w_{ij}^2}$$

i.e. the sum of norm of each row.

Read more in the [User Guide](#).

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If y is mono-output then X can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values

l1_ratio [float, optional] float between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). l1_ratio=1 corresponds to the Lasso

eps [float] Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically

precompute [True | False | ‘auto’ | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to ‘auto’ let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] $Xy = \text{np.dot}(X.T, y)$ that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when y .ndim == 1).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

[`MultiTaskElasticNet`](#)

[`MultiTaskElasticNetCV`](#)

[`ElasticNet`](#)

[`ElasticNetCV`](#)

Notes

For an example, see [`examples/linear_model/plot_lasso_coordinate_descent_path.py`](#).

predict (*self*, *X*)

Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (self, X, y, sample_weight=None)

Returns the coefficient of determination R² of the prediction.

The coefficient R² is defined as $(1 - u/v)$, where u is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y , disregarding the input features, would get a R² score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R² of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer 'r2' uses `multioutput='uniform_average'`).

set_params (self, **params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns

self

sklearn.linear_model.LarsCV

```
class sklearn.linear_model.LarsCV(fit_intercept=True,      verbose=False,      max_iter=500,
                                   normalize=True,      precompute='auto',
                                   cv='warn',          max_n_alphas=1000,      n_jobs=None,
                                   eps=2.220446049250313e-16, copy_X=True, positive=False)
```

Cross-validated Least Angle Regression model.

See glossary entry for [cross-validation estimator](#).

Read more in the [User Guide](#).

Parameters

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

verbose [boolean or integer, optional] Sets the verbosity amount

max_iter [integer, optional] Maximum number of iterations to perform.

normalize [boolean, optional, default True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use `sklearn.preprocessing.StandardScaler` before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | ‘auto’ | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to ‘auto’ let us decide. The Gram matrix cannot be passed as argument since we will use only subsets of X.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- `CV splitter`,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer [User Guide](#) for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

max_n_alphas [integer, optional] The maximum number of points on the path used to compute the residuals in the cross-validation

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See [Glossary](#) for more details.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

positive [boolean (default=False)] Restrict coefficients to be ≥ 0 . Be aware that you might want to remove `fit_intercept` which is set True by default.

Deprecated since version 0.20: The option is broken and deprecated. It will be removed in v0.22.

Attributes

coef_ [array, shape (n_features,)] parameter vector (w in the formulation formula)

intercept_ [float] independent term in decision function

coef_path_ [array, shape (n_features, n_alphas)] the varying values of the coefficients along the path

alpha_ [float] the estimated regularization parameter alpha

alphas_ [array, shape (n_alphas,)] the different values of alpha along the path

cv_alphas_ [array, shape (n_cv_alphas,)] all the values of alpha along the path for the different folds

`mse_path_` [array, shape (n_folds, n_cv_alphas)] the mean square error on left-out for each fold along the path (alpha values given by `cv_alphas`)

`n_iter_` [array-like or int] the number of iterations run by Lars with the optimal alpha.

See also:

[`lars_path`](#), [`LassoLars`](#), [`LassoLarsCV`](#)

Examples

```
>>> from sklearn.linear_model import LarsCV
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_samples=200, noise=4.0, random_state=0)
>>> reg = LarsCV(cv=5).fit(X, y)
>>> reg.score(X, y)
0.9996...
>>> reg.alpha_
0.0254...
>>> reg.predict(X[:1, :])
array([154.0842...])
```

Methods

<code>fit(self, X, y)</code>	Fit the model using X, y as training data.
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict using the linear model
<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, fit_intercept=True, verbose=False, max_iter=500, normalize=True, precompute='auto', cv='warn', max_n_alphas=1000, n_jobs=None, eps=2.220446049250313e-16, copy_X=True, positive=False)`

`fit (self, X, y)`
Fit the model using X, y as training data.

Parameters

X [array-like, shape (n_samples, n_features)] Training data.

y [array-like, shape (n_samples,)] Target values.

Returns

self [object] returns an instance of self.

`get_params (self, deep=True)`
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)

Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the coefficient of determination R² of the prediction.

The coefficient R² is defined as $(1 - u/v)$, where *u* is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and *v* is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R² score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R² of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer 'r2' uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns

self

sklearn.linear_model.LassoCV

```
class sklearn.linear_model.LassoCV(eps=0.001, n_alphas=100, alphas=None, fit_intercept=True,
                                    normalize=False, precompute='auto', max_iter=1000,
                                    tol=0.0001, copy_X=True, cv='warn', verbose=False,
                                    n_jobs=None, positive=False, random_state=None, selection='cyclic')
```

Lasso linear model with iterative fitting along a regularization path.

See glossary entry for [cross-validation estimator](#).

The best model is selected by cross-validation.

The optimization objective for Lasso is:

$$(1 / (2 * n_samples)) * ||y - Xw||^2 + \alpha * ||w||_1$$

Read more in the [User Guide](#).

Parameters

eps [float, optional] Length of the path. `eps=1e-3` means that $\alpha_{\min} / \alpha_{\max} = 1e-3$.

n_alphas [int, optional] Number of alphas along the regularization path

alphas [numpy array, optional] List of alphas where to compute the models. If `None` alphas are set automatically

fit_intercept [boolean, default True] whether to calculate the intercept for this model. If set to `false`, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to `False`. If `True`, the regressors `X` will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use [`sklearn.preprocessing.StandardScaler`](#) before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to '`auto`' let us decide. The Gram matrix can also be passed as argument.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

copy_X [boolean, optional, default True] If `True`, `X` will be copied; else, it may be overwritten.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for `cv` are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds,
- [CV splitter](#),
- An iterable yielding (train, test) splits as arrays of indices.

For integer/`None` inputs, `KFold` is used.

Refer [User Guide](#) for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if `None` will change from 3-fold to 5-fold in v0.22.

verbose [bool or integer] Amount of verbosity.

n_jobs [int or `None`, optional (default=`None`)] Number of CPUs to use during the cross validation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See [Glossary](#) for more details.

positive [bool, optional] If positive, restrict regression coefficients to be positive

random_state [int, RandomState instance or `None`, optional, default `None`] The seed of the pseudo random number generator that selects a random feature to update. If int, `random_state` is the seed used by the random number generator; If RandomState instance, `random_state` is the random number generator; If `None`, the random number generator is the RandomState instance used by `np.random`. Used when `selection == 'random'`.

selection [str, default ‘cyclic’] If set to ‘random’, a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to ‘random’) often leads to significantly faster convergence especially when `tol` is higher than `1e-4`.

Attributes

alpha_ [float] The amount of penalization chosen by cross validation

coef_ [array, shape (n_features,) | (n_targets, n_features)] parameter vector (w in the cost function formula)

intercept_ [float | array, shape (n_targets,)] independent term in decision function.

mse_path_ [array, shape (n_alphas, n_folds)] mean square error for the test set on each fold, varying alpha

alphas_ [numpy array, shape (n_alphas,)] The grid of alphas used for fitting

dual_gap_ [ndarray, shape ()] The dual gap at the end of the optimization for the optimal alpha (`alpha_`).

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance for the optimal alpha.

See also:

[`LarsPath`](#)

[`LassoPath`](#)

[`LassoLars`](#)

[`Lasso`](#)

[`LassoLarsCV`](#)

Notes

For an example, see [`examples/linear_model/plot_lasso_model_selection.py`](#).

To avoid unnecessary memory duplication the `X` argument of the `fit` method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn.linear_model import LassoCV
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(noise=4, random_state=0)
>>> reg = LassoCV(cv=5, random_state=0).fit(X, y)
>>> reg.score(X, y)
0.9993...
>>> reg.predict(X[:1, :])
array([-78.4951...])
```

Methods

<code>fit(self, X, y)</code>	Fit linear model with coordinate descent
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>path(X, y[, eps, n_alphas, alphas, ...])</code>	Compute Lasso path with coordinate descent
<code>predict(self, X)</code>	Predict using the linear model
<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, eps=0.001, n_alphas=100, alphas=None, fit_intercept=True, normalize=False, precompute='auto', max_iter=1000, tol=0.0001, copy_X=True, cv='warn', verbose=False, n_jobs=None, positive=False, random_state=None, selection='cyclic')`

`fit(self, X, y)`

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

Parameters

`X` [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output, `X` can be sparse.

`y` [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values

`get_params(self, deep=True)`

Get parameters for this estimator.

Parameters

`deep` [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

`params` [mapping of string to any] Parameter names mapped to their values.

`static path(X, y, eps=0.001, n_alphas=100, alphas=None, precompute='auto', Xy=None, copy_X=True, coef_init=None, verbose=False, return_n_iter=False, positive=False, **params)`

Compute Lasso path with coordinate descent

The Lasso optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^2_Fro + alpha * ||W||_21
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the [User Guide](#).

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If y is mono-output then X can be sparse.

y [ndarray, shape (n_samples,), or (n_samples, n_outputs)] Target values

eps [float, optional] Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically

precompute [True | False | ‘auto’ | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to ‘auto’ let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when y.ndim == 1).

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha.

See also:

[lars_path](#)
[Lasso](#)
[LassoLars](#)
[LassoCV](#)
[LassoLarsCV](#)
[sklearn.decomposition.sparse_encode](#)

Notes

For an example, see [examples/linear_model/plot_lasso_coordinate_descent_path.py](#).

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Note that in certain cases, the Lars solver may be significantly faster to implement this functionality. In particular, linear interpolation can be used to retrieve model coefficients between the values output by lars_path

Examples

Comparing lasso_path and lars_path with interpolation:

```
>>> X = np.array([[1, 2, 3.1], [2.3, 5.4, 4.3]]).T
>>> y = np.array([1, 2, 3.1])
>>> # Use lasso_path to compute a coefficient path
>>> _, coef_path, _ = lasso_path(X, y, alphas=[5., 1., .5])
>>> print(coef_path)
[[0.          0.        0.46874778]
 [0.2159048  0.4425765  0.23689075]]
```

```
>>> # Now use lars_path and 1D linear interpolation to compute the
>>> # same path
>>> from sklearn.linear_model import lars_path
>>> alphas, active, coef_path_lars = lars_path(X, y, method='lasso')
>>> from scipy import interpolate
>>> coef_path_continuous = interpolate.interp1d(alphas[::-1],
...                                              coef_path_lars[:, ::-1])
>>> print(coef_path_continuous([5., 1., .5]))
[[0.          0.        0.46915237]
 [0.2159048  0.4425765  0.23668876]]
```

`predict(self, X)`

Predict using the linear model

Parameters

`X` [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

`C` [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer 'r2' uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns

self

Examples using `sklearn.linear_model.LassoCV`

- *Cross-validation on diabetes Dataset Exercise*
- *Feature selection using SelectFromModel and LassoCV*
- *Lasso model selection: Cross-Validation / AIC / BIC*

`sklearn.linear_model.LassoLarsCV`

```
class sklearn.linear_model.LassoLarsCV(fit_intercept=True, verbose=False, max_iter=500,
                                         normalize=True, precompute='auto',
                                         cv='warn', max_n_alphas=1000, n_jobs=None,
                                         eps=2.220446049250313e-16, copy_X=True, positive=False)
```

Cross-validated Lasso, using the LARS algorithm.

See glossary entry for [cross-validation estimator](#).

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2 + alpha * ||w||_1
```

Read more in the [User Guide](#).

Parameters

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

verbose [boolean or integer, optional] Sets the verbosity amount

max_iter [integer, optional] Maximum number of iterations to perform.

normalize [boolean, optional, default True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors `X` will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use `sklearn.preprocessing.StandardScaler` before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | ‘auto’] Whether to use a precomputed Gram matrix to speed up calculations. If set to ‘auto’ let us decide. The Gram matrix cannot be passed as argument since we will use only subsets of `X`.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- [CV splitter](#),
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer [User Guide](#) for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

max_n_alphas [integer, optional] The maximum number of points on the path used to compute the residuals in the cross-validation

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See [Glossary](#) for more details.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

copy_X [boolean, optional, default True] If True, `X` will be copied; else, it may be overwritten.

positive [boolean (default=False)] Restrict coefficients to be ≥ 0 . Be aware that you might want to remove `fit_intercept` which is set True by default. Under the positive restriction the model coefficients do not converge to the ordinary-least-squares solution for small values of alpha. Only coefficients up to the smallest alpha value (`alphas_[alphas_ > 0.] . min()` when `fit_path=True`) reached by the stepwise Lars-Lasso algorithm are typically in congruence with the solution of the coordinate descent Lasso estimator. As a consequence

using LassoLarsCV only makes sense for problems where a sparse solution is expected and/or reached.

Attributes

coef_ [array, shape (n_features,)] parameter vector (w in the formulation formula)

intercept_ [float] independent term in decision function.

coef_path_ [array, shape (n_features, n_alphas)] the varying values of the coefficients along the path

alpha_ [float] the estimated regularization parameter alpha

alphas_ [array, shape (n_alphas,)] the different values of alpha along the path

cv_alphas_ [array, shape (n_cv_alphas,)] all the values of alpha along the path for the different folds

mse_path_ [array, shape (n_folds, n_cv_alphas)] the mean square error on left-out for each fold along the path (alpha values given by cv_alphas)

n_iter_ [array-like or int] the number of iterations run by Lars with the optimal alpha.

See also:

[lars_path](#), [LassoLars](#), [LarsCV](#), [LassoCV](#)

Notes

The object solves the same problem as the LassoCV object. However, unlike the LassoCV, it find the relevant alphas values by itself. In general, because of this property, it will be more stable. However, it is more fragile to heavily multicollinear datasets.

It is more efficient than the LassoCV if only a small number of features are selected compared to the total number, for instance if there are very few samples compared to the number of features.

Examples

```
>>> from sklearn.linear_model import LassoLarsCV
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(noise=4.0, random_state=0)
>>> reg = LassoLarsCV(cv=5).fit(X, y)
>>> reg.score(X, y)
0.9992...
>>> reg.alpha_
0.0484...
>>> reg.predict(X[:, 1])
array([-77.8723...])
```

Methods

[`fit`\(self, X, y\)](#) Fit the model using X, y as training data.

[`get_params`\(self\[, deep\]\)](#) Get parameters for this estimator.

[`predict`\(self, X\)](#) Predict using the linear model

Continued on next page

Table 3.5 – continued from previous page

<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, fit_intercept=True, verbose=False, max_iter=500, normalize=True, precompute='auto', cv='warn', max_n_alphas=1000, n_jobs=None, eps=2.220446049250313e-16, copy_X=True, positive=False)`

`fit(self, X, y)`

Fit the model using X, y as training data.

Parameters

`X` [array-like, shape (n_samples, n_features)] Training data.

`y` [array-like, shape (n_samples,)] Target values.

Returns

`self` [object] returns an instance of self.

`get_params(self, deep=True)`

Get parameters for this estimator.

Parameters

`deep` [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

`params` [mapping of string to any] Parameter names mapped to their values.

`predict(self, X)`

Predict using the linear model

Parameters

`X` [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

`C` [array, shape (n_samples,)] Returns predicted values.

`score(self, X, y, sample_weight=None)`

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as $(1 - u/v)$, where u is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

`X` [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator.

`y` [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

`sample_weight` [array-like, shape = [n_samples], optional] Sample weights.

Returns

`score` [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer '`r2`' uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

Examples using `sklearn.linear_model.LassoLarsCV`

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

`sklearn.linear_model.LogisticRegressionCV`

```
class sklearn.linear_model.LogisticRegressionCV(Cs=10, fit_intercept=True, cv='warn',
                                                dual=False, penalty='l2', scoring=None,
                                                solver='lbfgs', tol=0.0001,
                                                max_iter=100, class_weight=None,
                                                n_jobs=None, verbose=0, refit=True,
                                                intercept_scaling=1.0, multi_class='warn',
                                                random_state=None, l1_ratio=None)
```

Logistic Regression CV (aka logit, MaxEnt) classifier.

See glossary entry for [cross-validation estimator](#).

This class implements logistic regression using liblinear, newton-cg, sag or lbfgs optimizer. The newton-cg, sag and lbfgs solvers support only L2 regularization with primal formulation. The liblinear solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty. Elastic-Net penalty is only supported by the saga solver.

For the grid of `Cs` values and `l1_ratios` values, the best hyperparameter is selected by the cross-validator `StratifiedKFold`, but it can be changed using the `cv` parameter. The ‘newton-cg’, ‘sag’, ‘saga’ and ‘lbfgs’ solvers can warm-start the coefficients (see [Glossary](#)).

Read more in the [User Guide](#).

Parameters

Cs [list of floats or int, optional (default=10)] Each of the values in Cs describes the inverse of regularization strength. If Cs is as an int, then a grid of Cs values are chosen in a logarithmic scale between 1e-4 and 1e4. Like in support vector machines, smaller values specify stronger regularization.

fit_intercept [bool, optional (default=True)] Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

cv [int or cross-validation generator, optional (default=None)] The default cross-validation generator used is Stratified K-Folds. If an integer is provided, then it is the number of folds used. See the module `sklearn.model_selection` module for the list of possible cross-validation objects.

Changed in version 0.20: cv default value if None will change from 3-fold to 5-fold in v0.22.

dual [bool, optional (default=False)] Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.

penalty [str, ‘l1’, ‘l2’, or ‘elasticnet’, optional (default=’l2’)] Used to specify the norm used in the penalization. The ‘newton-cg’, ‘sag’ and ‘lbfgs’ solvers support only l2 penalties. ‘elasticnet’ is only supported by the ‘saga’ solver.

scoring [string, callable, or None, optional (default=None)] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`. For a list of scoring functions that can be used, look at `sklearn.metrics`. The default scoring option used is ‘accuracy’.

solver [str, {‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, ‘saga’}, optional (default=’lbfgs’)] Algorithm to use in the optimization problem.

- For small datasets, ‘liblinear’ is a good choice, whereas ‘sag’ and ‘saga’ are faster for large ones.
- For multiclass problems, only ‘newton-cg’, ‘sag’, ‘saga’ and ‘lbfgs’ handle multinomial loss; ‘liblinear’ is limited to one-versus-rest schemes.
- ‘newton-cg’, ‘lbfgs’ and ‘sag’ only handle L2 penalty, whereas ‘liblinear’ and ‘saga’ handle L1 penalty.
- ‘liblinear’ might be slower in LogisticRegressionCV because it does not handle warm-starting.

Note that ‘sag’ and ‘saga’ fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from `sklearn.preprocessing`.

New in version 0.17: Stochastic Average Gradient descent solver.

New in version 0.19: SAGA solver.

tol [float, optional (default=1e-4)] Tolerance for stopping criteria.

max_iter [int, optional (default=100)] Maximum number of iterations of the optimization algorithm.

class_weight [dict or ‘balanced’, optional (default=None)] Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one.

The “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

New in version 0.17: class_weight == ‘balanced’

n_jobs [int or None, optional (default=None)] Number of CPU cores used during the cross-validation loop. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See [Glossary](#) for more details.

verbose [int, optional (default=0)] For the ‘liblinear’, ‘sag’ and ‘lbfgs’ solvers set verbose to any positive number for verbosity.

refit [bool, optional (default=True)] If set to True, the scores are averaged across all folds, and the coefs and the C that corresponds to the best score is taken, and a final refit is done using these parameters. Otherwise the coefs, intercepts and C that correspond to the best scores across folds are averaged.

intercept_scaling [float, optional (default=1)] Useful only when the solver ‘liblinear’ is used and `self.fit_intercept` is set to True. In this case, `x` becomes `[x, self.intercept_scaling]`, i.e. a “synthetic” feature with constant value equal to `intercept_scaling` is appended to the instance vector. The intercept becomes `intercept_scaling * synthetic_feature_weight`.

Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) `intercept_scaling` has to be increased.

multi_class [str, {‘ovr’, ‘multinomial’, ‘auto’}, optional (default=‘ovr’)] If the option chosen is ‘ovr’, then a binary problem is fit for each label. For ‘multinomial’ the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. ‘multinomial’ is unavailable when `solver=‘liblinear’`. ‘auto’ selects ‘ovr’ if the data is binary, or if `solver=‘liblinear’`, and otherwise selects ‘multinomial’.

New in version 0.18: Stochastic Average Gradient descent solver for ‘multinomial’ case.

Changed in version 0.20: Default will change from ‘ovr’ to ‘auto’ in 0.22.

random_state [int, RandomState instance or None, optional (default=None)] If int, `random_state` is the seed used by the random number generator; If RandomState instance, `random_state` is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

l1_ratio [list of float or None, optional (default=None)] The list of Elastic-Net mixing parameter, with $0 \leq l1_ratio \leq 1$. Only used if `penalty=‘elasticnet’`. A value of 0 is equivalent to using `penalty=‘l2’`, while 1 is equivalent to using `penalty=‘l1’`. For $0 < l1_ratio < 1$, the penalty is a combination of L1 and L2.

Attributes

classes_ [array, shape (n_classes,)] A list of class labels known to the classifier.

coef_ [array, shape (1, n_features) or (n_classes, n_features)] Coefficient of the features in the decision function.

`coef_` is of shape (1, n_features) when the given problem is binary.

intercept_ [array, shape (1,) or (n_classes,)] Intercept (a.k.a. bias) added to the decision function.

If `fit_intercept` is set to False, the intercept is set to zero. `intercept_` is of shape(1,) when the problem is binary.

Cs_ [array, shape (n_cs)] Array of C i.e. inverse of regularization parameter values used for cross-validation.

l1_ratios_ [array, shape (n_l1_ratios)] Array of l1_ratios used for cross-validation. If no l1_ratio is used (i.e. penalty is not ‘elasticnet’), this is set to [None]

coefs_paths_ [array, shape (n_folds, n_cs, n_features) or (n_folds, n_cs, n_features + 1)] dict with classes as the keys, and the path of coefficients obtained during cross-validating across each fold and then across each Cs after doing an OvR for the corresponding class as values. If the ‘multi_class’ option is set to ‘multinomial’, then the coefs_paths are the coefficients corresponding to each class. Each dict value has shape (n_folds, n_cs, n_features) or (n_folds, n_cs, n_features + 1) depending on whether the intercept is fit or not. If penalty='elasticnet', the shape is (n_folds, n_cs, n_l1_ratios_, n_features) or (n_folds, n_cs, n_l1_ratios_, n_features + 1).

scores_ [dict] dict with classes as the keys, and the values as the grid of scores obtained during cross-validating each fold, after doing an OvR for the corresponding class. If the ‘multi_class’ option given is ‘multinomial’ then the same scores are repeated across all classes, since this is the multinomial class. Each dict value has shape (n_folds, n_cs) or (n_folds, n_l1_ratios) if penalty='elasticnet'.

C_ [array, shape (n_classes,) or (n_classes - 1,)] Array of C that maps to the best scores across every class. If refit is set to False, then for each class, the best C is the average of the C’s that correspond to the best scores for each fold. C_ is of shape(n_classes,) when the problem is binary.

l1_ratio_ [array, shape (n_classes,) or (n_classes - 1,)] Array of l1_ratio that maps to the best scores across every class. If refit is set to False, then for each class, the best l1_ratio is the average of the l1_ratio’s that correspond to the best scores for each fold. l1_ratio_ is of shape(n_classes,) when the problem is binary.

n_iter_ [array, shape (n_classes, n_folds, n_cs) or (1, n_folds, n_cs)] Actual number of iterations for all classes, folds and Cs. In the binary or multinomial cases, the first dimension is equal to 1. If penalty='elasticnet', the shape is (n_classes, n_folds, n_l1_ratios) or (1, n_folds, n_l1_ratios).

See also:

[LogisticRegression](#)

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegressionCV
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegressionCV(cv=5, random_state=0,
...                                multi_class='multinomial').fit(X, y)
>>> clf.predict(X[:2, :])
array([0, 0])
>>> clf.predict_proba(X[:2, :]).shape
(2, 3)
>>> clf.score(X, y)
0.98...
```

Methods

<code>decision_function(self, X)</code>	Predict confidence scores for samples.
<code>densify(self)</code>	Convert coefficient matrix to dense array format.

Continued on next page

Table 3.6 – continued from previous page

<code>fit(self, X, y[, sample_weight])</code>	Fit the model according to the given training data.
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict class labels for samples in X.
<code>predict_log_proba(self, X)</code>	Log of probability estimates.
<code>predict_proba(self, X)</code>	Probability estimates.
<code>score(self, X, y[, sample_weight])</code>	Returns the score using the <code>scoring</code> option on the given test data and labels.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.
<code>sparsify(self)</code>	Convert coefficient matrix to sparse format.

`__init__(self, Cs=10, fit_intercept=True, cv='warn', dual=False, penalty='l2', scoring=None, solver='lbfgs', tol=0.0001, max_iter=100, class_weight=None, n_jobs=None, verbose=0, refit=True, intercept_scaling=1.0, multi_class='warn', random_state=None, l1_ratio=None)`

`decision_function(self, X)`

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

`X` [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

`array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes)` Confidence scores per (sample, class) combination. In the binary case, confidence score for `self.classes_[1]` where >0 means this class would be predicted.

`densify(self)`

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

`self` [estimator]

`fit(self, X, y, sample_weight=None)`

Fit the model according to the given training data.

Parameters

`X` [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

`y` [array-like, shape (n_samples,)] Target vector relative to X.

`sample_weight` [array-like, shape (n_samples,), optional] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

Returns

`self` [object]

`get_params(self, deep=True)`

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

`predict(self, X)`

Predict class labels for samples in X.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

`predict_log_proba(self, X)`

Log of probability estimates.

The returned estimates for all classes are ordered by the label of classes.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

T [array-like, shape = [n_samples, n_classes]] Returns the log-probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

`predict_proba(self, X)`

Probability estimates.

The returned estimates for all classes are ordered by the label of classes.

For a multi_class problem, if `multi_class` is set to be “multinomial” the softmax function is used to find the predicted probability of each class. Else use a one-vs-rest approach, i.e calculate the probability of each class assuming it to be positive using the logistic function. and normalize these values across all the classes.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

T [array-like, shape = [n_samples, n_classes]] Returns the probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

`score(self, X, y, sample_weight=None)`

Returns the score using the `scoring` option on the given test data and labels.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples,)] True labels for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] Score of `self.predict(X)` wrt. y.

set_params(self, **params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns**self****sparsify**(self)

Convert coefficient matrix to sparse format.

Converts the `coef_` member to a `scipy.sparse` matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual `numpy.ndarray` representation.

The `intercept_` member is not converted.

Returns**self** [estimator]**Notes**

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the `partial_fit` method (if any) will not work until you call `densify`.

sklearn.linear_model.MultiTaskElasticNetCV

```
class sklearn.linear_model.MultiTaskElasticNetCV(l1_ratio=0.5, eps=0.001, n_alphas=100,
                                             alphas=None,          fit_intercept=True,
                                             normalize=False,       max_iter=1000,
                                             tol=0.0001,           cv='warn', copy_X=True,
                                             verbose=0,            n_jobs=None, random_state=None, selection='cyclic')
```

Multi-task L1/L2 ElasticNet with built-in cross-validation.

See glossary entry for [cross-validation estimator](#).

The optimization objective for MultiTaskElasticNet is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the [User Guide](#).

Parameters

l1_ratio [float or array of floats] The ElasticNet mixing parameter, with $0 < l1_ratio \leq 1$. For $l1_ratio = 1$ the penalty is an L1/L2 penalty. For $l1_ratio = 0$ it is an L2 penalty. For $0 < l1_ratio < 1$, the penalty is a combination of L1/L2 and L2. This parameter can be a list, in which case the different values are tested by cross-validation and the one giving the best prediction score is used. Note that a good choice of list of values for $l1_ratio$ is often to put more values close to 1 (i.e. Lasso) and less close to 0 (i.e. Ridge), as in `[.1, .5, .7, .9, .95, .99, 1]`

eps [float, optional] Length of the path. $\text{eps}=1e-3$ means that $\text{alpha}_{\min} / \text{alpha}_{\max} = 1e-3$.

n_alphas [int, optional] Number of alphas along the regularization path

alphas [array-like, optional] List of alphas where to compute the models. If not provided, set automatically.

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

normalize [boolean, optional, default False] This parameter is ignored when **fit_intercept** is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use `sklearn.preprocessing.StandardScaler` before calling `fit` on an estimator with `normalize=False`.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds,
- `CV splitter`,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer [User Guide](#) for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

verbose [bool or integer] Amount of verbosity.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. Note that this is used only if multiple values for $l1_ratio$ are given. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See [Glossary](#) for more details.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, `random_state` is the seed used by the random number generator; If RandomState instance, `random_state` is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection == 'random'`.

selection [str, default ‘cyclic’] If set to ‘random’, a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to ‘random’) often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

intercept_ [array, shape (n_tasks,)] Independent term in decision function.

coef_ [array, shape (n_tasks, n_features)] Parameter vector (W in the cost function formula). Note that `coef_` stores the transpose of W , W^T .

alpha_ [float] The amount of penalization chosen by cross validation

mse_path_ [array, shape (n_alphas, n_folds) or (n_l1_ratio, n_alphas, n_folds)] mean square error for the test set on each fold, varying alpha

alphas_ [numpy array, shape (n_alphas,) or (n_l1_ratio, n_alphas)] The grid of alphas used for fitting, for each l1_ratio

l1_ratio_ [float] best l1_ratio obtained by cross-validation.

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance for the optimal alpha.

See also:

[MultiTaskElasticNet](#)

[ElasticNetCV](#)

[MultiTaskLassoCV](#)

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.MultiTaskElasticNetCV(cv=3)
>>> clf.fit([[0,0], [1, 1], [2, 2]],
...           [[0, 0], [1, 1], [2, 2]])
...
MultiTaskElasticNetCV(alphas=None, copy_X=True, cv=3, eps=0.001,
                      fit_intercept=True, l1_ratio=0.5, max_iter=1000, n_alphas=100,
                      n_jobs=None, normalize=False, random_state=None, selection='cyclic',
                      tol=0.0001, verbose=0)
>>> print(clf.coef_)
[[0.52875032 0.46958558]
 [0.52875032 0.46958558]]
>>> print(clf.intercept_)
[0.00166409 0.00166409]
```

Methods

<code>fit(self, X, y)</code>	Fit linear model with coordinate descent
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>path(X, y[, l1_ratio, eps, n_alphas, ...])</code>	Compute elastic net path with coordinate descent
<code>predict(self, X)</code>	Predict using the linear model
<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, l1_ratio=0.5, eps=0.001, n_alphas=100, alphas=None, fit_intercept=True, normalize=False, max_iter=1000, tol=0.0001, cv='warn', copy_X=True, verbose=0, n_jobs=None, random_state=None, selection='cyclic')`

`fit(self, X, y)`

Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

Parameters

`X` [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If y is mono-output, X can be sparse.

`y` [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values

`get_params(self, deep=True)`

Get parameters for this estimator.

Parameters

`deep` [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

`params` [mapping of string to any] Parameter names mapped to their values.

`static path(X, y, l1_ratio=0.5, eps=0.001, n_alphas=100, alphas=None, precompute='auto', Xy=None, copy_X=True, coef_init=None, verbose=False, return_n_iter=False, positive=False, check_input=True, **params)`

Compute elastic net path with coordinate descent

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

$$\|W\|_{21} = \sum_i \sqrt{\sum_j w_{ij}^2}$$

i.e. the sum of norm of each row.

Read more in the [User Guide](#).

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If y is mono-output then X can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values

l1_ratio [float, optional] float between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). l1_ratio=1 corresponds to the Lasso

eps [float] Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically

precompute [True | False | ‘auto’ | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to ‘auto’ let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when y.ndim == 1).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when return_n_iter is set to True).

See also:

`MultiTaskElasticNet`
`MultiTaskElasticNetCV`
`ElasticNet`
`ElasticNetCV`

Notes

For an example, see [examples/linear_model/plot_lasso_coordinate_descent_path.py](#).

predict (*self*, *X*)

Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as $(1 - u/v)$, where u is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y , disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer '`r2`' uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns**self****sklearn.linear_model.MultiTaskLassoCV**

```
class sklearn.linear_model.MultiTaskLassoCV(eps=0.001, n_alphas=100, alphas=None,
                                             fit_intercept=True, normalize=False,
                                             max_iter=1000, tol=0.0001, copy_X=True,
                                             cv='warn', verbose=False, n_jobs=None,
                                             random_state=None, selection='cyclic')
```

Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.

See glossary entry for [cross-validation estimator](#).

The optimization objective for MultiTaskLasso is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2 + alpha * ||W||_21
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the [User Guide](#).

Parameters

eps [float, optional] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`.

n_alphas [int, optional] Number of alphas along the regularization path

alphas [array-like, optional] List of alphas where to compute the models. If not provided, set automatically.

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

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors `X` will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use [sklearn.preprocessing.StandardScaler](#) before calling `fit` on an estimator with `normalize=False`.

max_iter [int, optional] The maximum number of iterations.

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

copy_X [boolean, optional, default True] If True, `X` will be copied; else, it may be overwritten.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for `cv` are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- [CV splitter](#),

- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer [User Guide](#) for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

verbose [bool or integer] Amount of verbosity.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. Note that this is used only if multiple values for `l1_ratio` are given. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See [Glossary](#) for more details.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, `random_state` is the seed used by the random number generator; If RandomState instance, `random_state` is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection == 'random'`

selection [str, default ‘cyclic’] If set to ‘random’, a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to ‘random’) often leads to significantly faster convergence especially when `tol` is higher than 1e-4.

Attributes

intercept_ [array, shape (n_tasks,)] Independent term in decision function.

coef_ [array, shape (n_tasks, n_features)] Parameter vector (W in the cost function formula). Note that `coef_` stores the transpose of W , $W.T$.

alpha_ [float] The amount of penalization chosen by cross validation

mse_path_ [array, shape (n_alphas, n_folds)] mean square error for the test set on each fold, varying alpha

alphas_ [numpy array, shape (n_alphas,)] The grid of alphas used for fitting.

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance for the optimal alpha.

See also:

[`MultitaskElasticNet`](#)

[`ElasticNetCV`](#)

[`MultitaskElasticNetCV`](#)

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the `X` argument of the `fit` method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn.linear_model import MultiTaskLassoCV
>>> from sklearn.datasets import make_regression
>>> from sklearn.metrics import r2_score
>>> X, y = make_regression(n_targets=2, noise=4, random_state=0)
>>> reg = MultiTaskLassoCV(cv=5, random_state=0).fit(X, y)
>>> r2_score(y, reg.predict(X))
0.9994...
>>> reg.alpha_
0.5713...
>>> reg.predict(X[:1,])
array([[153.7971..., 94.9015...]])
```

Methods

<code>fit(self, X, y)</code>	Fit linear model with coordinate descent
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>path(X, y[, eps, n_alphas, alphas, ...])</code>	Compute Lasso path with coordinate descent
<code>predict(self, X)</code>	Predict using the linear model
<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, eps=0.001, n_alphas=100, alphas=None, fit_intercept=True, normalize=False, max_iter=1000, tol=0.0001, copy_X=True, cv='warn', verbose=False, n_jobs=None, random_state=None, selection='cyclic')`

`fit(self, X, y)`
Fit linear model with coordinate descent

Fit is on grid of alphas and best alpha estimated by cross-validation.

Parameters

`X` [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output, `X` can be sparse.

`y` [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values

`get_params(self, deep=True)`
Get parameters for this estimator.

Parameters

`deep` [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

`params` [mapping of string to any] Parameter names mapped to their values.

`static path(X, y, eps=0.001, n_alphas=100, alphas=None, precompute='auto', Xy=None, copy_X=True, coef_init=None, verbose=False, return_n_iter=False, positive=False, **params)`

Compute Lasso path with coordinate descent

The Lasso optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

$$(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1$$

For multi-output tasks it is:

$$(1 / (2 * n_samples)) * ||Y - XW||^2_{Fro} + alpha * ||W||_21$$

Where:

$$||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}$$

i.e. the sum of norm of each row.

Read more in the [User Guide](#).

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If y is mono-output then X can be sparse.

y [ndarray, shape (n_samples,), or (n_samples, n_outputs)] Target values

eps [float, optional] Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically

precompute [True | False | ‘auto’ | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to ‘auto’ let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when y.ndim == 1).

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha.

See also:

[lars_path](#)
[Lasso](#)
[LassoLars](#)
[LassoCV](#)
[LassoLarsCV](#)
[sklearn.decomposition.sparse_encode](#)

Notes

For an example, see [examples/linear_model/plot_lasso_coordinate_descent_path.py](#).

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Note that in certain cases, the Lars solver may be significantly faster to implement this functionality. In particular, linear interpolation can be used to retrieve model coefficients between the values output by lars_path

Examples

Comparing lasso_path and lars_path with interpolation:

```
>>> X = np.array([[1, 2, 3.1], [2.3, 5.4, 4.3]]).T
>>> y = np.array([1, 2, 3.1])
>>> # Use lasso_path to compute a coefficient path
>>> _, coef_path, _ = lasso_path(X, y, alphas=[5., 1., .5])
>>> print(coef_path)
[[0.          0.          0.46874778]
 [0.2159048  0.4425765  0.23689075]]
```



```
>>> # Now use lars_path and 1D linear interpolation to compute the
>>> # same path
>>> from sklearn.linear_model import lars_path
>>> alphas, active, coef_path_lars = lars_path(X, y, method='lasso')
>>> from scipy import interpolate
>>> coef_path_continuous = interpolate.interp1d(alphas[::-1],
...                                              coef_path_lars[:, ::-1])
>>> print(coef_path_continuous([5., 1., .5]))
[[0.          0.          0.46915237]
 [0.2159048  0.4425765  0.23668876]]
```

`predict(self, X)`

Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (self, X, y, sample_weight=None)

Returns the coefficient of determination R² of the prediction.

The coefficient R² is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R² score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R² of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer 'r2' uses `multioutput='uniform_average'`).

set_params (self, **params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns

self

sklearn.linear_model.OrthogonalMatchingPursuitCV

```
class sklearn.linear_model.OrthogonalMatchingPursuitCV(copy=True, fit_intercept=True,
                                                       normalize=True,
                                                       max_iter=None, cv='warn',
                                                       n_jobs=None, verbose=False)
```

Cross-validated Orthogonal Matching Pursuit model (OMP).

See glossary entry for [cross-validation estimator](#).

Read more in the [User Guide](#).

Parameters

copy [bool, optional] Whether the design matrix X must be copied by the algorithm. A false value is only helpful if X is already Fortran-ordered, otherwise a copy is made anyway.

fit_intercept [boolean, optional] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use `sklearn.preprocessing.StandardScaler` before calling fit on an estimator with `normalize=False`.

max_iter [integer, optional] Maximum numbers of iterations to perform, therefore maximum features to include. 10% of `n_features` but at least 5 if available.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- `CV splitter`,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, `KFold` is used.

Refer [User Guide](#) for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See [Glossary](#) for more details.

verbose [boolean or integer, optional] Sets the verbosity amount

Attributes

intercept_ [float or array, shape (n_targets,)] Independent term in decision function.

coef_ [array, shape (n_features,) or (n_targets, n_features)] Parameter vector (w in the problem formulation).

n_nonzero_coefs_ [int] Estimated number of non-zero coefficients giving the best mean squared error over the cross-validation folds.

n_iter_ [int or array-like] Number of active features across every target for the model refit with the best hyperparameters got by cross-validating across all folds.

See also:

[`orthogonal_mp`](#)

[`orthogonal_mp_gram`](#)

[`lars_path`](#)

[`Lars`](#)

[`LassoLars`](#)

[`OrthogonalMatchingPursuit`](#)

[`LarsCV`](#)

[`LassoLarsCV`](#)

decomposition.sparse_encode**Examples**

```
>>> from sklearn.linear_model import OrthogonalMatchingPursuitCV
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_features=100, n_informative=10,
...                         noise=4, random_state=0)
>>> reg = OrthogonalMatchingPursuitCV(cv=5).fit(X, y)
>>> reg.score(X, y)
0.9991...
>>> reg.n_nonzero_coefs_
10
>>> reg.predict(X[:1,])
array([-78.3854...])
```

Methods

<code>fit(self, X, y)</code>	Fit the model using X, y as training data.
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict using the linear model
<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, copy=True, fit_intercept=True, normalize=True, max_iter=None, cv='warn', n_jobs=None, verbose=False)`

fit (self, X, y)
Fit the model using X, y as training data.

Parameters

X [array-like, shape [n_samples, n_features]] Training data.

y [array-like, shape [n_samples]] Target values. Will be cast to X's dtype if necessary

Returns

self [object] returns an instance of self.

get_params (self, deep=True)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (self, X)
Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (self, X, y, sample_weight=None)

Returns the coefficient of determination R² of the prediction.

The coefficient R² is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R² score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R² of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer '`r2`' uses `multioutput='uniform_average'`).

set_params (self, **params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns

self

Examples using `sklearn.linear_model.OrthogonalMatchingPursuitCV`

- *Orthogonal Matching Pursuit*

`sklearn.linear_model.RidgeCV`

```
class sklearn.linear_model.RidgeCV(alphas=(0.1, 1.0, 10.0), fit_intercept=True, normalize=False, scoring=None, cv=None, gcv_mode=None, store_cv_values=False)
```

Ridge regression with built-in cross-validation.

See glossary entry for [cross-validation estimator](#).

By default, it performs Generalized Cross-Validation, which is a form of efficient Leave-One-Out cross-validation.

Read more in the [User Guide](#).

Parameters

alphas [numpy array of shape [n_alphas]] Array of alpha values to try. Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to C^{-1} in other linear models such as LogisticRegression or LinearSVC. If using generalized cross-validation, alphas must be positive.

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

normalize [boolean, optional, default False] This parameter is ignored when fit_intercept is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use [sklearn.preprocessing.StandardScaler](#) before calling fit on an estimator with normalize=False.

scoring [string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature scorer(estimator, X, y). If None, the negative mean squared error if cv is ‘auto’ or None (i.e. when using generalized cross-validation), and r2 score otherwise.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the efficient Leave-One-Out cross-validation (also known as Generalized Cross-Validation).
- integer, to specify the number of folds.
- [CV splitter](#),
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if y is binary or multiclass, [sklearn.model_selection.StratifiedKFold](#) is used, else, [sklearn.model_selection.KFold](#) is used.

Refer [User Guide](#) for the various cross-validation strategies that can be used here.

gcv_mode [{None, ‘auto’, ‘svd’, ‘eigen’}, optional] Flag indicating which strategy to use when performing Generalized Cross-Validation. Options are:

```
'auto' : use 'svd' if n_samples > n_features, otherwise use 'eigen'
'svd' : force use of singular value decomposition of X when X is
        dense, eigenvalue decomposition of X^T.X when X is sparse.
'eigen' : force computation via eigendecomposition of X.X^T
```

The ‘auto’ mode is the default and is intended to pick the cheaper option of the two depending on the shape of the training data.

store_cv_values [boolean, default=False] Flag indicating if the cross-validation values corresponding to each alpha should be stored in the cv_values_ attribute (see below). This flag is only compatible with cv=None (i.e. using Generalized Cross-Validation).

Attributes

cv_values_ [array, shape = [n_samples, n_alphas] or shape = [n_samples, n_targets, n_alphas], optional] Cross-validation values for each alpha (if `store_cv_values=True` and `cv=None`). After `fit()` has been called, this attribute will contain the mean squared errors (by default) or the values of the `{loss, score}_func` function (if provided in the constructor).

coef_ [array, shape = [n_features] or [n_targets, n_features]] Weight vector(s).

intercept_ [float | array, shape = (n_targets,)] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

alpha_ [float] Estimated regularization parameter.

See also:

[**Ridge**](#) Ridge regression

[**RidgeClassifier**](#) Ridge classifier

[**RidgeClassifierCV**](#) Ridge classifier with built-in cross validation

Examples

```
>>> from sklearn.datasets import load_diabetes
>>> from sklearn.linear_model import RidgeCV
>>> X, y = load_diabetes(return_X_y=True)
>>> clf = RidgeCV(alphas=[1e-3, 1e-2, 1e-1, 1]).fit(X, y)
>>> clf.score(X, y)
0.5166...
```

Methods

<code>fit(self, X, y[, sample_weight])</code>	Fit Ridge regression model
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict using the linear model
<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

```
__init__(self, alphas=(0.1, 1.0, 10.0), fit_intercept=True, normalize=False, scoring=None, cv=None,
         gcv_mode=None, store_cv_values=False)

fit(self, X, y, sample_weight=None)
Fit Ridge regression model
```

Parameters

X [array-like, shape = [n_samples, n_features]] Training data. If using GCV, will be cast to float64 if necessary.

y [array-like, shape = [n_samples] or [n_samples, n_targets]] Target values. Will be cast to X's dtype if necessary

sample_weight [float or array-like of shape [n_samples]] Sample weight

Returns

self [object]

Notes

When `sample_weight` is provided, the selected hyperparameter may depend on whether we use generalized cross-validation (`cv=None` or `cv='auto'`) or another form of cross-validation, because only generalized cross-validation takes the sample weights into account when computing the validation score.

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)

Predict using the linear model

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the coefficient of determination R² of the prediction.

The coefficient R² is defined as $(1 - u/v)$, where u is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y , disregarding the input features, would get a R² score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R² of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer 'r2' uses `multioutput='uniform_average'`).

set_params(self, **params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

Examples using `sklearn.linear_model.RidgeCV`

- *Face completion with a multi-output estimators*
- *Effect of transforming the targets in regression model*

`sklearn.linear_model.RidgeClassifierCV`

```
class sklearn.linear_model.RidgeClassifierCV(alphas=(0.1, 1.0, 10.0), fit_intercept=True,
                                             normalize=False, scoring=None, cv=None,
                                             class_weight=None, store_cv_values=False)
```

Ridge classifier with built-in cross-validation.

See glossary entry for [cross-validation estimator](#).

By default, it performs Generalized Cross-Validation, which is a form of efficient Leave-One-Out cross-validation. Currently, only the `n_features > n_samples` case is handled efficiently.

Read more in the [User Guide](#).

Parameters

alphas [numpy array of shape [n_alphas]] Array of alpha values to try. Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to C^{-1} in other linear models such as LogisticRegression or LinearSVC.

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

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use [sklearn.preprocessing.StandardScaler](#) before calling `fit` on an estimator with `normalize=False`.

scoring [string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the efficient Leave-One-Out cross-validation
- integer, to specify the number of folds.
- [CV splitter](#),
- An iterable yielding (train, test) splits as arrays of indices.

Refer [User Guide](#) for the various cross-validation strategies that can be used here.

class_weight [dict or ‘balanced’, optional] Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one.

The “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as $n_{samples} / (n_{classes} * np.bincount(y))$

store_cv_values [boolean, default=False] Flag indicating if the cross-validation values corresponding to each alpha should be stored in the cv_values_ attribute (see below). This flag is only compatible with cv=None (i.e. using Generalized Cross-Validation).

Attributes

cv_values_ [array, shape = [n_samples, n_targets, n_alphas], optional] Cross-validation values for each alpha (if store_cv_values=True and cv=None). After fit() has been called, this attribute will contain the mean squared errors (by default) or the values of the {loss, score}_func function (if provided in the constructor).

coef_ [array, shape (1, n_features) or (n_targets, n_features)] Coefficient of the features in the decision function.

coef_ is of shape (1, n_features) when the given problem is binary.

intercept_ [float | array, shape = (n_targets,)] Independent term in decision function. Set to 0.0 if fit_intercept = False.

alpha_ [float] Estimated regularization parameter

See also:

[**Ridge**](#) Ridge regression

[**RidgeClassifier**](#) Ridge classifier

[**RidgeCV**](#) Ridge regression with built-in cross validation

Notes

For multi-class classification, n_class classifiers are trained in a one-versus-all approach. Concretely, this is implemented by taking advantage of the multi-variate response support in Ridge.

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.linear_model import RidgeClassifierCV
>>> X, y = load_breast_cancer(return_X_y=True)
>>> clf = RidgeClassifierCV(alphas=[1e-3, 1e-2, 1e-1, 1]).fit(X, y)
>>> clf.score(X, y)
0.9630...
```

Methods

<code>decision_function(self, X)</code>	Predict confidence scores for samples.
<code>fit(self, X, y[, sample_weight])</code>	Fit the ridge classifier.
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict class labels for samples in X.
<code>score(self, X, y[, sample_weight])</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, alphas=(0.1, 1.0, 10.0), fit_intercept=True, normalize=False, scoring=None, cv=None, class_weight=None, store_cv_values=False)`

`decision_function(self, X)`

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

`X` [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

`array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes)` Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

`fit(self, X, y, sample_weight=None)`

Fit the ridge classifier.

Parameters

`X` [array-like, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features. When using GCV, will be cast to float64 if necessary.

`y` [array-like, shape (n_samples,)] Target values. Will be cast to X's dtype if necessary

`sample_weight` [float or numpy array of shape (n_samples,)] Sample weight.

Returns

`self` [object]

`get_params(self, deep=True)`

Get parameters for this estimator.

Parameters

`deep` [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

`params` [mapping of string to any] Parameter names mapped to their values.

`predict(self, X)`

Predict class labels for samples in X.

Parameters

`X` [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns

self

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 benefiting from the Akaike 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

`sklearn.linear_model.LassoLarsIC`

```
class sklearn.linear_model.LassoLarsIC(criterion='aic', fit_intercept=True, verbose=False,
                                         normalize=True, precompute='auto', max_iter=500,
                                         eps=2.220446049250313e-16, copy_X=True, positive=False)
```

Lasso model fit with Lars using BIC or AIC for model selection

The optimization objective for Lasso is:

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

AIC is the Akaike information criterion and BIC is the Bayes Information criterion. Such criteria are useful to select the value of the regularization parameter by making a trade-off between the goodness of fit and the complexity of the model. A good model should explain well the data while being simple.

Read more in the [User Guide](#).

Parameters

criterion [‘bic’ | ‘aic’] The type of criterion to use.

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

verbose [boolean or integer, optional] Sets the verbosity amount

normalize [boolean, optional, default True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use `sklearn.preprocessing.StandardScaler` before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | ‘auto’ | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to ‘auto’ let us decide. The Gram matrix can also be passed as argument.

max_iter [integer, optional] Maximum number of iterations to perform. Can be used for early stopping.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the `tol` parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

positive [boolean (default=False)] Restrict coefficients to be ≥ 0 . Be aware that you might want to remove `fit_intercept` which is set True by default. Under the positive restriction the model coefficients do not converge to the ordinary-least-squares solution for small values of alpha. Only coefficients up to the smallest alpha value (`alphas_[alphas_ > 0.] . min()` when `fit_path=True`) reached by the stepwise Lars-Lasso algorithm are typically in congruence with the solution of the coordinate descent Lasso estimator. As a consequence using `LassoLarsIC` only makes sense for problems where a sparse solution is expected and/or reached.

Attributes

coef_ [array, shape (n_features,)] parameter vector (w in the formulation formula)

intercept_ [float] independent term in decision function.

alpha_ [float] the alpha parameter chosen by the information criterion

n_iter_ [int] number of iterations run by `lars_path` to find the grid of alphas.

criterion_ [array, shape (n_alphas,)] The value of the information criteria (‘aic’, ‘bic’) across all alphas. The alpha which has the smallest information criterion is chosen. This value is larger by a factor of `n_samples` compared to Eqns. 2.15 and 2.16 in (Zou et al, 2007).

See also:

`lars_path`, `LassoLars`, `LassoLarsCV`

Notes

The estimation of the number of degrees of freedom is given by:

“On the degrees of freedom of the lasso” Hui Zou, Trevor Hastie, and Robert Tibshirani Ann. Statist. Volume 35, Number 5 (2007), 2173–2192.

https://en.wikipedia.org/wiki/Akaike_information_criterion

https://en.wikipedia.org/wiki/Bayesian_information_criterion

Examples

```
>>> from sklearn import linear_model
>>> reg = linear_model.LassoLarsIC(criterion='bic')
>>> reg.fit([-1, 1], [0, 0], [1, 1], [-1.1111, 0, -1.1111])
...
LassoLarsIC(copy_X=True, criterion='bic', eps=..., fit_intercept=True,
            max_iter=500, normalize=True, positive=False, precompute='auto',
            verbose=False)
>>> print(reg.coef_)
[ 0. -1.11...]
```

Methods

<code>fit(self, X, y[, copy_X])</code>	Fit the model using X, y as training data.
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict using the linear model
<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, criterion='aic', fit_intercept=True, verbose=False, normalize=True, precompute='auto', max_iter=500, eps=2.220446049250313e-16, copy_X=True, positive=False)`

`fit(self, X, y, copy_X=None)`
Fit the model using X, y as training data.

Parameters

`X` [array-like, shape (n_samples, n_features)] training data.

`y` [array-like, shape (n_samples,)] target values. Will be cast to X’s dtype if necessary

`copy_X` [boolean, optional, default None] If provided, this parameter will override the choice of copy_X made at instance creation. If True, X will be copied; else, it may be overwritten.

Returns

`self` [object] returns an instance of self.

`get_params(self, deep=True)`
Get parameters for this estimator.

Parameters

`deep` [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)

Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the coefficient of determination R² of the prediction.

The coefficient R² is defined as $(1 - u/v)$, where *u* is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and *v* is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R² score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R² of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer 'r2' uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns

self

Examples using `sklearn.linear_model.LassoLarsIC`

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

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:

<code>ensemble.RandomForestClassifier([...])</code>	A random forest classifier.
<code>ensemble.RandomForestRegressor([...])</code>	A random forest regressor.
<code>ensemble.ExtraTreesClassifier([...])</code>	An extra-trees classifier.
<code>ensemble.ExtraTreesRegressor([n_estimators,...])</code>	An extra-trees regressor.
<code>ensemble.GradientBoostingClassifier([loss,...])</code>	Gradient Boosting for classification.
<code>ensemble.GradientBoostingRegressor([loss,...])</code>	Gradient Boosting for regression.

`sklearn.ensemble.RandomForestClassifier`

```
class sklearn.ensemble.RandomForestClassifier(n_estimators='warn', criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None)
```

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if `bootstrap=True` (default).

Read more in the [User Guide](#).

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

Changed in version 0.20: The default value of `n_estimators` will change from 10 in version 0.20 to 100 in version 0.22.

criterion [string, optional (default=”gini”)] The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. Note: this parameter is tree-specific.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than

min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider min_samples_split as the minimum number.
- If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider min_samples_leaf as the minimum number.
- If float, then min_samples_leaf is a fraction and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- If int, then consider max_features features at each split.
- If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
- If "auto", then max_features=sqrt(n_features).
- If "sqrt", then max_features=sqrt(n_features) (same as "auto").
- If "log2", then max_features=log2(n_features).
- If None, then max_features=n_features.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

$$\begin{aligned} & N_t / N * (\text{impurity} - N_{t_R} / N_t * \text{right_impurity} \\ & - N_{t_L} / N_t * \text{left_impurity}) \end{aligned}$$

where N is the total number of samples, N_t is the number of samples at the current node, N_t_L is the number of samples in the left child, and N_t_R is the number of samples in the right child.

N, N_t, N_t_R and N_t_L all refer to the weighted sum, if sample_weight is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

bootstrap [boolean, optional (default=True)] Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score [bool (default=False)] Whether to use out-of-bag samples to estimate the generalization accuracy.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both `fit` and `predict`. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See [Glossary](#) for more details.

random_state [int, RandomState instance or None, optional (default=None)] If int, `random_state` is the seed used by the random number generator; If RandomState instance, `random_state` is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

warm_start [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See [the Glossary](#).

class_weight [dict, list of dicts, “balanced”, “balanced_subsample” or None, optional (default=None)] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of `y`.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be `[{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}]` instead of `[{1:1}, {2:5}, {3:1}, {4:1}]`.

The “balanced” mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

The “balanced_subsample” mode is the same as “balanced” except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of `y` will be multiplied.

Note that these weights will be multiplied with `sample_weight` (passed through the `fit` method) if `sample_weight` is specified.

Attributes

estimators_ [list of DecisionTreeClassifier] The collection of fitted sub-estimators.

classes_ [array of shape = [n_classes] or a list of such arrays] The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

n_classes_ [int or list] The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

feature_importances_ [array of shape = [n_features]] Return the feature importances (the higher, the more important the feature).

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_decision_function_ [array of shape = [n_samples, n_classes]] Decision function computed with out-of-bag estimate on the training set. If `n_estimators` is small it might be possible that a data point was never left out during the bootstrap. In this case, `oob_decision_function_` might contain NaN.

See also:

`DecisionTreeClassifier`, `ExtraTreesClassifier`

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, `max_features=n_features` and `bootstrap=False`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

References

[R45f14345c000-1]

Examples

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_classification

>>> X, y = make_classification(n_samples=1000, n_features=4,
...                             n_informative=2, n_redundant=0,
...                             random_state=0, shuffle=False)
>>> clf = RandomForestClassifier(n_estimators=100, max_depth=2,
...                               random_state=0)
>>> clf.fit(X, y)
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                      max_depth=2, max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
                      oob_score=False, random_state=0, verbose=0, warm_start=False)
>>> print(clf.feature_importances_)
[0.14205973 0.76664038 0.0282433 0.06305659]
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

Methods

<code>apply(self, X)</code>	Apply trees in the forest to X, return leaf indices.
<code>decision_path(self, X)</code>	Return the decision path in the forest
<code>fit(self, X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict class for X.
<code>predict_log_proba(self, X)</code>	Predict class log-probabilities for X.
<code>predict_proba(self, X)</code>	Predict class probabilities for X.
<code>score(self, X, y[, sample_weight])</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, n_estimators='warn', criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None)`

apply (self, X)

Apply trees in the forest to X, return leaf indices.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

X_leaves [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

decision_path (self, X)

Return the decision path in the forest

New in version 0.18.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit (*self*, *X*, *y*, *sample_weight=None*)
Build a forest of trees from the training set (*X*, *y*).

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)
Predict class for *X*.

The predicted class of an input sample is a vote by the trees in the forest, weighted by their probability estimates. That is, the predicted class is the one with highest mean probability estimate across the trees.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

y [array of shape = [n_samples] or [n_samples, n_outputs]] The predicted classes.

predict_log_proba (*self*, *X*)
Predict class log-probabilities for *X*.

The predicted class log-probabilities of an input sample is computed as the log of the mean predicted class probabilities of the trees in the forest.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

p [array of shape = [n_samples, n_classes], or a list of n_outputs] such arrays if n_outputs > 1. The class probabilities of the input samples. The order of the classes corresponds to that in the attribute `classes_`.

`predict_proba(self, X)`

Predict class probabilities for X.

The predicted class probabilities of an input sample are computed as the mean predicted class probabilities of the trees in the forest. The class probability of a single tree is the fraction of samples of the same class in a leaf.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

p [array of shape = [n_samples, n_classes], or a list of n_outputs] such arrays if n_outputs > 1. The class probabilities of the input samples. The order of the classes corresponds to that in the attribute `classes_`.

`score(self, X, y, sample_weight=None)`

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

`set_params(self, **params)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

Examples using `sklearn.ensemble.RandomForestClassifier`

- [Comparison of Calibration of Classifiers](#)
- [Probability Calibration for 3-class classification](#)

- [Classifier comparison](#)
- [Inductive Clustering](#)
- [Plot class probabilities calculated by the VotingClassifier](#)
- [OOB Errors for Random Forests](#)
- [Feature transformations with ensembles of trees](#)
- [Plot the decision surfaces of ensembles of trees on the iris dataset](#)
- [Comparing randomized search and grid search for hyperparameter estimation](#)
- [Classification of text documents using sparse features](#)

`sklearn.ensemble.RandomForestRegressor`

```
class sklearn.ensemble.RandomForestRegressor(n_estimators='warn',  
                                             criterion='mse', max_depth=None,  
                                             min_samples_split=2, min_samples_leaf=1,  
                                             min_weight_fraction_leaf=0.0,  
                                             max_features='auto', max_leaf_nodes=None,  
                                             min_impurity_decrease=0.0,  
                                             min_impurity_split=None, bootstrap=True,  
                                             oob_score=False, n_jobs=None,  
                                             random_state=None, verbose=0,  
                                             warm_start=False)
```

A random forest regressor.

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if `bootstrap=True` (default).

Read more in the [User Guide](#).

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

Changed in version 0.20: The default value of `n_estimators` will change from 10 in version 0.20 to 100 in version 0.22.

criterion [string, optional (default="mse")] The function to measure the quality of a split. Supported criteria are “mse” for the mean squared error, which is equal to variance reduction as feature selection criterion, and “mae” for the mean absolute error.

New in version 0.18: Mean Absolute Error (MAE) criterion.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than `min_samples_split` samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when `sample_weight` is not provided.

max_features [int, float, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

$$\frac{N_t}{N} \cdot (\text{impurity} - \frac{N_{t_R}}{N_t} \cdot \text{right_impurity} - \frac{N_{t_L}}{N_t} \cdot \text{left_impurity})$$

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from `1e-7` to `0` in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

bootstrap [boolean, optional (default=True)] Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score [bool, optional (default=False)] whether to use out-of-bag samples to estimate the R² on unseen data.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both `fit` and `predict`. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See [Glossary](#) for more details.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

warm_start [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See [the Glossary](#).

Attributes

estimators_ [list of DecisionTreeRegressor] The collection of fitted sub-estimators.

feature_importances_ [array of shape = [n_features]] Return the feature importances (the higher, the more important the feature).

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_prediction_ [array of shape = [n_samples]] Prediction computed with out-of-bag estimate on the training set.

See also:

`DecisionTreeRegressor`, `ExtraTreesRegressor`

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, `max_features=n_features` and `bootstrap=False`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

The default value `max_features="auto"` uses `n_features` rather than `n_features / 3`. The latter was originally suggested in [1], whereas the former was more recently justified empirically in [2].

References

[Rf91cab2dc427-1], [Rf91cab2dc427-2]

Examples

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.datasets import make_regression

>>> X, y = make_regression(n_features=4, n_informative=2,
...                         random_state=0, shuffle=False)
>>> regr = RandomForestRegressor(max_depth=2, random_state=0,
...                               n_estimators=100)
...
>>> regr.fit(X, y)
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=2,
                      max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
                      oob_score=False, random_state=0, verbose=0, warm_start=False)
>>> print(regr.feature_importances_)
[0.18146984 0.81473937 0.00145312 0.00233767]
>>> print(regr.predict([[0, 0, 0, 0]]))
[-8.32987858]
```

Methods

<code>apply(self, X)</code>	Apply trees in the forest to X, return leaf indices.
<code>decision_path(self, X)</code>	Return the decision path in the forest
<code>fit(self, X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict regression target for X.
<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, n_estimators='warn', criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False)`

`apply(self, X)`

Apply trees in the forest to X, return leaf indices.

Parameters

`X` [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

Returns

`X_leaves` [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

`decision_path(self, X)`

Return the decision path in the forest

New in version 0.18.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from `indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]]` gives the indicator value for the i-th estimator.

feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit (self, X, y, sample_weight=None)

Build a forest of trees from the training set (X, y).

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

self [object]

get_params (self, deep=True)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (self, X)

Predict regression target for X.

The predicted regression target of an input sample is computed as the mean predicted regression targets of the trees in the forest.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

y [array of shape = [n_samples] or [n_samples, n_outputs]] The predicted values.

score (*self*, X, y, *sample_weight=None*)

Returns the coefficient of determination R² of the prediction.

The coefficient R² is defined as $(1 - u/v)$, where u is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R² score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R² of *self.predict(X)* wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer 'r2' uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

Examples using `sklearn.ensemble.RandomForestRegressor`

- *Prediction Latency*
- *Plot individual and voting regression predictions*

- *Comparing random forests and the multi-output meta estimator*
- *Imputing missing values before building an estimator*

sklearn.ensemble.ExtraTreesClassifier

```
class sklearn.ensemble.ExtraTreesClassifier(n_estimators='warn', crite-  
rion='gini', max_depth=None,  
min_samples_split=2, min_samples_leaf=1,  
min_weight_fraction_leaf=0.0,  
max_features='auto', max_leaf_nodes=None,  
min_impurity_decrease=0.0,  
min_impurity_split=None, bootstrap=False,  
oob_score=False, n_jobs=None,  
random_state=None, verbose=0,  
warm_start=False, class_weight=None)
```

An extra-trees classifier.

This class implements a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

Read more in the [User Guide](#).

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

Changed in version 0.20: The default value of n_estimators will change from 10 in version 0.20 to 100 in version 0.22.

criterion [string, optional (default="gini")] The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider min_samples_split as the minimum number.
- If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider min_samples_leaf as the minimum number.
- If float, then min_samples_leaf is a fraction and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, string or None, optional (default=’auto’)] The number of features to consider when looking for the best split:

- If int, then consider max_features features at each split.
- If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
- If “auto”, then max_features=sqrt (n_features).
- If “sqrt”, then max_features=sqrt (n_features).
- If “log2”, then max_features=log2 (n_features).
- If None, then max_features=n_features.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

$$\frac{N_t}{N} \cdot (\text{impurity} - \frac{N_{t_R}}{N_t} \cdot \text{right_impurity} - \frac{N_{t_L}}{N_t} \cdot \text{left_impurity})$$

where N is the total number of samples, N_t is the number of samples at the current node, N_{t_L} is the number of samples in the left child, and N_{t_R} is the number of samples in the right child.

N , N_t , N_{t_R} and N_{t_L} all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

bootstrap [boolean, optional (default=False)] Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score [bool, optional (default=False)] Whether to use out-of-bag samples to estimate the generalization accuracy.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both `fit` and `predict`. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See [Glossary](#) for more details.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See [the Glossary](#).

class_weight [dict, list of dicts, “balanced”, “balanced_subsample” or None, optional (default=None)] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of `y`.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be `[{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}]` instead of `[{1:1}, {2:5}, {3:1}, {4:1}]`.

The “balanced” mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

The “balanced_subsample” mode is the same as “balanced” except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of `y` will be multiplied.

Note that these weights will be multiplied with `sample_weight` (passed through the `fit` method) if `sample_weight` is specified.

Attributes

estimators_ [list of DecisionTreeClassifier] The collection of fitted sub-estimators.

classes_ [array of shape = `[n_classes]` or a list of such arrays] The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

n_classes_ [int or list] The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).

feature_importances_ [array of shape = `[n_features]`] Return the feature importances (the higher, the more important the feature).

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_decision_function_ [array of shape = `[n_samples, n_classes]`] Decision function computed with out-of-bag estimate on the training set. If `n_estimators` is small it might be possible that a data point was never left out during the bootstrap. In this case, `oob_decision_function_` might contain NaN.

See also:

`sklearn.tree.ExtraTreeClassifier` Base classifier for this ensemble.

`RandomForestClassifier` Ensemble Classifier based on trees with optimal splits.

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on

some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

References

[Rc8f28bfad63f-1]

Methods

<code>apply(self, X)</code>	Apply trees in the forest to X, return leaf indices.
<code>decision_path(self, X)</code>	Return the decision path in the forest
<code>fit(self, X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict class for X.
<code>predict_log_proba(self, X)</code>	Predict class log-probabilities for X.
<code>predict_proba(self, X)</code>	Predict class probabilities for X.
<code>score(self, X, y[, sample_weight])</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

```
__init__(self, n_estimators='warn', criterion='gini', max_depth=None, min_samples_split=2,
        min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto',
        max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=False,
        oob_score=False, n_jobs=None, random_state=None, verbose=0,
        warm_start=False, class_weight=None)
```

apply (self, X)

Apply trees in the forest to X, return leaf indices.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

X_leaves [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

decision_path (self, X)

Return the decision path in the forest

New in version 0.18.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit (self, X, y, sample_weight=None)

Build a forest of trees from the training set (X, y).

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

self [object]

get_params (self, deep=True)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (self, X)

Predict class for X.

The predicted class of an input sample is a vote by the trees in the forest, weighted by their probability estimates. That is, the predicted class is the one with highest mean probability estimate across the trees.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

Returns

y [array of shape = [n_samples] or [n_samples, n_outputs]] The predicted classes.

`predict_log_proba(self, X)`

Predict class log-probabilities for X.

The predicted class log-probabilities of an input sample is computed as the log of the mean predicted class probabilities of the trees in the forest.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

p [array of shape = [n_samples, n_classes], or a list of n_outputs] such arrays if n_outputs > 1. The class probabilities of the input samples. The order of the classes corresponds to that in the attribute `classes_`.

`predict_proba(self, X)`

Predict class probabilities for X.

The predicted class probabilities of an input sample are computed as the mean predicted class probabilities of the trees in the forest. The class probability of a single tree is the fraction of samples of the same class in a leaf.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

p [array of shape = [n_samples, n_classes], or a list of n_outputs] such arrays if n_outputs > 1. The class probabilities of the input samples. The order of the classes corresponds to that in the attribute `classes_`.

`score(self, X, y, sample_weight=None)`

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

`set_params(self, **params)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

Examples using `sklearn.ensemble.ExtraTreesClassifier`

- *Pixel importances with a parallel forest of trees*
- *Feature importances with forests of trees*
- *Hashing feature transformation using Totally Random Trees*
- *Plot the decision surfaces of ensembles of trees on the iris dataset*

sklearn.ensemble.ExtraTreesRegressor

```
class sklearn.ensemble.ExtraTreesRegressor(n_estimators='warn',  
                                         criterion='mse',  
                                         max_depth=None,  
                                         min_samples_split=2, min_samples_leaf=1,  
                                         min_weight_fraction_leaf=0.0,  
                                         max_features='auto', max_leaf_nodes=None,  
                                         min_impurity_decrease=0.0,  
                                         min_impurity_split=None, bootstrap=False,  
                                         oob_score=False, n_jobs=None,  
                                         random_state=None, verbose=0,  
                                         warm_start=False)
```

An extra-trees regressor.

This class implements a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

Read more in the [User Guide](#).

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

Changed in version 0.20: The default value of `n_estimators` will change from 10 in version 0.20 to 100 in version 0.22.

criterion [string, optional (default="mse")] The function to measure the quality of a split. Supported criteria are "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion, and "mae" for the mean absolute error.

New in version 0.18: Mean Absolute Error (MAE) criterion.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than `min_samples_split` samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and $\text{ceil}(\text{min_samples_split} * \text{n_samples})$ are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when `sample_weight` is not provided.

max_features [int, float, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

$$\frac{N_t}{N} \cdot (\text{impurity} - \frac{N_{t_R}}{N_t} \cdot \text{right_impurity} - \frac{N_{t_L}}{N_t} \cdot \text{left_impurity})$$

where N is the total number of samples, N_t is the number of samples at the current node, N_{t_L} is the number of samples in the left child, and N_{t_R} is the number of samples in the right child.

N , N_t , N_{t_R} and N_{t_L} all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

bootstrap [boolean, optional (default=False)] Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score [bool, optional (default=False)] Whether to use out-of-bag samples to estimate the R^2 on unseen data.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both `fit` and `predict`. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See [Glossary](#) for more details.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

warm_start [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See [the Glossary](#).

Attributes

estimators_ [list of DecisionTreeRegressor] The collection of fitted sub-estimators.

feature_importances_ [array of shape = [n_features]] Return the feature importances (the higher, the more important the feature).

n_features_ [int] The number of features.

n_outputs_ [int] The number of outputs.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_prediction_ [array of shape = [n_samples]] Prediction computed with out-of-bag estimate on the training set.

See also:

[`sklearn.tree.ExtraTreeRegressor`](#) Base estimator for this ensemble.

[`RandomForestRegressor`](#) Ensemble regressor using trees with optimal splits.

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

References

[Ra7d0c8995fbc-1]

Methods

<code>apply(self, X)</code>	Apply trees in the forest to X, return leaf indices.
<code>decision_path(self, X)</code>	Return the decision path in the forest
<code>fit(self, X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict regression target for X.

Continued on next page

Table 3.18 – continued from previous page

<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.

`__init__(self, n_estimators='warn', criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=False, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False)`

`apply(self, X)`

Apply trees in the forest to X, return leaf indices.

Parameters

`X` [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

Returns

`X_leaves` [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

`decision_path(self, X)`

Return the decision path in the forest

New in version 0.18.

Parameters

`X` [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

Returns

`indicator` [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

`n_nodes_ptr` [array of size (n_estimators + 1,)] The columns from `indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]]` gives the indicator value for the i-th estimator.

`feature_importances_`

Return the feature importances (the higher, the more important the feature).

Returns

`feature_importances_` [array, shape = [n_features]] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

`fit(self, X, y, sample_weight=None)`

Build a forest of trees from the training set (X, y).

Parameters

`X` [array-like or sparse matrix of shape = [n_samples, n_features]] The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

self [object]

get_params (self, deep=True)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (self, X)

Predict regression target for X.

The predicted regression target of an input sample is computed as the mean predicted regression targets of the trees in the forest.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

y [array of shape = [n_samples] or [n_samples, n_outputs]] The predicted values.

score (self, X, y, sample_weight=None)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as $(1 - u/v)$, where u is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer 'r2' uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

Examples using `sklearn.ensemble.ExtraTreesRegressor`

- *Face completion with a multi-output estimators*
- *Imputing missing values with variants of IterativeImputer*

sklearn.ensemble.GradientBoostingClassifier

```
class sklearn.ensemble.GradientBoostingClassifier(loss='deviance', learning_rate=0.1,
                                                 n_estimators=100, subsample=1.0,
                                                 criterion='friedman_mse',
                                                 min_samples_split=2,
                                                 min_samples_leaf=1,
                                                 min_weight_fraction_leaf=0.0,
                                                 max_depth=3,
                                                 min_impurity_decrease=0.0,
                                                 min_impurity_split=None,
                                                 init=None, random_state=None,
                                                 max_features=None, verbose=0,
                                                 max_leaf_nodes=None,
                                                 warm_start=False, presort='auto',
                                                 validation_fraction=0.1,
                                                 n_iter_no_change=None, tol=0.0001)
```

Gradient Boosting for classification.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes` regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

Read more in the *User Guide*.

Parameters

loss [{‘deviance’, ‘exponential’}, optional (default=’deviance’)] loss function to be optimized. ‘deviance’ refers to deviance (= logistic regression) for classification with probabilistic outputs. For loss ‘exponential’ gradient boosting recovers the AdaBoost algorithm.

learning_rate [float, optional (default=0.1)] learning rate shrinks the contribution of each tree by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

n_estimators [int (default=100)] The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

subsample [float, optional (default=1.0)] The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. `subsample` interacts with the parameter `n_estimators`. Choosing `subsample` < 1.0 leads to a reduction of variance and an increase in bias.

criterion [string, optional (default=”friedman_mse”)] The function to measure the quality of a split. Supported criteria are “friedman_mse” for the mean squared error with improvement score by Friedman, “mse” for mean squared error, and “mae” for the mean absolute error. The default value of “friedman_mse” is generally the best as it can provide a better approximation in some cases.

New in version 0.18.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when `sample_weight` is not provided.

max_depth [integer, optional (default=3)] maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

$$\frac{N_t}{N} \cdot (\text{impurity} - \frac{N_{t,R}}{N_t} \cdot \text{right_impurity} - \frac{N_{t,L}}{N_t} \cdot \text{left_impurity})$$

where N is the total number of samples, N_t is the number of samples at the current node, N_{t_L} is the number of samples in the left child, and N_{t_R} is the number of samples in the right child.

N , N_t , N_{t_R} and N_{t_L} all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

init [estimator or ‘zero’, optional (default=None)] An estimator object that is used to compute the initial predictions. `init` has to provide `fit` and `predict_proba`. If ‘zero’, the initial raw predictions are set to zero. By default, a `DummyEstimator` predicting the classes priors is used.

random_state [int, RandomState instance or None, optional (default=None)] If int, `random_state` is the seed used by the random number generator; If RandomState instance, `random_state` is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

max_features [int, float, string or None, optional (default=None)] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If “auto”, then `max_features=sqrt(n_features)`.
- If “sqrt”, then `max_features=sqrt(n_features)`.
- If “log2”, then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Choosing `max_features < n_features` leads to a reduction of variance and an increase in bias.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

verbose [int, default: 0] Enable verbose output. If 1 then it prints progress and performance once in a while (the more trees the lower the frequency). If greater than 1 then it prints progress and performance for every tree.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

warm_start [bool, default: False] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just erase the previous solution. See [the Glossary](#).

presort [bool or ‘auto’, optional (default='auto')] Whether to presort the data to speed up the finding of best splits in fitting. Auto mode by default will use presorting on dense data and

default to normal sorting on sparse data. Setting presort to true on sparse data will raise an error.

New in version 0.17: *presort* parameter.

validation_fraction [float, optional, default 0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if *n_iter_no_change* is set to an integer.

New in version 0.20.

n_iter_no_change [int, default None] *n_iter_no_change* is used to decide if early stopping will be used to terminate training when validation score is not improving. By default it is set to None to disable early stopping. If set to a number, it will set aside *validation_fraction* size of the training data as validation and terminate training when validation score is not improving in all of the previous *n_iter_no_change* numbers of iterations. The split is stratified.

New in version 0.20.

tol [float, optional, default 1e-4] Tolerance for the early stopping. When the loss is not improving by at least tol for *n_iter_no_change* iterations (if set to a number), the training stops.

New in version 0.20.

Attributes

n_estimators_ [int] The number of estimators as selected by early stopping (if *n_iter_no_change* is specified). Otherwise it is set to *n_estimators*.

New in version 0.20.

feature_importances_ [array, shape (n_features,)] Return the feature importances (the higher, the more important the feature).

oob_improvement_ [array, shape (n_estimators,)] The improvement in loss (= deviance) on the out-of-bag samples relative to the previous iteration. *oob_improvement_[0]* is the improvement in loss of the first stage over the *init* estimator.

train_score_ [array, shape (n_estimators,)] The i-th score *train_score_[i]* is the deviance (= loss) of the model at iteration *i* on the in-bag sample. If *subsample == 1* this is the deviance on the training data.

loss_ [LossFunction] The concrete *LossFunction* object.

init_ [estimator] The estimator that provides the initial predictions. Set via the *init* argument or *loss.init_estimator*.

estimators_ [ndarray of DecisionTreeRegressor, shape (n_estimators, loss_.K)] The collection of fitted sub-estimators. *loss_.K* is 1 for binary classification, otherwise n_classes.

See also:

[*sklearn.ensemble.HistGradientBoostingClassifier*](#)

[*sklearn.tree.DecisionTreeClassifier, RandomForestClassifier*](#)

[*AdaBoostClassifier*](#)

Notes

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data and `max_features=n_features`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

References

- J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, *The Annals of Statistics*, Vol. 29, No. 5, 2001.
10. Friedman, Stochastic Gradient Boosting, 1999
- T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning Ed. 2, Springer, 2009.

Methods

<code>apply(self, X)</code>	Apply trees in the ensemble to X, return leaf indices.
<code>decision_function(self, X)</code>	Compute the decision function of X.
<code>fit(self, X, y[, sample_weight, monitor])</code>	Fit the gradient boosting model.
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict class for X.
<code>predict_log_proba(self, X)</code>	Predict class log-probabilities for X.
<code>predict_proba(self, X)</code>	Predict class probabilities for X.
<code>score(self, X, y[, sample_weight])</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.
<code>staged_decision_function(self, X)</code>	Compute decision function of X for each iteration.
<code>staged_predict(self, X)</code>	Predict class at each stage for X.
<code>staged_predict_proba(self, X)</code>	Predict class probabilities at each stage for X.

```
__init__(self, loss='deviance', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, min_impurity_split=None, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, presort='auto', validation_fraction=0.1, n_iter_no_change=None, tol=0.0001)
```

`apply(self, X)`
Apply trees in the ensemble to X, return leaf indices.

New in version 0.17.

Parameters

`X` [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, its `dtype` will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted to a sparse `csc_matrix`.

Returns

`X_leaves` [array-like, shape (n_samples, n_estimators, n_classes)] For each datapoint x in X and for each tree in the ensemble, return the index of the leaf x ends up in each estimator.

In the case of binary classification n_classes is 1.

decision_function(*self*, *X*)

Compute the decision function of *X*.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

score [array, shape (n_samples, n_classes) or (n_samples,)] The decision function of the input samples, which corresponds to the raw values predicted from the trees of the ensemble. The order of the classes corresponds to that in the attribute `classes_`. Regression and binary classification produce an array of shape [n_samples].

feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape (n_features,)] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit(*self*, *X*, *y*, *sample_weight=None*, *monitor=None*)

Fit the gradient boosting model.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

y [array-like, shape (n_samples,)] Target values (strings or integers in classification, real numbers in regression) For classification, labels must correspond to classes.

sample_weight [array-like, shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

monitor [callable, optional] The monitor is called after each iteration with the current iteration, a reference to the estimator and the local variables of `_fit_stages` as keyword arguments `callable(i, self, locals())`. If the callable returns `True` the fitting procedure is stopped. The monitor can be used for various things such as computing held-out estimates, early stopping, model introspect, and snapshoting.

Returns

self [object]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*self*, *X*)

Predict class for *X*.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

y [array, shape (n_samples,)] The predicted values.

predict_log_proba(*self*, *X*)

Predict class log-probabilities for *X*.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

p [array, shape (n_samples, n_classes)] The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute `classes_`.

Raises

AttributeError If the `loss` does not support probabilities.

predict_proba(*self*, *X*)

Predict class probabilities for *X*.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

p [array, shape (n_samples, n_classes)] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute `classes_`.

Raises

AttributeError If the `loss` does not support probabilities.

score(*self*, *X*, *y*, *sample_weight=None*)

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (self, **params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns

self

staged_decision_function (self, X)

Compute decision function of X for each iteration.

This method allows monitoring (i.e. determine error on testing set) after each stage.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

score [generator of array, shape (n_samples, k)] The decision function of the input samples, which corresponds to the raw values predicted from the trees of the ensemble . The classes corresponds to that in the attribute `classes_`. Regression and binary classification are special cases with k == 1, otherwise k==n_classes.

staged_predict (self, X)

Predict class at each stage for X.

This method allows monitoring (i.e. determine error on testing set) after each stage.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

y [generator of array of shape (n_samples,)] The predicted value of the input samples.

staged_predict_proba (self, X)

Predict class probabilities at each stage for X.

This method allows monitoring (i.e. determine error on testing set) after each stage.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

y [generator of array of shape (n_samples,)] The predicted value of the input samples.

Examples using `sklearn.ensemble.GradientBoostingClassifier`

- Gradient Boosting regularization
- Early stopping of Gradient Boosting
- Feature transformations with ensembles of trees
- Gradient Boosting Out-of-Bag estimates
- Feature discretization

`sklearn.ensemble.GradientBoostingRegressor`

```
class sklearn.ensemble.GradientBoostingRegressor(loss='ls',           learning_rate=0.1,
                                                n_estimators=100,         subsample=1.0,
                                                criterion='friedman_mse',
                                                min_samples_split=2,
                                                min_samples_leaf=1,
                                                min_weight_fraction_leaf=0.0,
                                                max_depth=3,
                                                min_impurity_decrease=0.0,
                                                min_impurity_split=None,
                                                init=None,               random_state=None,
                                                max_features=None,       alpha=0.9,
                                                verbose=0,               max_leaf_nodes=None,
                                                warm_start=False,         pre-
                                                sort='auto',              validation_fraction=0.1,
                                                n_iter_no_change=None,    tol=0.0001)
```

Gradient Boosting for regression.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage a regression tree is fit on the negative gradient of the given loss function.

Read more in the [User Guide](#).

Parameters

loss [{‘ls’, ‘lad’, ‘huber’, ‘quantile’}], optional (default=’ls’)] loss function to be optimized. ‘ls’ refers to least squares regression. ‘lad’ (least absolute deviation) is a highly robust loss function solely based on order information of the input variables. ‘huber’ is a combination of the two. ‘quantile’ allows quantile regression (use `alpha` to specify the quantile).

learning_rate [float, optional (default=0.1)] learning rate shrinks the contribution of each tree by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

n_estimators [int (default=100)] The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

subsample [float, optional (default=1.0)] The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. `subsample` interacts with the parameter `n_estimators`. Choosing `subsample` < 1.0 leads to a reduction of variance and an increase in bias.

criterion [string, optional (default=”friedman_mse”)] The function to measure the quality of a split. Supported criteria are “friedman_mse” for the mean squared error with improvement score by Friedman, “mse” for mean squared error, and “mae” for the mean absolute error. The default value of “friedman_mse” is generally the best as it can provide a better approximation in some cases.

New in version 0.18.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when `sample_weight` is not provided.

max_depth [integer, optional (default=3)] maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

$$\frac{N_t}{N} \cdot (\text{impurity} - \frac{N_{t_R}}{N_t} \cdot \text{right_impurity} - \frac{N_{t_L}}{N_t} \cdot \text{left_impurity})$$

where N is the total number of samples, N_t is the number of samples at the current node, N_{t_L} is the number of samples in the left child, and N_{t_R} is the number of samples in the right child.

N , N_t , N_{t_R} and N_{t_L} all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

init [estimator or ‘zero’, optional (default=None)] An estimator object that is used to compute the initial predictions. `init` has to provide `fit` and `predict`. If ‘zero’, the initial raw predictions are set to zero. By default a `DummyEstimator` is used, predicting either the average target value (for loss='ls'), or a quantile for the other losses.

random_state [int, RandomState instance or None, optional (default=None)] If int, `random_state` is the seed used by the random number generator; If RandomState instance,

`random_state` is the random number generator; If `None`, the random number generator is the `RandomState` instance used by `np.random`.

max_features [int, float, string or `None`, optional (default=`None`)] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If “auto”, then `max_features=n_features`.
- If “sqrt”, then `max_features=sqrt(n_features)`.
- If “log2”, then `max_features=log2(n_features)`.
- If `None`, then `max_features=n_features`.

Choosing `max_features < n_features` leads to a reduction of variance and an increase in bias.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

alpha [float (default=0.9)] The alpha-quantile of the huber loss function and the quantile loss function. Only if `loss='huber'` or `loss='quantile'`.

verbose [int, default: 0] Enable verbose output. If 1 then it prints progress and performance once in a while (the more trees the lower the frequency). If greater than 1 then it prints progress and performance for every tree.

max_leaf_nodes [int or `None`, optional (default=`None`)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If `None` then unlimited number of leaf nodes.

warm_start [bool, default: False] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just erase the previous solution. See [the Glossary](#).

presort [bool or ‘auto’, optional (default=‘auto’)] Whether to presort the data to speed up the finding of best splits in fitting. Auto mode by default will use presorting on dense data and default to normal sorting on sparse data. Setting presort to true on sparse data will raise an error.

New in version 0.17: optional parameter `presort`.

validation_fraction [float, optional, default 0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if `n_iter_no_change` is set to an integer.

New in version 0.20.

n_iter_no_change [int, default `None`] `n_iter_no_change` is used to decide if early stopping will be used to terminate training when validation score is not improving. By default it is set to `None` to disable early stopping. If set to a number, it will set aside `validation_fraction` size of the training data as validation and terminate training when validation score is not improving in all of the previous `n_iter_no_change` numbers of iterations.

New in version 0.20.

tol [float, optional, default 1e-4] Tolerance for the early stopping. When the loss is not improving by at least tol for n_iter_no_change iterations (if set to a number), the training stops.

New in version 0.20.

Attributes

feature_importances_ [array, shape (n_features,)] Return the feature importances (the higher, the more important the feature).

oob_improvement_ [array, shape (n_estimators,)] The improvement in loss (= deviance) on the out-of-bag samples relative to the previous iteration. oob_improvement_[0] is the improvement in loss of the first stage over the init estimator.

train_score_ [array, shape (n_estimators,)] The i-th score train_score_[i] is the deviance (= loss) of the model at iteration i on the in-bag sample. If subsample == 1 this is the deviance on the training data.

loss_ [LossFunction] The concrete LossFunction object.

init_ [estimator] The estimator that provides the initial predictions. Set via the init argument or loss.init_estimator.

estimators_ [array of DecisionTreeRegressor, shape (n_estimators, 1)] The collection of fitted sub-estimators.

See also:

[sklearn.ensemble.HistGradientBoostingRegressor](#)

[sklearn.tree.DecisionTreeRegressor, RandomForestRegressor](#)

Notes

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data and max_features=n_features, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, random_state has to be fixed.

References

J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.

10. Friedman, Stochastic Gradient Boosting, 1999

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

Methods

<code>apply(self, X)</code>	Apply trees in the ensemble to X, return leaf indices.
<code>fit(self, X, y[, sample_weight, monitor])</code>	Fit the gradient boosting model.
<code>get_params(self[, deep])</code>	Get parameters for this estimator.
<code>predict(self, X)</code>	Predict regression target for X.

Continued on next page

Table 3.20 – continued from previous page

<code>score(self, X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of the prediction.
<code>set_params(self, **params)</code>	Set the parameters of this estimator.
<code>staged_predict(self, X)</code>	Predict regression target at each stage for X.

`__init__(self, loss='ls', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, min_impurity_split=None, init=None, random_state=None, max_features=None, alpha=0.9, verbose=0, max_leaf_nodes=None, warm_start=False, presort='auto', validation_fraction=0.1, n_iter_no_change=None, tol=0.0001)`

`apply(self, X)`

Apply trees in the ensemble to X, return leaf indices.

New in version 0.17.

Parameters

`X` [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted to a sparse `csr_matrix`.

Returns

`X_leaves` [array-like, shape (n_samples, n_estimators)] For each datapoint x in X and for each tree in the ensemble, return the index of the leaf x ends up in each estimator.

`feature_importances_`

Return the feature importances (the higher, the more important the feature).

Returns

`feature_importances_` [array, shape (n_features,)] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

`fit(self, X, y, sample_weight=None, monitor=None)`

Fit the gradient boosting model.

Parameters

`X` [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

`y` [array-like, shape (n_samples,)] Target values (strings or integers in classification, real numbers in regression) For classification, labels must correspond to classes.

`sample_weight` [array-like, shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

`monitor` [callable, optional] The monitor is called after each iteration with the current iteration, a reference to the estimator and the local variables of `_fit_stages` as keyword

arguments `callable(i, self, locals())`). If the callable returns `True` the fitting procedure is stopped. The monitor can be used for various things such as computing held-out estimates, early stopping, model introspect, and snapshoting.

Returns

self [object]

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If `True`, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*self*, *X*)

Predict regression target for *X*.

Parameters

X [{array-like, sparse matrix}, shape (*n_samples*, *n_features*)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

y [array, shape (*n_samples*,)] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as $(1 - u/v)$, where *u* is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and *v* is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape = (*n_samples*, *n_features*)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (*n_samples*, *n_samples_fitted*), where *n_samples_fitted* is the number of samples used in the fitting for the estimator.

y [array-like, shape = (*n_samples*) or (*n_samples*, *n_outputs*)] True values for *X*.

sample_weight [array-like, shape = [*n_samples*], optional] Sample weights.

Returns

score [float] R^2 of `self.predict(X)` wrt. *y*.

Notes

The R^2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer '`r2`' uses `multioutput='uniform_average'`).

set_params(self, **params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns**self****staged_predict**(self, X)

Predict regression target at each stage for X.

This method allows monitoring (i.e. determine error on testing set) after each stage.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

Returns

y [generator of array of shape (n_samples,)] The predicted value of the input samples.

Examples using `sklearn.ensemble.GradientBoostingRegressor`

- [Model Complexity Influence](#)
- [Plot individual and voting regression predictions](#)
- [Prediction Intervals for Gradient Boosting Regression](#)
- [Gradient Boosting regression](#)
- [Partial Dependence Plots](#)

3.3.3 Model evaluation: quantifying the quality of predictions

There are 3 different APIs for evaluating the quality of a model's predictions:

- **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 `model_selection.cross_val_score` and `model_selection.GridSearchCV`) rely on an internal `scoring` strategy. This is discussed in the section [The scoring parameter: defining model evaluation rules](#).
- **Metric functions:** The `metrics` module implements functions assessing prediction error for specific purposes. These metrics are detailed in sections on [Classification metrics](#), [Multilabel ranking metrics](#), [Regression metrics](#) and [Clustering 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.

The scoring parameter: defining model evaluation rules

Model selection and evaluation using tools, such as `model_selection.GridSearchCV` and `model_selection.cross_val_score`, take a scoring parameter that controls what metric they apply to the estimators evaluated.

Common cases: predefined values

For the most common use cases, you can designate a scorer object with the scoring parameter; the table below shows all possible values. All scorer objects follow the convention that **higher return values are better than lower return values**. Thus metrics which measure the distance between the model and the data, like `metrics.mean_squared_error`, are available as `neg_mean_squared_error` which return the negated value of the metric.

Scoring	Function	Comment
Classification		
'accuracy'	<code>metrics.accuracy_score</code>	
'balanced_accuracy'	<code>metrics.balanced_accuracy_score</code>	
'average_precision'	<code>metrics.average_precision_score</code>	
'brier_score_loss'	<code>metrics.brier_score_loss</code>	
'f1'	<code>metrics.f1_score</code>	for binary targets
'f1_micro'	<code>metrics.f1_score</code>	micro-averaged
'f1_macro'	<code>metrics.f1_score</code>	macro-averaged
'f1_weighted'	<code>metrics.f1_score</code>	weighted average
'f1_samples'	<code>metrics.f1_score</code>	by multilabel sample
'neg_log_loss'	<code>metrics.log_loss</code>	requires <code>predict_proba</code> support
'precision' etc.	<code>metrics.precision_score</code>	suffixes apply as with 'f1'
'recall' etc.	<code>metrics.recall_score</code>	suffixes apply as with 'f1'
'jaccard' etc.	<code>metrics.jaccard_score</code>	suffixes apply as with 'f1'
'roc_auc'	<code>metrics.roc_auc_score</code>	
Clustering		
'adjusted_mutual_info_score'	<code>metrics.adjusted_mutual_info_score</code>	
'adjusted_rand_score'	<code>metrics.adjusted_rand_score</code>	
'completeness_score'	<code>metrics.completeness_score</code>	
'fowlkes_mallows_score'	<code>metrics.fowlkes_mallows_score</code>	
'homogeneity_score'	<code>metrics.homogeneity_score</code>	
'mutual_info_score'	<code>metrics.mutual_info_score</code>	
'normalized_mutual_info_score'	<code>metrics.normalized_mutual_info_score</code>	
'v_measure_score'	<code>metrics.v_measure_score</code>	
Regression		
'explained_variance'	<code>metrics.explained_variance_score</code>	
'max_error'	<code>metrics.max_error</code>	
'neg_mean_absolute_error'	<code>metrics.mean_absolute_error</code>	
'neg_mean_squared_error'	<code>metrics.mean_squared_error</code>	
'neg_mean_squared_log_error'	<code>metrics.mean_squared_log_error</code>	
'neg_median_absolute_error'	<code>metrics.median_absolute_error</code>	
'r2'	<code>metrics.r2_score</code>	

Usage examples:

```
>>> from sklearn import svm, datasets
>>> from sklearn.model_selection import cross_val_score
>>> iris = datasets.load_iris()
```

```
>>> X, y = iris.data, iris.target
>>> clf = svm.SVC(gamma='scale', random_state=0)
>>> cross_val_score(clf, X, y, scoring='recall_macro',
...                   cv=5)
array([0.96..., 0.96..., 0.96..., 0.93..., 1.        ])
>>> model = svm.SVC()
>>> cross_val_score(model, X, y, cv=5, scoring='wrong_choice')
Traceback (most recent call last):
ValueError: 'wrong_choice' is not a valid scoring value. Use sorted(sklearn.metrics.
->SCORERS.keys()) to get valid options.
```

Note: The values listed by the `ValueError` exception correspond to the functions measuring prediction accuracy described in the following sections. The scorer objects for those functions are stored in the dictionary `sklearn.metrics.SCORERS`.

Defining your scoring strategy from metric functions

The module `sklearn.metrics` 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. When converting into a scorer object using `make_scoring`, set the `greater_is_better` parameter to `False` (`True` by default; see the parameter description below).

Metrics available for various machine learning tasks are detailed in sections below.

Many metrics are not given names to be used as `scoring` values, sometimes because they require additional parameters, such as `fbeta_score`. In such cases, you need to generate an appropriate scoring object. The simplest way to generate a callable object for scoring is by using `make_scoring`. That function converts metrics into callables that can be used for model evaluation.

One typical use case is to wrap an existing metric function from the library with non-default values for its parameters, such as the `beta` parameter for the `fbeta_score` function:

```
>>> from sklearn.metrics import fbeta_score, make_scoring
>>> ftwo_scoring = make_scoring(fbeta_score, beta=2)
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> grid = GridSearchCV(LinearSVC(), param_grid={'C': [1, 10]},
...                      scoring=ftwo_scoring, cv=5)
```

The second use case is to build a completely custom scorer object from a simple python function using `make_scoring`, which can take several parameters:

- the python function you want to use (`my_custom_loss_func` in the example below)
- whether the python function returns a score (`greater_is_better=True`, the default) or a loss (`greater_is_better=False`). If a loss, the output of the python function is negated by the scorer object, conforming to the cross validation convention that scorers return higher values for better models.
- for classification metrics only: whether the python function you provided requires continuous decision certainties (`needs_threshold=True`). The default value is `False`.
- any additional parameters, such as `beta` or `labels` in `f1_score`.

Here is an example of building custom scorers, and of using the `greater_is_better` parameter:

```
>>> import numpy as np
>>> def my_custom_loss_func(y_true, y_pred):
...     diff = np.abs(y_true - y_pred).max()
...     return np.log1p(diff)
...
>>> # score will negate the return value of my_custom_loss_func,
>>> # which will be np.log(2), 0.693, given the values for X
>>> # and y defined below.
>>> score = make_scorer(my_custom_loss_func, greater_is_better=False)
>>> X = [[1], [1]]
>>> y = [0, 1]
>>> from sklearn.dummy import DummyClassifier
>>> clf = DummyClassifier(strategy='most_frequent', random_state=0)
>>> clf = clf.fit(X, y)
>>> my_custom_loss_func(clf.predict(X), y)
0.69...
>>> score(clf, X, y)
-0.69...
```

Implementing your own scoring object

You can generate even more flexible model scorers 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 `estimator` prediction quality on `X`, with reference to `y`. Again, by convention higher numbers are better, so if your scorer returns loss, that value should be negated.

Note: Using custom scorers in functions where `n_jobs > 1`

While defining the custom scoring function alongside the calling function should work out of the box with the default `joblib` backend (`loky`), importing it from another module will be a more robust approach and work independently of the `joblib` backend.

For example, to use `n_jobs` greater than 1 in the example below, `custom_scoring_function` function is saved in a user-created module (`custom_scorer_module.py`) and imported:

```
>>> from custom_scorer_module import custom_scoring_function
>>> cross_val_score(model,
...     X_train,
...     y_train,
...     scoring=make_scorer(custom_scoring_function, greater_is_better=False),
...     cv=5,
...     n_jobs=-1)
```

Using multiple metric evaluation

Scikit-learn also permits evaluation of multiple metrics in `GridSearchCV`, `RandomizedSearchCV` and `cross_validate`.

There are two ways to specify multiple scoring metrics for the `scoring` parameter:

- As an iterable of string metrics::

```
>>> scoring = ['accuracy', 'precision']
```

- As a dict mapping the scorer name to the scoring function::

```
>>> from sklearn.metrics import accuracy_score
>>> from sklearn.metrics import make_scorer
>>> scoring = {'accuracy': make_scorer(accuracy_score),
...             'prec': 'precision'}
```

Note that the dict values can either be scorer functions or one of the predefined metric strings.

Currently only those scorer functions that return a single score can be passed inside the dict. Scorer functions that return multiple values are not permitted and will require a wrapper to return a single metric:

```
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics import confusion_matrix
>>> # A sample toy binary classification dataset
>>> X, y = datasets.make_classification(n_classes=2, random_state=0)
>>> svm = LinearSVC(random_state=0)
>>> def tn(y_true, y_pred): return confusion_matrix(y_true, y_pred)[0, 0]
>>> def fp(y_true, y_pred): return confusion_matrix(y_true, y_pred)[0, 1]
>>> def fn(y_true, y_pred): return confusion_matrix(y_true, y_pred)[1, 0]
>>> def tp(y_true, y_pred): return confusion_matrix(y_true, y_pred)[1, 1]
>>> scoring = {'tp': make_scorer(tp), 'tn': make_scorer(tn),
...             'fp': make_scorer(fp), 'fn': make_scorer(fn)}
>>> cv_results = cross_validate(svm.fit(X, y), X, y,
...                               scoring=scoring, cv=5)
>>> # Getting the test set true positive scores
>>> print(cv_results['test_tp'])
[10 9 8 7 8]
>>> # Getting the test set false negative scores
>>> print(cv_results['test_fn'])
[0 1 2 3 2]
```

Classification metrics

The `sklearn.metrics` module implements several loss, score, and utility functions to measure classification performance. Some metrics might require probability estimates of the positive class, confidence values, or binary decisions values. Most implementations allow each sample to provide a weighted contribution to the overall score, through the `sample_weight` parameter.

Some of these are restricted to the binary classification case:

<code>precision_recall_curve(y_true, probas_pred)</code>	Compute precision-recall pairs for different probability thresholds
<code>roc_curve(y_true, y_score[, pos_label, ...])</code>	Compute Receiver operating characteristic (ROC)
<code>balanced_accuracy_score(y_true, y_pred[, ...])</code>	Compute the balanced accuracy

Others also work in the multiclass case:

<code>cohen_kappa_score(y1, y2[, labels, weights, ...])</code>	Cohen's kappa: a statistic that measures inter-annotator agreement.
<code>confusion_matrix(y_true, y_pred[, labels, ...])</code>	Compute confusion matrix to evaluate the accuracy of a classification
<code>hinge_loss(y_true, pred_decision[, labels, ...])</code>	Average hinge loss (non-regularized)
<code>matthews_corrcoef(y_true, y_pred[, ...])</code>	Compute the Matthews correlation coefficient (MCC)

Some also work in the multilabel case:

<code>accuracy_score(y_true, y_pred[, normalize, ...])</code>	Accuracy classification score.
<code>classification_report(y_true, y_pred[, ...])</code>	Build a text report showing the main classification metrics
<code>f1_score(y_true, y_pred[, labels, ...])</code>	Compute the F1 score, also known as balanced F-score or F-measure
<code>fbeta_score(y_true, y_pred, beta[, labels, ...])</code>	Compute the F-beta score
<code>hamming_loss(y_true, y_pred[, labels, ...])</code>	Compute the average Hamming loss.
<code>jaccard_score(y_true, y_pred[, labels, ...])</code>	Jaccard similarity coefficient score
<code>log_loss(y_true, y_pred[, eps, normalize, ...])</code>	Log loss, aka logistic loss or cross-entropy loss.
<code>multilabel_confusion_matrix(y_true, y_pred)</code>	Compute a confusion matrix for each class or sample
<code>precision_recall_fscore_support(y_true, y_pred)</code>	Compute precision, recall, F-measure and support for each class
<code>precision_score(y_true, y_pred[, labels, ...])</code>	Compute the precision
<code>recall_score(y_true, y_pred[, labels, ...])</code>	Compute the recall
<code>zero_one_loss(y_true, y_pred[, normalize, ...])</code>	Zero-one classification loss.

And some work with binary and multilabel (but not multiclass) problems:

<code>average_precision_score(y_true, y_score[, ...])</code>	Compute average precision (AP) from prediction scores
<code>roc_auc_score(y_true, y_score[, average, ...])</code>	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

In the following sub-sections, we will describe each of those functions, preceded by some notes on common API and metric definition.

From binary to multiclass and multilabel

Some metrics are essentially defined for binary classification tasks (e.g. `f1_score`, `roc_auc_score`). In these cases, by default only the positive label is evaluated, assuming by default that the positive class is labelled 1 (though this may be configurable through the `pos_label` parameter).

In extending a binary metric to multiclass or multilabel problems, the data is treated as a collection of binary problems, one for each class. There are then a number of ways to average binary metric calculations across the set of classes, each of which may be useful in some scenario. Where available, you should select among these using the `average` parameter.

- "macro" simply calculates the mean of the binary metrics, giving equal weight to each class. In problems where infrequent classes are nonetheless important, macro-averaging may be a means of highlighting their performance. On the other hand, the assumption that all classes are equally important is often untrue, such that macro-averaging will over-emphasize the typically low performance on an infrequent class.

- "weighted" accounts for class imbalance by computing the average of binary metrics in which each class's score is weighted by its presence in the true data sample.
- "micro" gives each sample-class pair an equal contribution to the overall metric (except as a result of sample-weight). Rather than summing the metric per class, this sums the dividends and divisors that make up the per-class metrics to calculate an overall quotient. Micro-averaging may be preferred in multilabel settings, including multiclass classification where a majority class is to be ignored.
- "samples" applies only to multilabel problems. It does not calculate a per-class measure, instead calculating the metric over the true and predicted classes for each sample in the evaluation data, and returning their (sample_weight-weighted) average.
- Selecting average=None will return an array with the score for each class.

While multiclass data is provided to the metric, like binary targets, as an array of class labels, multilabel data is specified as an indicator matrix, in which cell [i, j] has value 1 if sample i has label j and value 0 otherwise.

Accuracy score

The `accuracy_score` function computes the accuracy, either the fraction (default) or the count (normalize=False) 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

$$\text{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y}_i = y_i)$$

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, 1], [1, 1]]), 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.

Balanced accuracy score

The `balanced_accuracy_score` function computes the **balanced accuracy**, which avoids inflated performance estimates on imbalanced datasets. It is the macro-average of recall scores per class or, equivalently, raw accuracy where each sample is weighted according to the inverse prevalence of its true class. Thus for balanced datasets, the score is equal to accuracy.

In the binary case, balanced accuracy is equal to the arithmetic mean of `sensitivity` (true positive rate) and `specificity` (true negative rate), or the area under the ROC curve with binary predictions rather than scores.

If the classifier performs equally well on either class, this term reduces to the conventional accuracy (i.e., the number of correct predictions divided by the total number of predictions).

In contrast, if the conventional accuracy is above chance only because the classifier takes advantage of an imbalanced test set, then the balanced accuracy, as appropriate, will drop to $\frac{1}{n_classes}$.

The score ranges from 0 to 1, or when `adjusted=True` is used, it rescaled to the range $\frac{1}{1-n_classes}$ to 1, inclusive, with performance at random scoring 0.

If y_i is the true value of the i -th sample, and w_i is the corresponding sample weight, then we adjust the sample weight to:

$$\hat{w}_i = \frac{w_i}{\sum_j 1(y_j = y_i) w_j}$$

where $1(x)$ is the indicator function. Given predicted \hat{y}_i for sample i , balanced accuracy is defined as:

$$\text{balanced-accuracy}(y, \hat{y}, w) = \frac{1}{\sum_i \hat{w}_i} \sum_i 1(\hat{y}_i = y_i) \hat{w}_i$$

With `adjusted=True`, balanced accuracy reports the relative increase from $\text{balanced-accuracy}(y, \mathbf{0}, w) = \frac{1}{n_classes}$. In the binary case, this is also known as ***Youden's J statistic***, or **informedness**.

Note: The multiclass definition here seems the most reasonable extension of the metric used in binary classification, though there is no certain consensus in the literature:

- Our definition: [\[Mosley2013\]](#), [\[Kelleher2015\]](#) and [\[Guyon2015\]](#), where [\[Guyon2015\]](#) adopt the adjusted version to ensure that random predictions have a score of 0 and perfect predictions have a score of 1..
 - Class balanced accuracy as described in [\[Mosley2013\]](#): the minimum between the precision and the recall for each class is computed. Those values are then averaged over the total number of classes to get the balanced accuracy.
 - Balanced Accuracy as described in [\[Urbanowicz2015\]](#): the average of sensitivity and specificity is computed for each class and then averaged over total number of classes.
-

References:

Cohen's kappa

The function `cohen_kappa_score` computes Cohen's kappa statistic. This measure is intended to compare labelings by different human annotators, not a classifier versus a ground truth.

The kappa score (see docstring) is a number between -1 and 1. Scores above .8 are generally considered good agreement; zero or lower means no agreement (practically random labels).

Kappa scores can be computed for binary or multiclass problems, but not for multilabel problems (except by manually computing a per-label score) and not for more than two annotators.

```
>>> from sklearn.metrics import cohen_kappa_score
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> cohen_kappa_score(y_true, y_pred)
0.4285714285714286
```

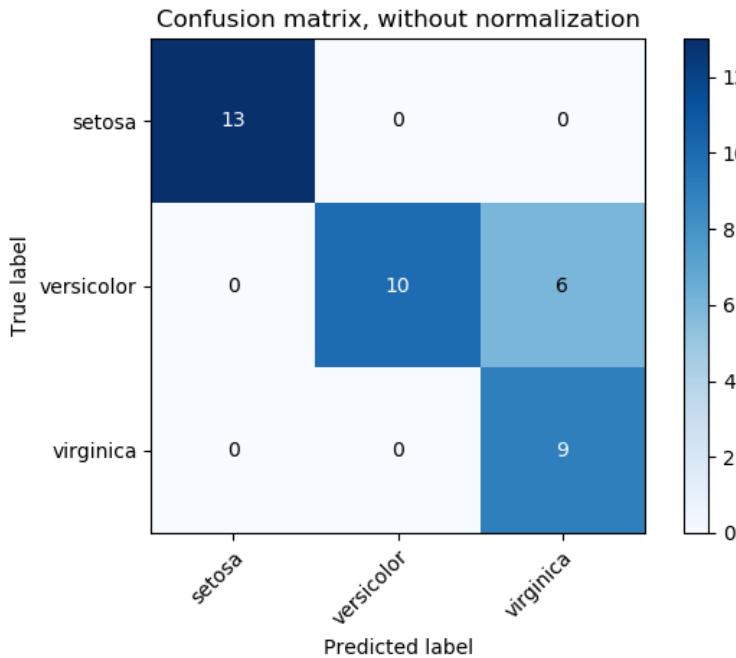
Confusion matrix

The `confusion_matrix` function evaluates classification accuracy by computing the confusion matrix with each row corresponding to the true class <https://en.wikipedia.org/wiki/Confusion_matrix>_. (Wikipedia and other references may use different convention for axes.)

By definition, entry i, j in a confusion matrix is the number of observations actually in group i , but predicted to be in group j . Here is an example:

```
>>> 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 is a visual representation of such a confusion matrix (this figure comes from the `Confusion matrix` example):



For binary problems, we can

get counts of true negatives, false positives, false negatives and true positives as follows:

```
>>> y_true = [0, 0, 0, 1, 1, 1, 1]
>>> y_pred = [0, 1, 0, 1, 0, 1, 1]
>>> tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
```

```
>>> tn, fp, fn, tp
(2, 1, 2, 3)
```

Example:

- See [Confusion matrix](#) for an example of using a confusion matrix to evaluate classifier output quality.
- See [Recognizing hand-written digits](#) for an example of using a confusion matrix to classify hand-written digits.
- See [Classification of text documents using sparse features](#) for an example of using a confusion matrix to classify text documents.

Classification report

The `classification_report` function builds a text report showing the main classification metrics. Here is 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, 1, 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    0.50    0.67      2

accuracy                           0.60      5
macro avg       0.56    0.50    0.49      5
weighted avg    0.67    0.60    0.59      5
```

Example:

- See [Recognizing hand-written digits](#) for an example of classification report usage for hand-written digits.
- See [Classification of text documents using sparse features](#) for an example of classification report usage for text documents.
- See [Parameter estimation using grid search with cross-validation](#) for an example of classification report usage for grid search with nested cross-validation.

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 label of a given sample, y_j is the corresponding true value, and n_{labels} is the number of classes or labels, then the Hamming loss L_{Hamming} between two samples is defined as:

$$L_{\text{Hamming}}(y, \hat{y}) = \frac{1}{n_{\text{labels}}} \sum_{j=0}^{n_{\text{labels}}-1} 1(\hat{y}_j \neq y_j)$$

where $\mathbf{1}(x)$ 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, 1], [1, 1]]), np.zeros((2, 2)))
0.75
```

Note: In multiclass classification, the Hamming loss corresponds to the Hamming distance between `y_true` and `y_pred` which is similar to the `Zero one loss` function. However, while zero-one loss penalizes prediction sets that do not strictly match true sets, the Hamming loss penalizes individual labels. Thus the Hamming loss, upper bounded by the zero-one loss, is always between zero and one, inclusive; and predicting a proper subset or superset of the true labels will give a Hamming loss between zero and one, exclusive.

Precision, recall and F-measures

Intuitively, `precision` is the ability of the classifier not to label as positive a sample that is negative, and `recall` is 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 its worst score at 0. With $\beta = 1$, F_β and F_1 are equivalent, and 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. The value is between 0 and 1 and higher is better. AP is defined as

$$\text{AP} = \sum_n (R_n - R_{n-1})P_n$$

where P_n and R_n are the precision and recall at the nth threshold. With random predictions, the AP is the fraction of positive samples.

References [[Manning2008](#)] and [[Everingham2010](#)] present alternative variants of AP that interpolate the precision-recall curve. Currently, `average_precision_score` does not implement any interpolated variant. References [[Davis2006](#)] and [[Flach2015](#)] describe why a linear interpolation of points on the precision-recall curve provides an overly-optimistic measure of classifier performance. This linear interpolation is used when computing area under the curve with the trapezoidal rule in `auc`.

Several functions allow you to analyze the precision, recall and F-measures score:

<code>average_precision_score(y_true, y_score[, ...])</code>	Compute average precision (AP) from prediction scores
<code>f1_score(y_true, y_pred[, labels, ...])</code>	Compute the F1 score, also known as balanced F-score or F-measure
<code>fbeta_score(y_true, y_pred, beta[, labels, ...])</code>	Compute the F-beta score
<code>precision_recall_curve(y_true, probas_pred)</code>	Compute precision-recall pairs for different probability thresholds

Continued on next page

Table 3.26 – continued from previous page

<code>precision_recall_fscore_support(y_true, y_pred)</code>	Compute precision, recall, F-measure and support for each class
<code>precision_score(y_true, y_pred[, labels, ...])</code>	Compute the precision
<code>recall_score(y_true, y_pred[, labels, ...])</code>	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 to classify text documents.
- See [Parameter estimation using grid search with cross-validation](#) for an example of `precision_score` and `recall_score` usage to estimate parameters using grid search with nested cross-validation.
- See [Precision-Recall](#) for an example of `precision_recall_curve` usage to evaluate classifier output quality.

References:

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)
	fn (false negative)	Missing result	Unexpected 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 are 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)
0.83...

```

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, as described [above](#). Note that if all labels are included, “micro”-averaging in a multiclass setting will produce precision, recall and F that are all identical to accuracy. Also note that “weighted” averaging may produce an F-score that is not between precision and recall.

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}
- $P(A, B) := \frac{|A \cap B|}{|A|}$ for some sets A and B
- $R(A, B) := \frac{|A \cap B|}{|B|}$ (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_β beta
"micro"	$P(y, \hat{y})$	$R(y, \hat{y})$	$F_\beta(y, \hat{y})$
"samples"	$\frac{1}{ S } \sum_{s \in S} P(y_s, \hat{y}_s)$	$\frac{1}{ S } \sum_{s \in S} R(y_s, \hat{y}_s)$	$\frac{1}{ S } \sum_{s \in S} F_\beta(y_s, \hat{y}_s)$
"macro"	$\frac{1}{ L } \sum_{l \in L} P(y_l, \hat{y}_l)$	$\frac{1}{ L } \sum_{l \in L} R(y_l, \hat{y}_l)$	$\frac{1}{ L } \sum_{l \in L} F_\beta(y_l, \hat{y}_l)$
"weighted"	$\frac{1}{\sum_{l \in L} \hat{y}_l } \sum_{l \in L} \hat{y}_l P(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l \in L} \hat{y}_l } \sum_{l \in L} \hat{y}_l R(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l \in L} \hat{y}_l } \sum_{l \in L} \hat{y}_l F_\beta(y_l, \hat{y}_l)$
None	$\langle P(y_l, \hat{y}_l) l \in L \rangle$	$\langle R(y_l, \hat{y}_l) l \in L \rangle$	$\langle F_\beta(y_l, \hat{y}_l) l \in L \rangle$

```
>>> 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.        ,
   , 0.        ]), array([2, 2, 2]...))
```

For multiclass classification with a “negative class”, it is possible to exclude some labels:

```
>>> metrics.recall_score(y_true, y_pred, labels=[1, 2], average='micro')
... # excluding 0, no labels were correctly recalled
0.0
```

Similarly, labels not present in the data sample may be accounted for in macro-averaging.

```
>>> metrics.precision_score(y_true, y_pred, labels=[0, 1, 2, 3], average='macro')
...
0.166...
```

Jaccard similarity coefficient score

The `jaccard_score` function computes the average of Jaccard similarity coefficients, also called the 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 predicted label set \hat{y}_i , is defined as

$$J(y_i, \hat{y}_i) = \frac{|y_i \cap \hat{y}_i|}{|y_i \cup \hat{y}_i|}.$$

`jaccard_score` works like `precision_recall_fscore_support` as a naively set-wise measure applying natively to binary targets, and extended to apply to multilabel and multiclass through the use of `From binary to multiclass and multilabel` (see above).

In the binary case:

```
>>> import numpy as np
>>> from sklearn.metrics import jaccard_score
>>> y_true = np.array([[0, 1, 1],
...                   [1, 1, 0]])
>>> y_pred = np.array([[1, 1, 1],
...                   [1, 0, 0]])
>>> jaccard_score(y_true[0], y_pred[0])
0.6666...
```

In the multilabel case with binary label indicators:

```
>>> jaccard_score(y_true, y_pred, average='samples')
0.5833...
>>> jaccard_score(y_true, y_pred, average='macro')
0.6666...
>>> jaccard_score(y_true, y_pred, average=None)
array([0.5, 0.5, 1.])
```

Multiclass problems are binarized and treated like the corresponding multilabel problem:

```
>>> y_pred = [0, 2, 1, 2]
>>> y_true = [0, 1, 2, 2]
>>> jaccard_score(y_true, y_pred, average=None)
...
array([1., 0., 0.33...])
>>> jaccard_score(y_true, y_pred, average='macro')
0.44...
>>> jaccard_score(y_true, y_pred, average='micro')
0.33...
```

Hinge loss

The `hinge_loss` function computes the average distance between the model and the data using hinge loss, a one-sided metric that considers only prediction errors. (Hinge loss is used in maximal margin classifiers such 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:

$$L_{\text{Hinge}}(y, w) = \max \{1 - wy, 0\} = |1 - wy|_+$$

If there are more than two labels, `hinge_loss` uses a multiclass variant due to Crammer & Singer. [Here](#) is the paper describing it.

If y_w is the predicted decision for true label and y_t is the maximum of the predicted decisions for all other labels, where predicted decisions are output by decision function, then multiclass hinge loss is defined by:

$$L_{\text{Hinge}}(y_w, y_t) = \max \{1 + y_t - y_w, 0\}$$

Here a small example demonstrating the use of the `hinge_loss` function with a svm classifier in a binary class problem:

```
>>> 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='squared_hinge', max_iter=1000,
          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...
```

Here is an example demonstrating the use of the `hinge_loss` function with a svm classifier in a multiclass problem:

```
>>> X = np.array([[0], [1], [2], [3]])
>>> Y = np.array([0, 1, 2, 3])
>>> labels = np.array([0, 1, 2, 3])
>>> est = svm.LinearSVC()
>>> est.fit(X, Y)
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
          intercept_scaling=1, loss='squared_hinge', max_iter=1000,
          multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
          verbose=0)
>>> pred_decision = est.decision_function([[[-1], [2], [3]]])
>>> y_true = [0, 2, 3]
>>> hinge_loss(y_true, pred_decision, labels)
0.56...
```

Log loss

Log loss, also called logistic regression loss or cross-entropy loss, is defined on probability estimates. It is commonly used in (multinomial) logistic regression and neural networks, as well as in some variants of expectation-maximization, and can be used to evaluate the probability outputs (`predict_proba`) of a classifier instead of 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 `log_loss` function 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.

Matthews correlation coefficient

The `matthews_corrcoef` function computes the Matthew's correlation coefficient (MCC) for binary classes. Quoting Wikipedia:

“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.”

In the binary (two-class) case, `tp`, `tn`, `fp` and `fn` are respectively the number of true positives, true negatives, false positives and false negatives, the MCC is defined as

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}.$$

In the multiclass case, the Matthews correlation coefficient can be defined in terms of a `confusion_matrix` C for K classes. To simplify the definition consider the following intermediate variables:

- $t_k = \sum_i^K C_{ik}$ the number of times class k truly occurred,
- $p_k = \sum_i^K C_{ki}$ the number of times class k was predicted,
- $c = \sum_k^K C_{kk}$ the total number of samples correctly predicted,
- $s = \sum_i^K \sum_j^K C_{ij}$ the total number of samples.

Then the multiclass MCC is defined as:

$$MCC = \frac{c \times s - \sum_k^K p_k \times t_k}{\sqrt{(s^2 - \sum_k^K p_k^2) \times (s^2 - \sum_k^K t_k^2)}}$$

When there are more than two labels, the value of the MCC will no longer range between -1 and +1. Instead the minimum value will be somewhere between -1 and 0 depending on the number and distribution of ground true labels. The maximum value is always +1.

Here is 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...
```

Multi-label confusion matrix

The `multilabel_confusion_matrix` function computes class-wise (default) or sample-wise (sample-wise=True) multilabel confusion matrix to evaluate the accuracy of a classification. `multilabel_confusion_matrix`

also treats multiclass data as if it were multilabel, as this is a transformation commonly applied to evaluate multiclass problems with binary classification metrics (such as precision, recall, etc.).

When calculating class-wise multilabel confusion matrix C , the count of true negatives for class i is $C_{i,0,0}$, false negatives is $C_{i,1,0}$, true positives is $C_{i,1,1}$ and false positives is $C_{i,0,1}$.

Here is an example demonstrating the use of the `multilabel_confusion_matrix` function with `multilabel indicator matrix` input:

```
>>> import numpy as np
>>> from sklearn.metrics import multilabel_confusion_matrix
>>> y_true = np.array([[1, 0, 1],
...                   [0, 1, 0]])
>>> y_pred = np.array([[1, 0, 0],
...                   [0, 1, 1]])
>>> multilabel_confusion_matrix(y_true, y_pred)
array([[[1, 0],
       [0, 1]],

       [[1, 0],
       [0, 1]],

       [[0, 1],
       [1, 0]]])
```

Or a confusion matrix can be constructed for each sample's labels:

```
>>> multilabel_confusion_matrix(y_true, y_pred, samplewise=True)
array([[[1, 0],
       [1, 1]],

       [[1, 1],
       [0, 1]]])
```

Here is an example demonstrating the use of the `multilabel_confusion_matrix` function with `multiclass` input:

```
>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> multilabel_confusion_matrix(y_true, y_pred,
...                               labels=["ant", "bird", "cat"])
array([[[3, 1],
       [0, 2]],

       [[5, 0],
       [1, 0]],

       [[2, 1],
       [1, 2]]])
```

Here are some examples demonstrating the use of the `multilabel_confusion_matrix` function to calculate recall (or sensitivity), specificity, fall out and miss rate for each class in a problem with multilabel indicator matrix input.

Calculating `recall` (also called the true positive rate or the sensitivity) for each class:

```
>>> y_true = np.array([[0, 0, 1],
...                   [0, 1, 0],
...                   [1, 1, 0]])
```

```
>>> y_pred = np.array([[0, 1, 0],
...                     [0, 0, 1],
...                     [1, 1, 0]])
>>> mcm = multilabel_confusion_matrix(y_true, y_pred)
>>> tn = mcm[:, 0, 0]
>>> tp = mcm[:, 1, 1]
>>> fn = mcm[:, 1, 0]
>>> fp = mcm[:, 0, 1]
>>> tp / (tp + fn)
array([1., 0.5, 0.])
```

Calculating `specificity` (also called the true negative rate) for each class:

```
>>> tn / (tn + fp)
array([1., 0., 0.5])
```

Calculating `false_out` (also called the false positive rate) for each class:

```
>>> fp / (fp + tn)
array([0., 1., 0.5])
```

Calculating `miss_rate` (also called the false negative rate) for each class:

```
>>> fn / (fn + tp)
array([0., 0.5, 1.])
```

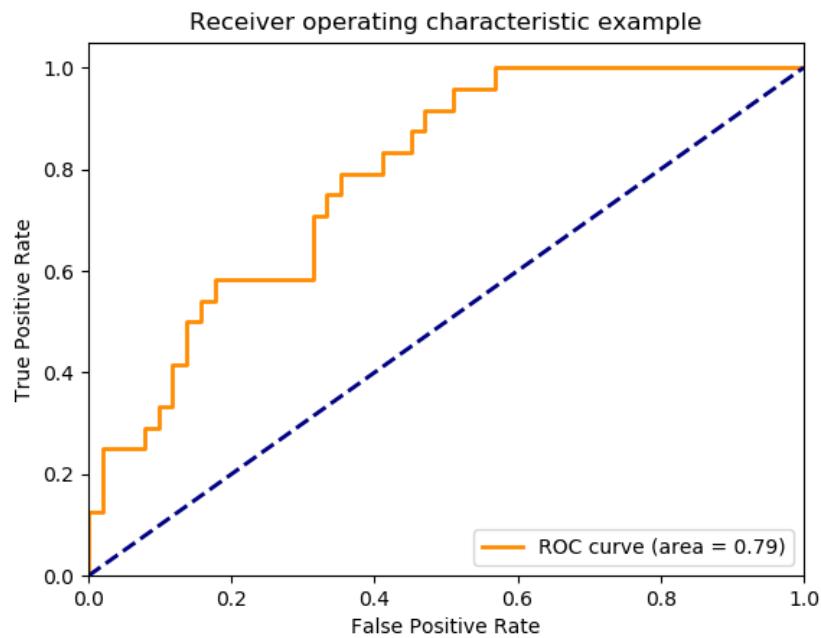
Receiver operating characteristic (ROC)

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 is 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., 0.5, 0.5, 1.])
>>> tpr
array([0., 0.5, 0.5, 1., 1.])
>>> thresholds
array([1.8, 0.8, 0.4, 0.35, 0.1])
```



This figure shows an example of such an 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](#).

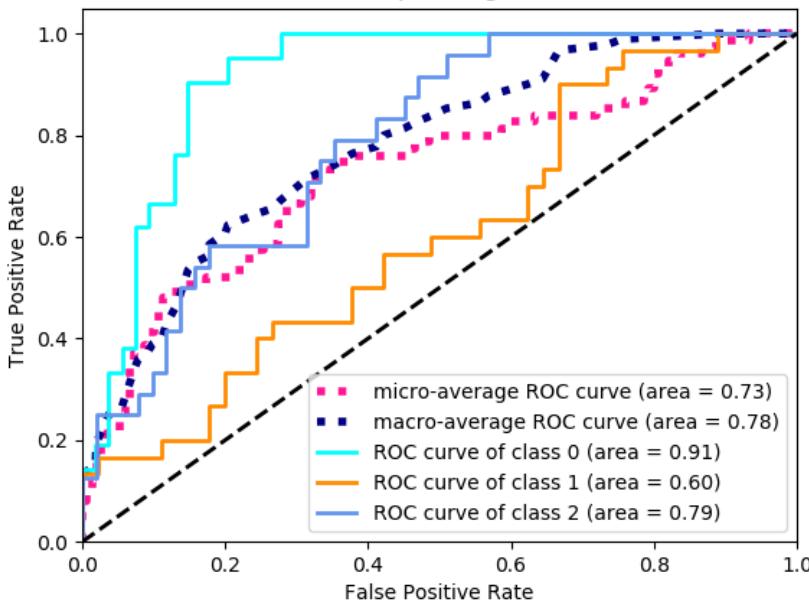
```
>>> import numpy as np
>>> from sklearn.metrics import roc_auc_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> roc_auc_score(y_true, y_scores)
0.75
```

In multi-label classification, the `roc_auc_score` function is extended by averaging over the labels as [above](#).

Compared to metrics such as the subset accuracy, the Hamming loss, or the F1 score, ROC doesn't require optimizing a threshold for each label. The `roc_auc_score` function can also be used in multi-class classification, if the predicted outputs have been binarized.

In applications where a high false positive rate is not tolerable the parameter `max_fpr` of `roc_auc_score` can be used to summarize the ROC curve up to the given limit.

Some extension of Receiver operating characteristic to multi-class

**Examples:**

- See [Receiver Operating Characteristic \(ROC\)](#) for an example of using ROC to evaluate the quality of the output of a classifier.
- See [Receiver Operating Characteristic \(ROC\) with cross validation](#) for an example of using ROC to evaluate classifier output quality, using cross-validation.
- See [Species distribution modeling](#) for an example of using ROC to model species distribution.

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 default, 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` scores a subset as one if its labels strictly match the predictions, and as a zero if there are any errors. By default, the function returns the percentage of imperfectly predicted subsets. To get the count of such subsets instead, set `normalize` to `False`.

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, where the first label set [0,1] has an error:

```
>>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5

>>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)), normalize=False)
1
```

Example:

- See [Recursive feature elimination with cross-validation](#) for an example of zero one loss usage to perform recursive feature elimination with cross-validation.

Brier score loss

The `brier_score_loss` function computes the Brier score for binary classes. Quoting Wikipedia:

“The Brier score is a proper score function that measures the accuracy of probabilistic predictions. It is applicable to tasks in which predictions must assign probabilities to a set of mutually exclusive discrete outcomes.”

This function returns a score of the mean square difference between the actual outcome and the predicted probability of the possible outcome. The actual outcome has to be 1 or 0 (true or false), while the predicted probability of the actual outcome can be a value between 0 and 1.

The brier score loss is also between 0 to 1 and the lower the score (the mean square difference is smaller), the more accurate the prediction is. It can be thought of as a measure of the “calibration” of a set of probabilistic predictions.

$$BS = \frac{1}{N} \sum_{t=1}^N (f_t - o_t)^2$$

where : N is the total number of predictions, f_t is the predicted probability of the actual outcome o_t .

Here is a small example of usage of this function::

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss
>>> y_true = np.array([0, 1, 1, 0])
>>> y_true_categorical = np.array(["spam", "ham", "ham", "spam"])
>>> y_prob = np.array([0.1, 0.9, 0.8, 0.4])
>>> y_pred = np.array([0, 1, 1, 0])
>>> brier_score_loss(y_true, y_prob)
0.055
>>> brier_score_loss(y_true, 1 - y_prob, pos_label=0)
0.055
>>> brier_score_loss(y_true_categorical, y_prob, pos_label="ham")
0.055
>>> brier_score_loss(y_true, y_prob > 0.5)
0.0
```

Example:

- See [Probability calibration of classifiers](#) for an example of Brier score loss usage to perform probability calibration of classifiers.

References:

- G. Brier, [Verification of forecasts expressed in terms of probability](#), Monthly weather review 78.1 (1950)

Multilabel ranking metrics

In multilabel learning, each sample can have any number of ground truth labels associated with it. The goal is to give high scores and better rank to the ground truth labels.

Coverage error

The [coverage_error](#) function computes the average number of labels that have to be included in the final prediction such that all true labels are predicted. This is useful if you want to know how many top-scored-labels you have to predict in average without missing any true one. The best value of this metric is thus the average number of true labels.

Note: Our implementation's score is 1 greater than the one given in Tsoumakas et al., 2010. This extends it to handle the degenerate case in which an instance has 0 true labels.

Formally, given a binary indicator matrix of the ground truth labels $y \in \{0, 1\}^{n_{\text{samples}} \times n_{\text{labels}}}$ and the score associated with each label $\hat{f} \in \mathbb{R}^{n_{\text{samples}} \times n_{\text{labels}}}$, the coverage is defined as

$$\text{coverage}(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \max_{j:y_{ij}=1} \text{rank}_{ij}$$

with $\text{rank}_{ij} = |\{k : \hat{f}_{ik} \geq \hat{f}_{ij}\}|$. Given the rank definition, ties in `y_scores` are broken by giving the maximal rank that would have been assigned to all tied values.

Here is a small example of usage of this function:

```
>>> import numpy as np
>>> from sklearn.metrics import coverage_error
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> coverage_error(y_true, y_score)
2.5
```

Label ranking average precision

The [label_ranking_average_precision_score](#) function implements label ranking average precision (LRAP). This metric is linked to the [average_precision_score](#) function, but is based on the notion of label ranking instead of precision and recall.

Label ranking average precision (LRAP) averages over the samples the answer to the following question: for each ground truth label, what fraction of higher-ranked labels were true labels? This performance measure will be higher if you are able to give better rank to the labels associated with each sample. The obtained score is always strictly greater than 0, and the best value is 1. If there is exactly one relevant label per sample, label ranking average precision is equivalent to the [mean reciprocal rank](#).

Formally, given a binary indicator matrix of the ground truth labels $y \in \{0, 1\}^{n_{\text{samples}} \times n_{\text{labels}}}$ and the score associated with each label $\hat{f} \in \mathbb{R}^{n_{\text{samples}} \times n_{\text{labels}}}$, the average precision is defined as

$$LRAP(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \frac{1}{\|y_i\|_0} \sum_{j:y_{ij}=1} \frac{|\mathcal{L}_{ij}|}{\text{rank}_{ij}}$$

where $\mathcal{L}_{ij} = \left\{ k : y_{ik} = 1, \hat{f}_{ik} \geq \hat{f}_{ij} \right\}$, $\text{rank}_{ij} = \left| \left\{ k : \hat{f}_{ik} \geq \hat{f}_{ij} \right\} \right|$, $|\cdot|$ computes the cardinality of the set (i.e., the number of elements in the set), and $\|\cdot\|_0$ is the ℓ_0 “norm” (which computes the number of nonzero elements in a vector).

Here is a small example of usage of this function:

```
>>> import numpy as np
>>> from sklearn.metrics import label_ranking_average_precision_score
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> label_ranking_average_precision_score(y_true, y_score)
0.416...
```

Ranking loss

The [`label_ranking_loss`](#) function computes the ranking loss which averages over the samples the number of label pairs that are incorrectly ordered, i.e. true labels have a lower score than false labels, weighted by the inverse of the number of ordered pairs of false and true labels. The lowest achievable ranking loss is zero.

Formally, given a binary indicator matrix of the ground truth labels $y \in \{0, 1\}^{n_{\text{samples}} \times n_{\text{labels}}}$ and the score associated with each label $\hat{f} \in \mathbb{R}^{n_{\text{samples}} \times n_{\text{labels}}}$, the ranking loss is defined as

$$\text{ranking_loss}(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \frac{1}{\|y_i\|_0(n_{\text{labels}} - \|y_i\|_0)} \left| \left\{ (k, l) : \hat{f}_{ik} \leq \hat{f}_{il}, y_{ik} = 1, y_{il} = 0 \right\} \right|$$

where $|\cdot|$ computes the cardinality of the set (i.e., the number of elements in the set) and $\|\cdot\|_0$ is the ℓ_0 “norm” (which computes the number of nonzero elements in a vector).

Here is a small example of usage of this function:

```
>>> import numpy as np
>>> from sklearn.metrics import label_ranking_loss
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> label_ranking_loss(y_true, y_score)
0.75...
>>> # With the following prediction, we have perfect and minimal loss
>>> y_score = np.array([[1.0, 0.1, 0.2], [0.1, 0.2, 0.9]])
>>> label_ranking_loss(y_true, y_score)
0.0
```

References:

- Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

Regression metrics

The `sklearn.metrics` module implements several loss, score, and utility functions to measure regression performance. Some of those have been enhanced to handle the multioutput case: `mean_squared_error`, `mean_absolute_error`, `explained_variance_score` and `r2_score`.

These functions have an `multioutput` keyword argument which specifies the way the scores or losses for each individual target should be averaged. The default is '`uniform_average`', which specifies a uniformly weighted mean over outputs. If an ndarray of shape `(n_outputs,)` is passed, then its entries are interpreted as weights and an according weighted average is returned. If `multioutput` is '`raw_values`' is specified, then all unaltered individual scores or losses will be returned in an array of shape `(n_outputs,)`.

The `r2_score` and `explained_variance_score` accept an additional value '`variance_weighted`' for the `multioutput` parameter. This option leads to a weighting of each individual score by the variance of the corresponding target variable. This setting quantifies the globally captured unscaled variance. If the target variables are of different scale, then this score puts more importance on well explaining the higher variance variables. `multioutput='variance_weighted'` is the default value for `r2_score` for backward compatibility. This will be changed to `uniform_average` in the future.

Explained variance score

The `explained_variance_score` computes the explained variance regression score.

If \hat{y} is the estimated target output, y the corresponding (correct) target output, and Var is Variance, the square of the standard deviation, then the explained variance is estimated as follow:

$$\text{explained_variance}(y, \hat{y}) = 1 - \frac{\text{Var}\{y - \hat{y}\}}{\text{Var}\{y\}}$$

The best possible score is 1.0, lower values are worse.

Here is 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...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> explained_variance_score(y_true, y_pred, multioutput='raw_values')
...
array([0.967..., 1.        ])
>>> explained_variance_score(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.990...
```

Max error

The `max_error` function computes the maximum residual error , a metric that captures the worst case error between the predicted value and the true value. In a perfectly fitted single output regression model, `max_error` would be 0 on the training set and though this would be highly unlikely in the real world, this metric shows the extent of error that the model had when it was fitted.

If \hat{y}_i is the predicted value of the i -th sample, and y_i is the corresponding true value, then the max error is defined as

$$\text{Max Error}(y, \hat{y}) = \max(|y_i - \hat{y}_i|)$$

Here is a small example of usage of the `max_error` function:

```
>>> from sklearn.metrics import max_error
>>> y_true = [3, 2, 7, 1]
>>> y_pred = [9, 2, 7, 1]
>>> max_error(y_true, y_pred)
6
```

The `max_error` does not support multioutput.

Mean absolute error

The `mean_absolute_error` function computes mean absolute error, a risk metric 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

$$\text{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} |y_i - \hat{y}_i|.$$

Here is 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
>>> mean_absolute_error(y_true, y_pred, multioutput='raw_values')
array([0.5, 1.])
>>> mean_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.85...
```

Mean squared error

The `mean_squared_error` function computes mean square error, a risk metric corresponding to the expected value of the squared (quadratic) error or 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 is 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.

Mean squared logarithmic error

The `mean_squared_log_error` function computes a risk metric corresponding to the expected value of the squared logarithmic (quadratic) error or 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 logarithmic error (MSLE) estimated over n_{samples} is defined as

$$\text{MSLE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (\log_e(1 + y_i) - \log_e(1 + \hat{y}_i))^2.$$

Where $\log_e(x)$ means the natural logarithm of x . This metric is best to use when targets having exponential growth, such as population counts, average sales of a commodity over a span of years etc. Note that this metric penalizes an under-predicted estimate greater than an over-predicted estimate.

Here is a small example of usage of the `mean_squared_log_error` function:

```
>>> from sklearn.metrics import mean_squared_log_error
>>> y_true = [3, 5, 2.5, 7]
>>> y_pred = [2.5, 5, 4, 8]
>>> mean_squared_log_error(y_true, y_pred)
0.039...
>>> y_true = [[0.5, 1], [1, 2], [7, 6]]
>>> y_pred = [[0.5, 2], [1, 2.5], [8, 8]]
>>> mean_squared_log_error(y_true, y_pred)
0.044...
```

Median absolute error

The `median_absolute_error` is particularly interesting because it is robust to outliers. The loss is calculated by taking the median of all absolute differences between the target and the prediction.

If \hat{y}_i is the predicted value of the i -th sample and y_i is the corresponding true value, then the median absolute error (MedAE) estimated over n_{samples} is defined as

$$\text{MedAE}(y, \hat{y}) = \text{median}(|y_1 - \hat{y}_1|, \dots, |y_n - \hat{y}_n|).$$

The `median_absolute_error` does not support multioutput.

Here is a small example of usage of the `median_absolute_error` function:

```
>>> from sklearn.metrics import median_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> median_absolute_error(y_true, y_pred)
0.5
```

R² score, the coefficient of determination

The `r2_score` function computes the coefficient of determination, usually denoted as R².

It represents the proportion of variance (of y) that has been explained by the independent variables in the model. It provides an indication of goodness of fit and therefore a measure of how well unseen samples are likely to be predicted by the model, through the proportion of explained variance.

As such variance is dataset dependent, R² may not be meaningfully comparable across different datasets. Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R² score of 0.0.

If \hat{y}_i is the predicted value of the i -th sample and y_i is the corresponding true value for total n samples, the estimated R² is defined as:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

where $\bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$ and $\sum_{i=1}^n (y_i - \hat{y}_i)^2 = \sum_{i=1}^n \epsilon_i^2$.

Note that `r2_score` calculates unadjusted R² without correcting for bias in sample variance of y.

Here is 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, multioutput='variance_weighted')
...
0.938...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred, multioutput='uniform_average')
...
0.936...
>>> r2_score(y_true, y_pred, multioutput='raw_values')
...
array([0.965..., 0.908...])
>>> r2_score(y_true, y_pred, multioutput=[0.3, 0.7])
```

```
...  
0.925...
```

Example:

- See [Lasso and Elastic Net for Sparse Signals](#) for an example of R² score usage to evaluate Lasso and Elastic Net on sparse signals.

Clustering metrics

The `sklearn.metrics` module implements several loss, score, and utility functions. For more information see the [Clustering performance evaluation](#) section for instance clustering, and [Biclustering evaluation](#) for biclustering.

Dummy estimators

When doing supervised learning, a simple sanity check consists of comparing one's estimator against simple rules of thumb. `DummyClassifier` implements several such simple strategies for classification:

- `stratified` generates random predictions by respecting the training set class distribution.
- `most_frequent` always predicts the most frequent label in the training set.
- `prior` always predicts the class that maximizes the class prior (like `most_frequent`) and `predict_proba` returns the class prior.
- `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.model_selection 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(gamma='scale', kernel='rbf', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.94...
```

We see that the accuracy was boosted to almost 100%. A cross validation strategy is recommended for a better estimate of the accuracy, 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 [Tuning the hyper-parameters of an estimator](#) section for details.

More generally, when the accuracy of a classifier is too close to random, it probably means that something went wrong: features are not helpful, a hyperparameter is not correctly tuned, the classifier is suffering from class imbalance, etc...

[DummyRegressor](#) also implements four simple rules of thumb for regression:

- mean always predicts the mean of the training targets.
- median always predicts the median of the training targets.
- quantile always predicts a user provided quantile of the training targets.
- constant always predicts a constant value that is provided by the user.

In all these strategies, the `predict` method completely ignores the input data.

3.3.4 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.

An alternative to pickling is to export the model to another format using one of the model export tools listed under [Related Projects](#). Unlike pickling, once exported you cannot recover the full Scikit-learn estimator object, but you can deploy the model for prediction, usually by using tools supporting open model interchange formats such as '[ONNX](#)' or '[PMML](#)'.

Persistence example

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

```
>>> from sklearn import svm
>>> from sklearn import datasets
>>> clf = svm.SVC(gamma='scale')
>>> 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,
     decision_function_shape='ovr', degree=3, gamma='scale', 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[0:1])
array([0])
>>> y[0]
0
```

In the specific case of scikit-learn, it may be better to use joblib's replacement of pickle (`dump` & `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 joblib import dump, load
>>> dump(clf, 'filename.joblib')
```

Later you can load back the pickled model (possibly in another Python process) with:

```
>>> clf = load('filename.joblib')
```

Note: `dump` and `load` functions also accept file-like object instead of filenames. More information on data persistence with Joblib is available [here](#).

Security & maintainability limitations

pickle (and joblib by extension), has some issues regarding maintainability and security. Because of this,

- Never unpickle untrusted data as it could lead to malicious code being executed upon loading.
- While models saved using one version of scikit-learn might load in other versions, this is entirely unsupported and inadvisable. It should also be kept in mind that operations performed on such data could give different and unexpected results.

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 an 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.

Since a model internal representation may be different on two different architectures, dumping a model on one architecture and loading it on another architecture is not supported.

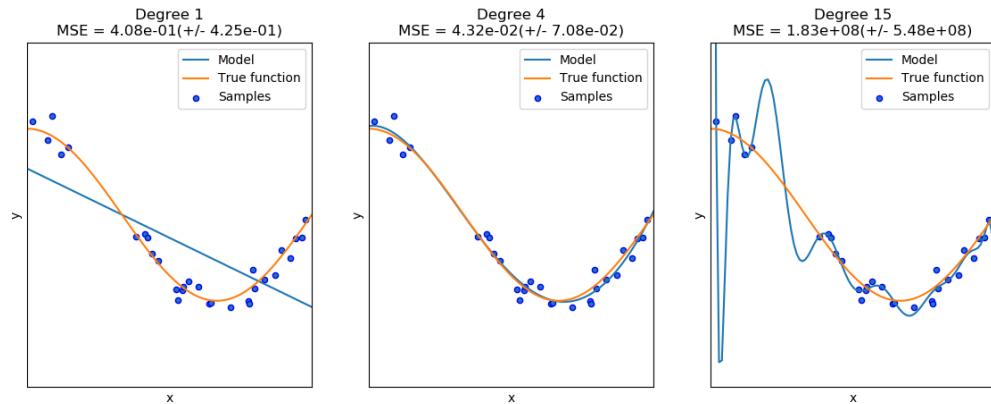
If you want to know more about these issues and explore other possible serialization methods, please refer to this [talk](#) by [Alex Gaynor](#).

3.3.5 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 hyper-parameters 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:

- *Underfitting vs. Overfitting*
- *Plotting Validation Curves*
- *Plotting Learning Curves*

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 [Tuning the hyper-parameters of an estimator](#)) 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:

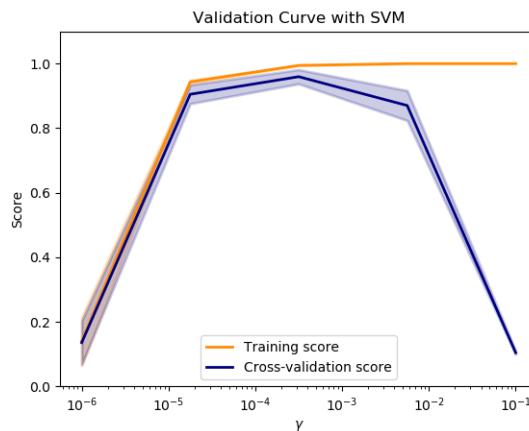
```
>>> import numpy as np
>>> from sklearn.model_selection import validation_curve
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import Ridge

>>> 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),
...                                                 cv=5)
...
>>> train_scores
array([[0.93..., 0.94..., 0.92..., 0.91..., 0.92...],
       [0.93..., 0.94..., 0.92..., 0.91..., 0.92...],
       [0.51..., 0.52..., 0.49..., 0.47..., 0.49...]])
>>> valid_scores
array([[0.90..., 0.84..., 0.94..., 0.96..., 0.93...],
       [0.90..., 0.84..., 0.94..., 0.96..., 0.93...],
       [0.46..., 0.25..., 0.50..., 0.49..., 0.52...]])
```

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.



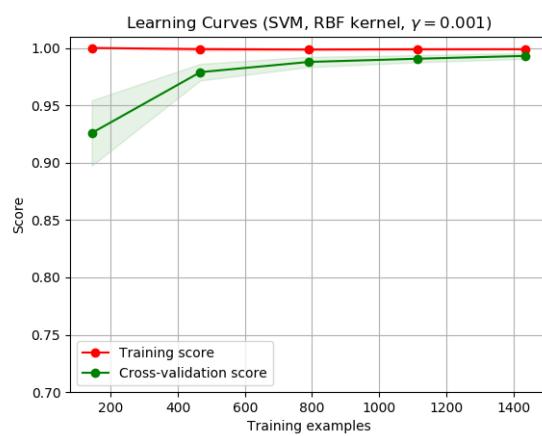
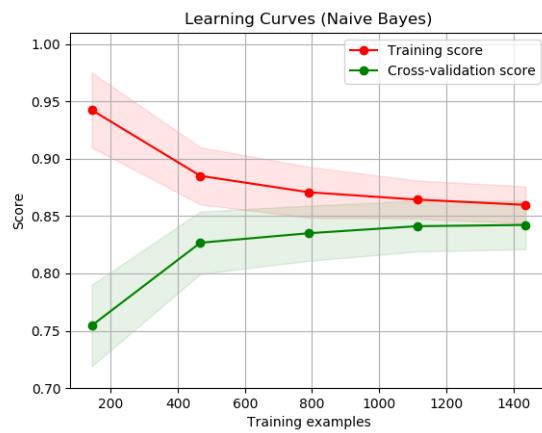
Learning curve

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.model_selection 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...]])
```

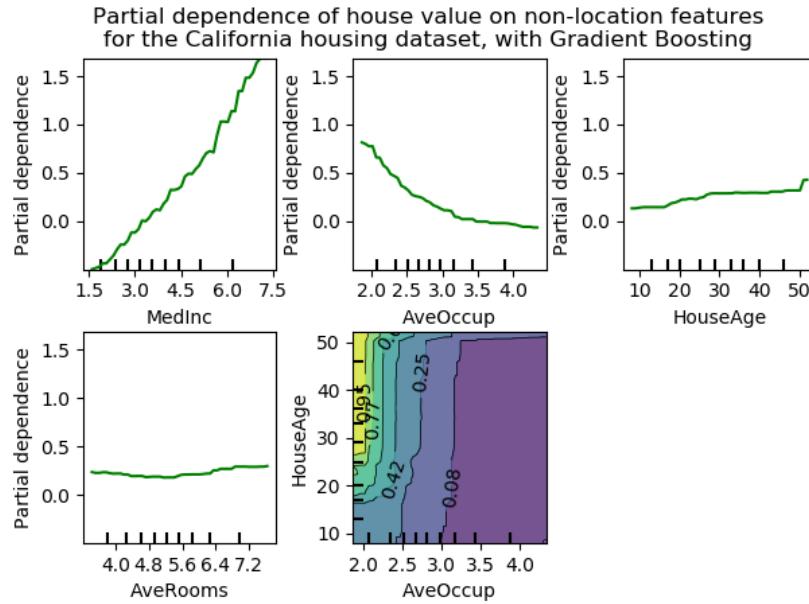
3.4 Inspection

3.4.1 Partial dependence plots

Partial dependence plots (PDP) show the dependence between the target response¹ 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 as a function of the ‘target’ features.

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, with a *GradientBoostingRegressor*:



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;

¹ For classification, the target response may be the probability of a class (the positive class for binary classification), or the decision function.

we can clearly see a linear relationship among them. Note that PDPs assume that the target features are independent from the complement features, and this assumption is often violated in practice.

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 average occupants per household. We can clearly see an interaction between the two features: for an average occupancy greater than two, the house price is nearly independent of the house age, whereas for values less than 2 there is a strong dependence on age.

The `sklearn.inspection` module 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.inspection 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)]
>>> plot_partial_dependence(clf, X, features)
```

You can access the newly created figure and Axes objects using `plt.gcf()` and `plt.gca()`.

For multi-class classification, you need to set the class label for which the PDPs should be created via the `target` 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)]
>>> plot_partial_dependence(mc_clf, X, features, target=0)
```

The same parameter `target` is used to specify the target in multi-output regression settings.

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

```
>>> from sklearn.inspection import partial_dependence

>>> pdp, axes = partial_dependence(clf, X, [0])
>>> pdp
array([[ 2.466...,  2.466..., ...,
       -1.624..., -1.592..., ...,
```

The values at which the partial dependence should be evaluated are directly generated from `X`. For 2-way partial dependence, a 2D-grid of values is generated. The `values` field returned by `sklearn.inspection.partial_dependence` gives the actual values used in the grid for each target feature. They also correspond to the axis of the plots.

For each value of the ‘target’ features in the grid the partial dependence function needs to marginalize the predictions of the estimator over all possible values of the ‘complement’ features. With the ‘brute’ method, this is done by replacing every target feature value of `X` by those in the grid, and computing the average prediction.

In decision trees this can be evaluated efficiently without reference to the training data (‘recursion’ method). 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. Note that with the 'recursion' method, X is only used to generate the grid, not to compute the averaged predictions. The averaged predictions will always be computed on the data with which the trees were trained.

Examples:

- *Partial Dependence Plots*

References

3.5 Dataset transformations

scikit-learn provides a library of transformers, which may clean (see [Preprocessing data](#)), reduce (see [Unsupervised dimensionality reduction](#)), expand (see [Kernel Approximation](#)) or generate (see [Feature extraction](#)) feature representations.

Like other estimators, these are represented by classes with a `fit` method, which learns model parameters (e.g. mean and standard deviation for normalization) from a training set, and a `transform` method which applies this transformation model to unseen data. `fit_transform` may be more convenient and efficient for modelling and transforming the training data simultaneously.

Combining such transformers, either in parallel or series is covered in [Pipelines and composite estimators](#). [Pairwise metrics, Affinities and Kernels](#) covers transforming feature spaces into affinity matrices, while [Transforming the prediction target \(\$y\$ \)](#) considers transformations of the target space (e.g. categorical labels) for use in scikit-learn.

3.5.1 Pipelines and composite estimators

Transformers are usually combined with classifiers, regressors or other estimators to build a composite estimator. The most common tool is a [Pipeline](#). Pipeline is often used in combination with [FeatureUnion](#) which concatenates the output of transformers into a composite feature space. [TransformedTargetRegressor](#) deals with transforming the `target` (i.e. log-transform y). In contrast, Pipelines only transform the observed data (X).

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 multiple purposes here:

Convenience and encapsulation 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.

Safety Pipelines help avoid leaking statistics from your test data into the trained model in cross-validation, by ensuring that the same samples are used to train the transformers and predictors.

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.).

Usage

Construction

The `Pipeline` is built using a list of (key, value) pairs, where the key is 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()), ('clf', SVC())]
>>> pipe = Pipeline(estimators)
>>> pipe
Pipeline(memory=None,
         steps=[('reduce_dim', PCA(copy=True, ...)),
                ('clf', SVC(C=1.0, ...))], 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(memory=None,
         steps=[('binarizer', Binarizer(copy=True, threshold=0.0)),
                ('multinomialnb', MultinomialNB(alpha=1.0,
                                                class_prior=None,
                                                fit_prior=True))],
         verbose=False)
```

Accessing steps

The estimators of a pipeline are stored as a list in the `steps` attribute, but can be accessed by index or name by indexing (with `[idx]`) the Pipeline:

```
>>> pipe.steps[0]
('reduce_dim', PCA(copy=True, iterated_power='auto', n_components=None,
                     random_state=None, svd_solver='auto', tol=0.0,
                     whiten=False))
>>> pipe[0]
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
     svd_solver='auto', tol=0.0, whiten=False)
>>> pipe['reduce_dim']
PCA(copy=True, ...)
```

Pipeline's `named_steps` attribute allows accessing steps by name with tab completion in interactive environments:

```
>>> pipe.named_steps.reduce_dim is pipe['reduce_dim']
True
```

A sub-pipeline can also be extracted using the slicing notation commonly used for Python Sequences such as lists or strings (although only a step of 1 is permitted). This is convenient for performing only some of the transformations (or their inverse):

```
>>> pipe[:1]
Pipeline(memory=None, steps=[('reduce_dim', PCA(copy=True, ...))], ...)
>>> pipe[-1:]
Pipeline(memory=None, steps=[('clf', SVC(C=1.0, ...))], ...)
```

Nested parameters

Parameters of the estimators in the pipeline can be accessed using the `<estimator>__<parameter>` syntax:

```
>>> pipe.set_params(clf__C=10)
Pipeline(memory=None,
         steps=[('reduce_dim', PCA(copy=True, iterated_power='auto', ...)),
                ('clf', SVC(C=10, cache_size=200, class_weight=None, ...))],
         verbose=False)
```

This is particularly important for doing grid searches:

```
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = dict(reduce_dim_n_components=[2, 5, 10],
...                      clf_C=[0.1, 10, 100])
>>> grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

Individual steps may also be replaced as parameters, and non-final steps may be ignored by setting them to 'passthrough':

```
>>> from sklearn.linear_model import LogisticRegression
>>> param_grid = dict(reduce_dim=['passthrough', PCA(5), PCA(10)],
...                      clf=[SVC(), LogisticRegression()],
...                      clf_C=[0.1, 10, 100])
>>> grid_search = GridSearchCV(pipe, param_grid=param_grid)
```

The estimators of the pipeline can be retrieved by index:

```
>>> pipe[0]
PCA(copy=True, ...)
```

or by name:

```
>>> pipe['reduce_dim']
PCA(copy=True, ...)
```

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](#)
- [Selecting dimensionality reduction with Pipeline and GridSearchCV](#)

See also:

- [Tuning the hyper-parameters of an estimator](#)

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.

Caching transformers: avoid repeated computation

Fitting transformers may be computationally expensive. With its `memory` parameter set, `Pipeline` will cache each transformer after calling `fit`. This feature is used to avoid computing the fit transformers within a pipeline if the parameters and input data are identical. A typical example is the case of a grid search in which the transformers can be fitted only once and reused for each configuration.

The parameter `memory` is needed in order to cache the transformers. `memory` can be either a string containing the directory where to cache the transformers or a `joblib.Memory` object:

```
>>> from tempfile import mkdtemp
>>> from shutil import rmtree
>>> from sklearn.decomposition import PCA
>>> from sklearn.svm import SVC
>>> from sklearn.pipeline import Pipeline
>>> estimators = [('reduce_dim', PCA()), ('clf', SVC())]
>>> cachedir = mkdtemp()
>>> pipe = Pipeline(estimators, memory=cachedir)
>>> pipe
Pipeline(...,
          steps=[('reduce_dim', PCA(copy=True, ...)),
                 ('clf', SVC(C=1.0, ...))], verbose=False)
>>> # Clear the cache directory when you don't need it anymore
>>> rmtree(cachedir)
```

Warning: Side effect of caching transformers

Using a `Pipeline` without cache enabled, it is possible to inspect the original instance such as:

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
>>> pca1 = PCA()
>>> svm1 = SVC(gamma='scale')
>>> pipe = Pipeline([('reduce_dim', pca1), ('clf', svm1)])
>>> pipe.fit(digits.data, digits.target)
...
Pipeline(memory=None,
          steps=[('reduce_dim', PCA(...)), ('clf', SVC(...))],
          verbose=False)
>>> # The pca instance can be inspected directly
>>> print(pca1.components_)
[[ -1.77484909e-19 ... 4.07058917e-18]]
```

Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. In following example, accessing the PCA instance `pca2` will raise an `AttributeError` since `pca2` will be an unfitted transformer. Instead, use the attribute `named_steps` to inspect estimators within the pipeline:

```
>>> cachedir = mkdtemp()
>>> pca2 = PCA()
>>> svm2 = SVC(gamma='scale')
>>> cached_pipe = Pipeline([('reduce_dim', pca2), ('clf', svm2)],
...                         memory=cachedir)
>>> cached_pipe.fit(digits.data, digits.target)
...
Pipeline(memory=...,
         steps=[('reduce_dim', PCA(...)), ('clf', SVC(...))],
         verbose=False)
>>> print(cached_pipe.named_steps['reduce_dim'].components_)
...
[[[-1.77484909e-19 ... 4.07058917e-18]]]
>>> # Remove the cache directory
>>> rmtree(cachedir)
```

Examples:

- *Selecting dimensionality reduction with Pipeline and GridSearchCV*

Transforming target in regression

`TransformedTargetRegressor` transforms the targets `y` before fitting a regression model. The predictions are mapped back to the original space via an inverse transform. It takes as an argument the regressor that will be used for prediction, and the transformer that will be applied to the target variable:

```
>>> import numpy as np
>>> from sklearn.datasets import load_boston
>>> from sklearn.compose import TransformedTargetRegressor
>>> from sklearn.preprocessing import QuantileTransformer
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.model_selection import train_test_split
>>> boston = load_boston()
>>> X = boston.data
>>> y = boston.target
>>> transformer = QuantileTransformer(output_distribution='normal')
>>> regressor = LinearRegression()
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                     transformer=transformer)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {:.2f}'.format(regr.score(X_test, y_test)))
R2 score: 0.67
>>> raw_target_regr = LinearRegression().fit(X_train, y_train)
>>> print('R2 score: {:.2f}'.format(raw_target_regr.score(X_test, y_test)))
R2 score: 0.64
```

For simple transformations, instead of a Transformer object, a pair of functions can be passed, defining the transformation and its inverse mapping:

```
>>> def func(x):
...     return np.log(x)
>>> def inverse_func(x):
...     return np.exp(x)
```

Subsequently, the object is created as:

```
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                     func=func,
...                                     inverse_func=inverse_func)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {:.2f}'.format(regr.score(X_test, y_test)))
R2 score: 0.65
```

By default, the provided functions are checked at each fit to be the inverse of each other. However, it is possible to bypass this checking by setting `check_inverse` to `False`:

```
>>> def inverse_func(x):
...     return x
>>> regr = TransformedTargetRegressor(regressor=regressor,
...                                     func=func,
...                                     inverse_func=inverse_func,
...                                     check_inverse=False)
>>> regr.fit(X_train, y_train)
TransformedTargetRegressor(...)
>>> print('R2 score: {:.2f}'.format(regr.score(X_test, y_test)))
R2 score: -4.50
```

Note: The transformation can be triggered by setting either `transformer` or the pair of functions `func` and `inverse_func`. However, setting both options will raise an error.

Examples:

- *Effect of transforming the targets in regression model*

FeatureUnion: composite feature spaces

`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. The transformers are applied in parallel, and the feature matrices they output are concatenated side-by-side into a larger matrix.

When you want to apply different transformations to each field of the data, see the related class `sklearn.compose.ColumnTransformer` (see [user guide](#)).

`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 the caller's responsibility.)

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=None,
             transformer_list=[('linear_pca', PCA(copy=True, ...)),
                               ('kernel_pca', KernelPCA(alpha=1.0, ...))],
             transformer_weights=None, verbose=False)
```

Like pipelines, feature unions have a shorthand constructor called `make_union` that does not require explicit naming of the components.

Like Pipeline, individual steps may be replaced using `set_params`, and ignored by setting to 'drop':

```
>>> combined.set_params(kernel_pca='drop')
...
FeatureUnion(n_jobs=None,
             transformer_list=[('linear_pca', PCA(copy=True, ...)),
                               ('kernel_pca', 'drop')],
             transformer_weights=None, verbose=False)
```

Examples:

- Concatenating multiple feature extraction methods

ColumnTransformer for heterogeneous data

Warning: The `compose.ColumnTransformer` class is experimental and the API is subject to change.

Many datasets contain features of different types, say text, floats, and dates, where each type of feature requires separate preprocessing or feature extraction steps. Often it is easiest to preprocess data before applying scikit-learn methods, for example using `pandas`. Processing your data before passing it to scikit-learn might be problematic for one of the following reasons:

1. Incorporating statistics from test data into the preprocessors makes cross-validation scores unreliable (known as *data leakage*), for example in the case of scalers or imputing missing values.
2. You may want to include the parameters of the preprocessors in a `parameter search`.

The `ColumnTransformer` helps performing different transformations for different columns of the data, within a `Pipeline` that is safe from data leakage and that can be parametrized. `ColumnTransformer` works on arrays, sparse matrices, and `pandas` `DataFrames`.

To each column, a different transformation can be applied, such as preprocessing or a specific feature extraction method:

```
>>> import pandas as pd
>>> X = pd.DataFrame(
...     {'city': ['London', 'London', 'Paris', 'Sallisaw'],
...      'title': ["His Last Bow", "How Watson Learned the Trick",
...                "A Moveable Feast", "The Grapes of Wrath"],
...      'expert_rating': [5, 3, 4, 5],
...      'user_rating': [4, 5, 4, 3]})
```

For this data, we might want to encode the 'city' column as a categorical variable using `preprocessing.OneHotEncoder` but apply a `feature_extraction.text.CountVectorizer` to the 'title' column. As we might use multiple feature extraction methods on the same column, we give each transformer a unique name, say 'city_category' and 'title_bow'. By default, the remaining rating columns are ignored (`remainder='drop'`):

```
>>> from sklearn.compose import ColumnTransformer
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> from sklearn.preprocessing import OneHotEncoder
>>> column_trans = ColumnTransformer(
...     [('city_category', OneHotEncoder(dtype='int'), ['city']),
...      ('title_bow', CountVectorizer(), 'title')],
...     remainder='drop')

>>> column_trans.fit(X)
ColumnTransformer(n_jobs=None, remainder='drop', sparse_threshold=0.3,
                  transformer_weights=None,
                  transformers=...)

>>> column_trans.get_feature_names()
...
['city_category_London', 'city_category_Paris', 'city_category_Sallisaw',
 'title_bow_bow', 'title_bow_feast', 'title_bow_grapes', 'title_bow_his',
 'title_bow_how', 'title_bow_last', 'title_bow_learned', 'title_bow_moveable',
 'title_bow_of', 'title_bow_the', 'title_bow_trick', 'title_bow_watson',
 'title_bow_wrath']

>>> column_trans.transform(X).toarray()
...
array([[1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0],
       [0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1]]...)
```

In the above example, the `CountVectorizer` expects a 1D array as input and therefore the columns were specified as a string ('title'). However, `preprocessing.OneHotEncoder` as most of other transformers expects 2D data, therefore in that case you need to specify the column as a list of strings (['city']).

Apart from a scalar or a single item list, the column selection can be specified as a list of multiple items, an integer array, a slice, or a boolean mask. Strings can reference columns if the input is a DataFrame, integers are always interpreted as the positional columns.

We can keep the remaining rating columns by setting `remainder='passthrough'`. The values are appended to the end of the transformation:

```
>>> column_trans = ColumnTransformer(
...     [('city_category', OneHotEncoder(dtype='int'), ['city']),
```

```

...      ('title_bow', CountVectorizer(), 'title')],
...      remainder='passthrough')

>>> column_trans.fit_transform(X)
...
array([[1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 4],
       [1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 3, 5],
       [0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 4, 4],
       [0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 5, 3]]...)

```

The `remainder` parameter can be set to an estimator to transform the remaining rating columns. The transformed values are appended to the end of the transformation:

```

>>> from sklearn.preprocessing import MinMaxScaler
>>> column_trans = ColumnTransformer(
...     [('city_category', OneHotEncoder(), ['city']),
...     ('title_bow', CountVectorizer(), 'title')],
...     remainder=MinMaxScaler())

>>> column_trans.fit_transform(X)[:, -2:]
...
array([[1., 0.5],
       [0., 1.],
       [0.5, 0.5],
       [1., 0.]])

```

The `make_column_transformer` function is available to more easily create a `ColumnTransformer` object. Specifically, the names will be given automatically. The equivalent for the above example would be:

```

>>> from sklearn.compose import make_column_transformer
>>> column_trans = make_column_transformer(
...     (OneHotEncoder(), ['city']),
...     (CountVectorizer(), 'title'),
...     remainder=MinMaxScaler())
>>> column_trans
ColumnTransformer(n_jobs=None, remainder=MinMaxScaler(copy=True, ...),
                  sparse_threshold=0.3,
                  transformer_weights=None,
                  transformers=[('onehotencoder', ...)])

```

Examples:

- [Column Transformer with Heterogeneous Data Sources](#)
- [Column Transformer with Mixed Types](#)

3.5.2 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.

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 Francisco', '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 Francisco', '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`.

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. This mechanism is enabled by default with `alternate_sign=True` and is particularly useful for small hash table sizes (`n_features < 10000`). For large hash table sizes, it can be disabled, to allow the output to be passed to estimators like `sklearn.naive_bayes.MultinomialNB` or `sklearn.feature_selection.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.

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](#).

Text feature extraction

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.

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.

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()
>>> vectorizer
CountVectorizer(analyzer='word', binary=False, 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+\\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, 0, 0, 0, 0, 0, 1, 1, 0, 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, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 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]...)
```

Using stop words

Stop words are words like “and”, “the”, “him”, which are presumed to be uninformative in representing the content of a text, and which may be removed to avoid them being construed as signal for prediction. Sometimes, however, similar words are useful for prediction, such as in classifying writing style or personality.

There are several known issues in our provided ‘english’ stop word list. See [\[NQY18\]](#).

Please take care in choosing a stop word list. Popular stop word lists may include words that are highly informative to some tasks, such as *computer*.

You should also make sure that the stop word list has had the same preprocessing and tokenization applied as the one used in the vectorizer. The word *we've* is split into *we* and *ve* by CountVectorizer’s default tokenizer, so if *we've* is in `stop_words`, but *ve* is not, *ve* will be retained from *we've* in transformed text. Our vectorizers will try to identify and warn about some kinds of inconsistencies.

References

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**: $\text{tf-idf}(t,d) = \text{tf}(t,d) \times \text{idf}(t)$.

Using the `TfidfTransformer`’s default settings, `TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False)` the term frequency, the number of times a term occurs in a given document, is multiplied with idf component, which is computed as

$$\text{idf}(t) = \log \frac{1+n}{1+\text{df}(t)} + 1,$$

where n is the total number of documents in the document set, and $\text{df}(t)$ is the number of documents in the document set that contain term t . The resulting tf-idf vectors are then normalized by the Euclidean norm:

$$v_{\text{norm}} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}}.$$

This was 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.

The following sections contain further explanations and examples that illustrate how the tf-idfs are computed exactly and how the tf-idfs computed in scikit-learn’s `TfidfTransformer` and `TfidfVectorizer` differ slightly from the standard textbook notation that defines the idf as

$$\text{idf}(t) = \log \frac{n}{1+\text{df}(t)}.$$

In the `TfidfTransformer` and `TfidfVectorizer` with `smooth_idf=False`, the “1” count is added to the idf instead of the idf’s denominator:

$$\text{idf}(t) = \log \frac{n}{\text{df}(t)} + 1$$

This normalization is implemented by the `TfidfTransformer` class:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> transformer = TfidfTransformer(smooth_idf=False)
>>> transformer
TfidfTransformer(norm='l2', smooth_idf=False, 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([[0.81940995, 0.           , 0.57320793],
       [1.          , 0.           , 0.           ],
       [1.          , 0.           , 0.           ],
       [1.          , 0.           , 0.           ],
       [0.47330339, 0.88089948, 0.           ],
       [0.58149261, 0.           , 0.81355169]])
```

Each row is normalized to have unit Euclidean norm:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}}$$

For example, we can compute the tf-idf of the first term in the first document in the `counts` array as follows:

$$n = 6$$

$$\text{df}(t)_{\text{term1}} = 6$$

$$\text{idf}(t)_{\text{term1}} = \log \frac{n}{\text{df}(t)} + 1 = \log(1) + 1 = 1$$

$$\text{tf-idf}_{\text{term1}} = \text{tf} \times \text{idf} = 3 \times 1 = 3$$

Now, if we repeat this computation for the remaining 2 terms in the document, we get

$$\text{tf-idf}_{\text{term2}} = 0 \times (\log(6/1) + 1) = 0$$

$$\text{tf-idf}_{\text{term3}} = 1 \times (\log(6/2) + 1) \approx 2.0986$$

and the vector of raw tf-idfs:

$$\text{tf-idf}_{\text{raw}} = [3, 0, 2.0986].$$

Then, applying the Euclidean (L2) norm, we obtain the following tf-idfs for document 1:

$$\frac{[3, 0, 2.0986]}{\sqrt{(3^2 + 0^2 + 2.0986^2)}} = [0.819, 0, 0.573].$$

Furthermore, the default parameter `smooth_idf=True` adds “1” to the numerator and denominator as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions:

$$\text{idf}(t) = \log \frac{1+n}{1+\text{df}(t)} + 1$$

Using this modification, the tf-idf of the third term in document 1 changes to 1.8473:

$$\text{tf-idf}_{\text{term}3} = 1 \times \log(7/3) + 1 \approx 1.8473$$

And the L2-normalized tf-idf changes to

$$\frac{[3, 0, 1.8473]}{\sqrt{(3^2+0^2+1.8473^2)}} = [0.8515, 0, 0.5243]:$$

```
>>> transformer = TfidfTransformer()
>>> transformer.fit_transform(counts).toarray()
array([[0.85151335, 0.           , 0.52433293],
       [1.           , 0.           , 0.           ],
       [1.           , 0.           , 0.           ],
       [1.           , 0.           , 0.           ],
       [0.55422893, 0.83236428, 0.           ],
       [0.63035731, 0.           , 0.77630514]])
```

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()
>>> 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*

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\x001\x00\xfc\x00g\x00e\x001\x00n\x00\x00\x00d\x00e\x00s\x00 \x00G\x00e\x00s\x00a\x00n\x00g\x00e\x00s\x00,\x00\x00H\x00e\x00r\x00z\x001\x00i\x00e\x00b\x00c\x00h\x00e\x00n\x00,\x00\x00t\x00r\x00a\x00g\x00 \x00i\x00c\x00h\x00 \x00d\x00i\x00c\x00h\x00\x00\x00f\x00o\x00r\x00t\x00\x00"
>>> 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](#).

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)*:

- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*

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))
>>> 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))
>>> 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))
>>> 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.

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.

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](#).

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.)

The following example will, for instance, transform some British spelling to American spelling:

```
>>> import re
>>> def to_british(tokens):
...     for t in tokens:
...         t = re.sub(r"(...)our$", r"\1or", t)
...         t = re.sub(r"([bt])re$", r"\1er", t)
...         t = re.sub(r"([iy])s(e$|ing|ation)", r"\1z\2", t)
...         t = re.sub(r"ogue$", "og", t)
...         yield t
...
>>> class CustomVectorizer(CountVectorizer):
...     def build_tokenizer(self):
...         tokenize = super().build_tokenizer()
...         return lambda doc: list(to_british(tokenize(doc)))
...
>>> print(CustomVectorizer().build_analyzer()(u"color colour"))
['color', 'color']
```

for other styles of preprocessing; examples include stemming, lemmatization, or normalizing numerical tokens, with the latter illustrated in:

- [Biclustering documents with the Spectral Co-clustering algorithm](#)

Customizing the vectorizer can also be useful when handling Asian languages that do not use an explicit word separator such as whitespace.

Image feature extraction

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)
```

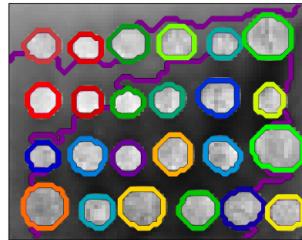
The `PatchExtractor` class works in the same way as `extract_patches_2d`, only it supports multiple images as input. It is implemented as an estimator, so it can be used in pipelines. See:

```
>>> 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)
```

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 an image of coins](#)
 - [Spectral clustering for image segmentation](#)
 - [Feature agglomeration vs. univariate selection](#)
-

3.5.3 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.

In general, learning algorithms benefit from standardization of the data set. If some outliers are present in the set, robust scalers or transformers are more appropriate. The behaviors of the different scalers, transformers, and normalizers on a dataset containing marginal outliers is highlighted in [Compare the effect of different scalers on data with outliers](#).

Standardization, or mean removal and variance scaling

Standardization of datasets is a **common requirement for many machine learning estimators** implemented in scikit-learn; they might behave badly if the individual features 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 than 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_train = np.array([[ 1., -1.,  2.],
...                      [ 2.,  0.,  0.],
...                      [ 0.,  1., -1.]])
>>> X_scaled = preprocessing.scale(X_train)

>>> 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_train)
>>> scaler
StandardScaler(copy=True, with_mean=True, with_std=True)

>>> scaler.mean_
array([1. ..., 0. ..., 0.33...])

>>> scaler.scale_
array([0.81..., 0.81..., 1.24...])

>>> scaler.transform(X_train)
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:

```
>>> X_test = [[-1., 1., 0.]]
>>> scaler.transform(X_test)
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`.

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, or so that the maximum absolute value of each feature is scaled to unit size. This can be achieved using `MinMaxScaler` or `MaxAbsScaler`, respectively.

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        , 0.33333333])
...
>>> min_max_scaler.min_
array([0.        , 0.5        , 0.33333333])
```

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
```

`MaxAbsScaler` works in a very similar fashion, but scales in a way that the training data lies within the range [-1, 1] by dividing through the largest maximum value in each feature. It is meant for data that is already centered at zero or sparse data.

Here is how to use the toy data from the previous example with this scaler:

```
>>> X_train = np.array([[ 1., -1.,  2.],
...                      [ 2.,  0.,  0.],
...                      [ 0.,  1., -1.]])
...
>>> max_abs_scaler = preprocessing.MaxAbsScaler()
>>> X_train_maxabs = max_abs_scaler.fit_transform(X_train)
>>> X_train_maxabs
# doctest +NORMALIZE_WHITESPACE^
array([[ 0.5, -1. ,  1. ],
       [ 1. ,  0. ,  0. ],
       [ 0. ,  1. , -0.5]])
>>> X_test = np.array([[-3., -1.,  4.]])
>>> X_test_maxabs = max_abs_scaler.transform(X_test)
>>> X_test_maxabs
array([-1.5, -1. ,  2. ])
>>> max_abs_scaler.scale_
array([2.,  1.,  2.])
```

As with `scale`, the module further provides convenience functions `minmax_scale` and `maxabs_scale` if you don't want to create an object.

Scaling sparse data

Centering sparse data would destroy the sparseness structure in the data, and thus rarely is a sensible thing to do. However, it can make sense to scale sparse inputs, especially if features are on different scales.

`MaxAbsScaler` and `maxabs_scale` were specifically designed for scaling sparse data, and are the recommended way to go about this. However, `scale` and `StandardScaler` can accept `scipy.sparse` matrices as input, as long as `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. `RobustScaler` cannot be fitted to sparse inputs, but you can use the `transform` method on sparse inputs.

Note that the scalers accept both Compressed Sparse Rows and Compressed Sparse Columns format (see `scipy.sparse.csr_matrix` and `scipy.sparse.csc_matrix`). Any other sparse input will be **converted to the Compressed Sparse Rows representation**. To avoid unnecessary memory copies, it is recommended to choose the CSR or CSC representation upstream.

Finally, if the centered data is expected to be small enough, explicitly converting the input to an array using the `toarray` method of sparse matrices is another option.

Scaling data with outliers

If your data contains many outliers, scaling using the mean and variance of the data is likely to not work very well. In these cases, you can use `robust_scale` and `RobustScaler` as drop-in replacements instead. They use more robust estimates for the center and range of your data.

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` with `whiten=True` to further remove the linear correlation across features.

Scaling a 1D array

All above functions (i.e. `scale`, `minmax_scale`, `maxabs_scale`, and `robust_scale`) accept 1D array which can be useful in some specific case.

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 ϕ , a `KernelCenterer` can transform the kernel matrix so that it contains inner products in the feature space defined by ϕ followed by removal of the mean in that space.

Non-linear transformation

Two types of transformations are available: quantile transforms and power transforms. Both quantile and power transforms are based on monotonic transformations of the features and thus preserve the rank of the values along each feature.

Quantile transforms put all features into the same desired distribution based on the formula $G^{-1}(F(X))$ where F is the cumulative distribution function of the feature and G^{-1} the [quantile function](#) of the desired output distribution G . This formula is using the two following facts: (i) if X is a random variable with a continuous cumulative distribution function F then $F(X)$ is uniformly distributed on $[0, 1]$; (ii) if U is a random variable with uniform distribution on $[0, 1]$ then $G^{-1}(U)$ has distribution G . By performing a rank transformation, a quantile transform smooths out unusual distributions and is less influenced by outliers than scaling methods. It does, however, distort correlations and distances within and across features.

Power transforms are a family of parametric transformations that aim to map data from any distribution to as close to a Gaussian distribution.

Mapping to a Uniform distribution

`QuantileTransformer` and `quantile_transform` provide a non-parametric transformation to map the data to a uniform distribution with values between 0 and 1:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
>>> quantile_transformer = preprocessing.QuantileTransformer(random_state=0)
>>> X_train_trans = quantile_transformer.fit_transform(X_train)
>>> X_test_trans = quantile_transformer.transform(X_test)
>>> np.percentile(X_train[:, 0], [0, 25, 50, 75, 100])
array([ 4.3,  5.1,  5.8,  6.5,  7.9])
```

This feature corresponds to the sepal length in cm. Once the quantile transformation applied, those landmarks approach closely the percentiles previously defined:

```
>>> np.percentile(X_train_trans[:, 0], [0, 25, 50, 75, 100])
...
array([ 0.00...,  0.24...,  0.49...,  0.73...,  0.99...])
```

This can be confirmed on a independent testing set with similar remarks:

```
>>> np.percentile(X_test[:, 0], [0, 25, 50, 75, 100])
...
array([ 4.4,  5.125,  5.75,  6.175,  7.3])
>>> np.percentile(X_test_trans[:, 0], [0, 25, 50, 75, 100])
...
array([ 0.01...,  0.25...,  0.46...,  0.60...,  0.94...])
```

Mapping to a Gaussian distribution

In many modeling scenarios, normality of the features in a dataset is desirable. Power transforms are a family of parametric, monotonic transformations that aim to map data from any distribution to as close to a Gaussian distribution as possible in order to stabilize variance and minimize skewness.

`PowerTransformer` currently provides two such power transformations, the Yeo-Johnson transform and the Box-Cox transform.

The Yeo-Johnson transform is given by:

$$x_i^{(\lambda)} = \begin{cases} [(x_i + 1)^\lambda - 1]/\lambda & \text{if } \lambda \neq 0, x_i \geq 0, \\ \ln(x_i) + 1 & \text{if } \lambda = 0, x_i \geq 0 \\ -[(-x_i + 1)^{2-\lambda} - 1]/(2 - \lambda) & \text{if } \lambda \neq 2, x_i < 0, \\ -\ln(-x_i + 1) & \text{if } \lambda = 2, x_i < 0 \end{cases}$$

while the Box-Cox transform is given by:

$$x_i^{(\lambda)} = \begin{cases} \frac{x_i^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln(x_i) & \text{if } \lambda = 0, \end{cases}$$

Box-Cox can only be applied to strictly positive data. In both methods, the transformation is parameterized by λ , which is determined through maximum likelihood estimation. Here is an example of using Box-Cox to map samples drawn from a lognormal distribution to a normal distribution:

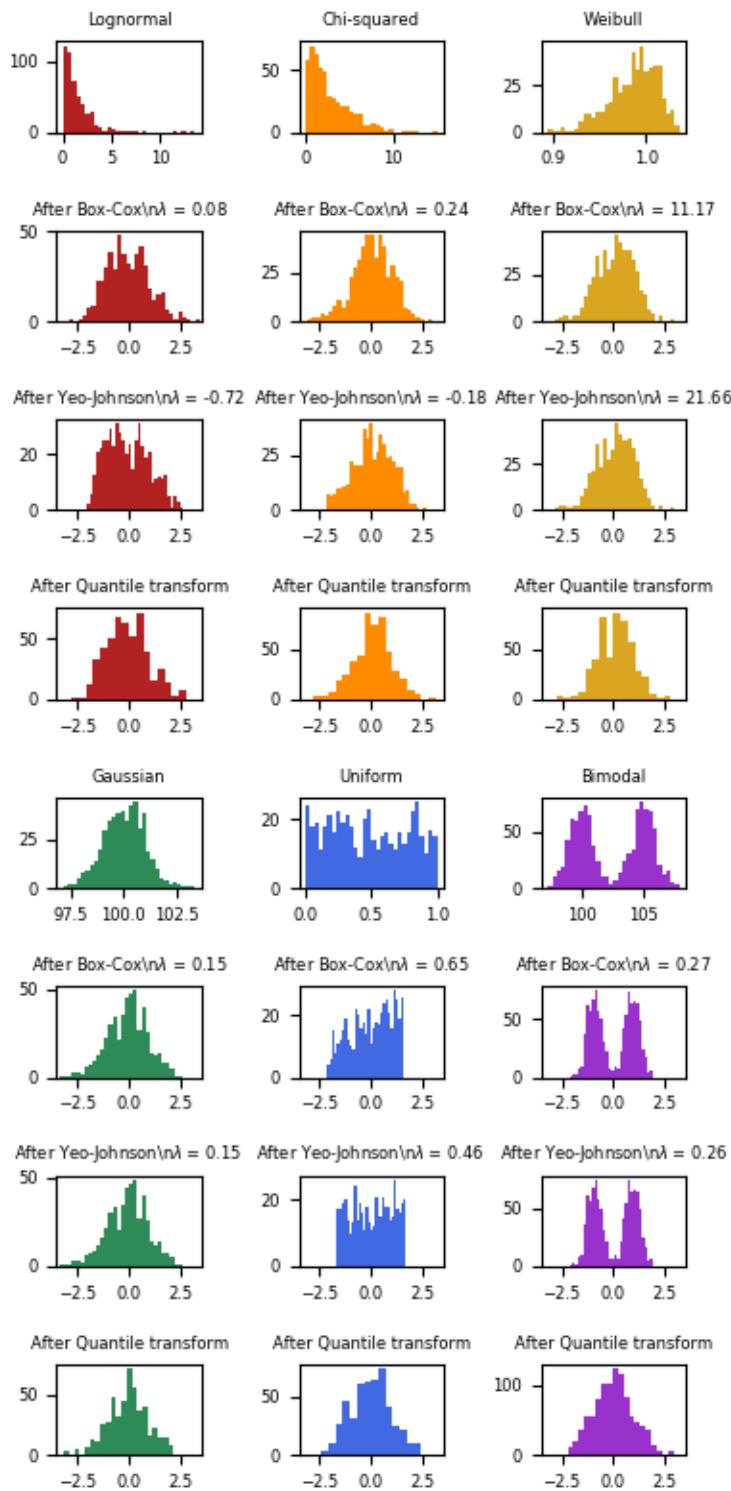
```
>>> pt = preprocessing.PowerTransformer(method='box-cox', standardize=False)
>>> X_lognormal = np.random.RandomState(616).lognormal(size=(3, 3))
>>> X_lognormal
array([[1.28..., 1.18..., 0.84...],
       [0.94..., 1.60..., 0.38...],
       [1.35..., 0.21..., 1.09...]])
>>> pt.fit_transform(X_lognormal)
array([[ 0.49...,  0.17..., -0.15...],
       [-0.05...,  0.58..., -0.57...],
       [ 0.69..., -0.84...,  0.10...]])
```

While the above example sets the `standardize` option to `False`, `PowerTransformer` will apply zero-mean, unit-variance normalization to the transformed output by default.

Below are examples of Box-Cox and Yeo-Johnson applied to various probability distributions. Note that when applied to certain distributions, the power transforms achieve very Gaussian-like results, but with others, they are ineffective. This highlights the importance of visualizing the data before and after transformation.

It is also possible to map data to a normal distribution using `QuantileTransformer` by setting `output_distribution='normal'`. Using the earlier example with the iris dataset:

```
>>> quantile_transformer = preprocessing.QuantileTransformer(
...     output_distribution='normal', random_state=0)
>>> X_trans = quantile_transformer.fit_transform(X)
>>> quantile_transformer.quantiles_
array([[4.3, 2., 1., 0.1],
       [4.4, 2.2, 1.1, 0.1],
       [4.4, 2.2, 1.2, 0.1],
       ...,
```



```
[7.7, 4.1, 6.7, 2.5],
[7.7, 4.2, 6.7, 2.5],
[7.9, 4.4, 6.9, 2.5]])
```

Thus the median of the input becomes the mean of the output, centered at 0. The normal output is clipped so that the input's minimum and maximum — corresponding to the 1e-7 and 1 - 1e-7 quantiles respectively — do not become infinite under the transformation.

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 11 or 12 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.

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].

To convert categorical features to such integer codes, we can use the `OrdinalEncoder`. This estimator transforms each categorical feature to one new feature of integers (0 to n_categories - 1):

```
>>> enc = preprocessing.OrdinalEncoder()
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox',
   <...>]]
>>> enc.fit(X)
OrdinalEncoder(categories='auto', dtype=<... 'numpy.float64'>)
>>> enc.transform([['female', 'from US', 'uses Safari']])
array([[0., 1., 1.]])
```

Such integer representation can, however, not be used directly with all 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).

Another possibility to convert categorical features to features that can be used with scikit-learn estimators is to use a one-of-K, also known as one-hot or dummy encoding. This type of encoding can be obtained with the `OneHotEncoder`, which transforms each categorical feature with n_categories possible values into n_categories binary features, with one of them 1, and all others 0.

Continuing the example above:

```
>>> enc = preprocessing.OneHotEncoder()
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox',
   <...>]]
>>> enc.fit(X)
OneHotEncoder(categorical_features=None, categories=None, drop=None,
              dtype=<... 'numpy.float64'>, handle_unknown='error',
              n_values=None, sparse=True)
>>> enc.transform([['female', 'from US', 'uses Safari'],
   ...           ['male', 'from Europe', 'uses Safari']]).toarray()
array([[1., 0., 0., 1., 0., 1.],
       [0., 1., 1., 0., 0., 1.]])
```

By default, the values each feature can take is inferred automatically from the dataset and can be found in the `categories_` attribute:

```
>>> enc.categories_
[array(['female', 'male'], dtype=object), array(['from Europe', 'from US'],  

   <...>),
 array(['uses Firefox', 'uses Safari'], dtype=object)]
```

It is possible to specify this explicitly using the parameter `categories`. There are two genders, four possible continents and four web browsers in our dataset:

```
>>> genders = ['female', 'male']
>>> locations = ['from Africa', 'from Asia', 'from Europe', 'from US']
>>> browsers = ['uses Chrome', 'uses Firefox', 'uses IE', 'uses Safari']
>>> enc = preprocessing.OneHotEncoder(categories=[genders, locations, browsers])
>>> # Note that for there are missing categorical values for the 2nd and 3rd
   # feature
```

```
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox'
   <...>]
>>> enc.fit(X)
OneHotEncoder(categorical_features=None,
               categories=[...], drop=None,
               dtype=<... 'numpy.float64'>, handle_unknown='error',
               n_values=None, sparse=True)
>>> enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
array([[1., 0., 0., 1., 0., 0., 1., 0., 0., 0.]])
```

If there is a possibility that the training data might have missing categorical features, it can often be better to specify `handle_unknown='ignore'` instead of setting the `categories` manually as above. When `handle_unknown='ignore'` is specified and unknown categories are encountered during transform, no error will be raised but the resulting one-hot encoded columns for this feature will be all zeros (`handle_unknown='ignore'` is only supported for one-hot encoding):

```
>>> enc = preprocessing.OneHotEncoder(handle_unknown='ignore')
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox'
   <...>]
>>> enc.fit(X)
OneHotEncoder(categorical_features=None, categories=None, drop=None,
               dtype=<... 'numpy.float64'>, handle_unknown='ignore',
               n_values=None, sparse=True)
>>> enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
array([[1., 0., 0., 0., 0., 0.]])
```

It is also possible to encode each column into `n_categories - 1` columns instead of `n_categories` columns by using the `drop` parameter. This parameter allows the user to specify a category for each feature to be dropped. This is useful to avoid co-linearity in the input matrix in some classifiers. Such functionality is useful, for example, when using non-regularized regression ([LinearRegression](#)), since co-linearity would cause the covariance matrix to be non-invertible. When this parameter is not `None`, `handle_unknown` must be set to `error`:

```
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox'
   <...>]
>>> drop_enc = preprocessing.OneHotEncoder(drop='first').fit(X)
>>> drop_enc.categories_
[array(['female', 'male'], dtype=object), array(['from Europe', 'from US'],
   <...>),
   array(['uses Firefox', 'uses Safari'], dtype=object)]
>>> drop_enc.transform(X).toarray()
array([[1., 1., 1.,
       0., 0., 0.]])
```

See [Loading features from dicts](#) for categorical features that are represented as a dict, not as scalars.

Discretization

Discretization (otherwise known as quantization or binning) provides a way to partition continuous features into discrete values. Certain datasets with continuous features may benefit from discretization, because discretization can transform the dataset of continuous attributes to one with only nominal attributes.

One-hot encoded discretized features can make a model more expressive, while maintaining interpretability. For instance, pre-processing with a discretizer can introduce nonlinearity to linear models.

K-bins discretization

KBinsDiscretizer discretizes features into k bins:

```
>>> X = np.array([[ -3.,  5., 15 ],
...                 [  0.,  6., 14 ],
...                 [  6.,  3., 11 ]])
>>> est = preprocessing.KBinsDiscretizer(n_bins=[3, 2, 2], encode='ordinal').fit(X)
```

By default the output is one-hot encoded into a sparse matrix (See [Encoding categorical features](#)) and this can be configured with the `encode` parameter. For each feature, the bin edges are computed during `fit` and together with the number of bins, they will define the intervals. Therefore, for the current example, these intervals are defined as:

- feature 1: $[-\infty, -1), [-1, 2), [2, \infty)$
- feature 2: $[-\infty, 5), [5, \infty)$
- feature 3: $[-\infty, 14), [14, \infty)$

Based on these bin intervals, `X` is transformed as follows:

```
>>> est.transform(X)
array([[ 0.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 2.,  0.,  0.]])
```

The resulting dataset contains ordinal attributes which can be further used in a [`sklearn.pipeline.Pipeline`](#).

Discretization is similar to constructing histograms for continuous data. However, histograms focus on counting features which fall into particular bins, whereas discretization focuses on assigning feature values to these bins.

KBinsDiscretizer implements different binning strategies, which can be selected with the `strategy` parameter. The ‘uniform’ strategy uses constant-width bins. The ‘quantile’ strategy uses the quantiles values to have equally populated bins in each feature. The ‘kmeans’ strategy defines bins based on a k-means clustering procedure performed on each feature independently.

Examples:

- [Using KBinsDiscretizer to discretize continuous features](#)
- [Feature discretization](#)
- [Demonstrating the different strategies of KBinsDiscretizer](#)

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.

Note that the `Binarizer` is similar to the `KBinsDiscretizer` when `k = 2`, and when the bin edge is at the value `threshold`.

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.

Imputation of missing values

Tools for imputing missing values are discussed at [Imputation of missing values](#).

Generating polynomial features

Often it's useful to add complexity to the model by considering nonlinear features of the input data. A simple and common method to use is polynomial features, which can get features' high-order and interaction terms. It is implemented in `PolynomialFeatures`:

```
>>> import numpy as np
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> poly = PolynomialFeatures(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 $(1, X_1, X_2, X_1^2, X_1X_2, X_2^2)$.

In some cases, only interaction terms among features are required, and it can be gotten with the setting `interaction_only=True`:

```
>>> X = np.arange(9).reshape(3, 3)
>>> X
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> poly = PolynomialFeatures(degree=3, interaction_only=True)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  2.,  0.,  0.,  2.,  0.],
       [ 1.,  3.,  4.,  5.,  12., 15., 20., 60.],
       [ 1.,  6.,  7.,  8.,  42., 48., 56., 336.]])
```

The features of X have been transformed from (X_1, X_2, X_3) to $(1, X_1, X_2, X_3, X_1X_2, X_1X_3, X_2X_3, X_1X_2X_3)$.

Note that polynomial features are used implicitly in kernel methods (e.g., `sklearn.svm.SVC`, `sklearn.decomposition.KernelPCA`) when using polynomial *Kernel functions*.

See *Polynomial interpolation* for Ridge regression using created polynomial features.

Custom transformers

Often, you will want to convert an existing Python function into a transformer to assist in data cleaning or processing. You can implement a transformer from an arbitrary function with `FunctionTransformer`. For example, to build a transformer that applies a log transformation in a pipeline, do:

```
>>> import numpy as np
>>> from sklearn.preprocessing import FunctionTransformer
>>> transformer = FunctionTransformer(np.log1p, validate=True)
>>> X = np.array([[0, 1], [2, 3]])
>>> transformer.transform(X)
array([[0.        , 0.69314718],
       [1.09861229, 1.38629436]])
```

You can ensure that `func` and `inverse_func` are the inverse of each other by setting `check_inverse=True` and calling `fit` before `transform`. Please note that a warning is raised and can be turned into an error with a `filterwarnings`:

```
>>> import warnings
>>> warnings.filterwarnings("error", message=".*check_inverse.*",
...                           category=UserWarning, append=False)
```

For a full code example that demonstrates using a `FunctionTransformer` to do custom feature selection, see *Using FunctionTransformer to select columns*

3.5.4 Imputation of missing values

For various reasons, many real world datasets contain missing values, often encoded as blanks, NaNs or other place-holders. 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. See the [Glossary of Common Terms and API Elements](#) entry on imputation.

Univariate vs. Multivariate Imputation

One type of imputation algorithm is univariate, which imputes values in the i -th feature dimension using only non-missing values in that feature dimension (e.g. `impute.SimpleImputer`). By contrast, multivariate imputation algorithms use the entire set of available feature dimensions to estimate the missing values (e.g. `impute.IterativeImputer`).

Univariate feature imputation

The `SimpleImputer` class provides basic strategies for imputing missing values. Missing values can be imputed with a provided constant value, or using the statistics (mean, median or most frequent) of each 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.impute import SimpleImputer
>>> imp = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imp.fit([[1, 2], [np.nan, 3], [7, 6]])
SimpleImputer(add_indicator=False, copy=True, fill_value=None,
              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 `SimpleImputer` class also supports sparse matrices:

```
>>> import scipy.sparse as sp
>>> X = sp.csc_matrix([[1, 2], [0, -1], [8, 4]])
>>> imp = SimpleImputer(missing_values=-1, strategy='mean')
>>> imp.fit(X)
SimpleImputer(add_indicator=False, copy=True, fill_value=None,
              missing_values=-1, strategy='mean', verbose=0)
>>> X_test = sp.csc_matrix([[-1, 2], [6, -1], [7, 6]])
>>> print(imp.transform(X_test).toarray())
[[3. 2.]
 [6. 3.]
 [7. 6.]]
```

Note that this format is not meant to be used to implicitly store missing values in the matrix because it would densify it at transform time. Missing values encoded by 0 must be used with dense input.

The `SimpleImputer` class also supports categorical data represented as string values or pandas `categoricals` when using the '`most_frequent`' or '`constant`' strategy:

```
>>> import pandas as pd
>>> df = pd.DataFrame([["a", "x"],
>>>                   [np.nan, "y"],
>>>                   ...])
```

```
...
        ["a", np.nan],
...
        ["b", "y"]], dtype="category")
...
>>> imp = SimpleImputer(strategy="most_frequent")
>>> print(imp.fit_transform(df))
[['a' 'x']
 ['a' 'y']
 ['a' 'y']
 ['b' 'y']]
```

Multivariate feature imputation

A more sophisticated approach is to use the `IterativeImputer` class, which models each feature with missing values as a function of other features, and uses that estimate for imputation. It does so in an iterated round-robin fashion: at each step, a feature column is designated as output `y` and the other feature columns are treated as inputs `X`. A regressor is fit on (X, y) for known `y`. Then, the regressor is used to predict the missing values of `y`. This is done for each feature in an iterative fashion, and then is repeated for `max_iter` imputation rounds. The results of the final imputation round are returned.

Note: This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import `enable_iterative_imputer`.

```
>>> import numpy as np
>>> from sklearn.experimental import enable_iterative_imputer
>>> from sklearn.impute import IterativeImputer
>>> imp = IterativeImputer(max_iter=10, random_state=0)
>>> imp.fit([[1, 2], [3, 6], [4, 8], [np.nan, 3], [7, np.nan]])
IterativeImputer(add_indicator=False, estimator=None,
                  imputation_order='ascending', initial_strategy='mean',
                  max_iter=10, max_value=None, min_value=None,
                  missing_values=np.nan, n_nearest_features=None,
                  random_state=0, sample_posterior=False, tol=0.001,
                  verbose=0)
>>> X_test = [[np.nan, 2], [6, np.nan], [np.nan, 6]]
>>> # the model learns that the second feature is double the first
>>> print(np.round(imp.transform(X_test)))
[[ 1.  2.]
 [ 6. 12.]
 [ 3.  6.]]
```

Both `SimpleImputer` and `IterativeImputer` 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](#).

Flexibility of IterativeImputer

There are many well-established imputation packages in the R data science ecosystem: Amelia, mi, mice, missForest, etc. missForest is popular, and turns out to be a particular instance of different sequential imputation algorithms that can all be implemented with `IterativeImputer` by passing in different regressors to be used for predicting missing feature values. In the case of missForest, this regressor is a Random Forest. See [sphx_glr_auto_examples_plot_iterative_imputer_variants_comparison.py](#).

Multiple vs. Single Imputation

In the statistics community, it is common practice to perform multiple imputations, generating, for example, m separate imputations for a single feature matrix. Each of these m imputations is then put through the subsequent analysis pipeline (e.g. feature engineering, clustering, regression, classification). The m final analysis results (e.g. held-out validation errors) allow the data scientist to obtain understanding of how analytic results may differ as a consequence of the inherent uncertainty caused by the missing values. The above practice is called multiple imputation.

Our implementation of `IterativeImputer` was inspired by the R MICE package (Multivariate Imputation by Chained Equations)¹, but differs from it by returning a single imputation instead of multiple imputations. However, `IterativeImputer` can also be used for multiple imputations by applying it repeatedly to the same dataset with different random seeds when `sample_posterior=True`. See², chapter 4 for more discussion on multiple vs. single imputations.

It is still an open problem as to how useful single vs. multiple imputation is in the context of prediction and classification when the user is not interested in measuring uncertainty due to missing values.

Note that a call to the `transform` method of `IterativeImputer` is not allowed to change the number of samples. Therefore multiple imputations cannot be achieved by a single call to `transform`.

References

Marking imputed values

The `MissingIndicator` transformer is useful to transform a dataset into corresponding binary matrix indicating the presence of missing values in the dataset. This transformation is useful in conjunction with imputation. When using imputation, preserving the information about which values had been missing can be informative.

`NaN` is usually used as the placeholder for missing values. However, it enforces the data type to be float. The parameter `missing_values` allows to specify other placeholder such as integer. In the following example, we will use `-1` as missing values:

```
>>> from sklearn.impute import MissingIndicator
>>> X = np.array([[-1, -1, 1, 3],
...                 [4, -1, 0, -1],
...                 [8, -1, 1, 0]])
>>> indicator = MissingIndicator(missing_values=-1)
>>> mask_missing_values_only = indicator.fit_transform(X)
>>> mask_missing_values_only
array([[ True,  True, False],
       [False,  True,  True],
       [False,  True, False]])
```

The `features` parameter is used to choose the features for which the mask is constructed. By default, it is '`missing-only`' which returns the imputer mask of the features containing missing values at `fit` time:

```
>>> indicator.features_
array([0, 1, 3])
```

The `features` parameter can be set to '`all`' to returned all features whether or not they contain missing values:

```
>>> indicator = MissingIndicator(missing_values=-1, features="all")
>>> mask_all = indicator.fit_transform(X)
```

¹ Stef van Buuren, Karin Groothuis-Oudshoorn (2011). “mice: Multivariate Imputation by Chained Equations in R”. Journal of Statistical Software 45: 1-67.

² Roderick J A Little and Donald B Rubin (1986). “Statistical Analysis with Missing Data”. John Wiley & Sons, Inc., New York, NY, USA.

```
>>> mask_all
array([[ True,  True, False, False],
       [False,  True, False,  True],
       [False,  True, False, False]])
>>> indicator.features_
array([0, 1, 2, 3])
```

When using the `MissingIndicator` in a Pipeline, be sure to use the FeatureUnion or ColumnTransformer to add the indicator features to the regular features. First we obtain the iris dataset, and add some missing values to it.

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.impute import SimpleImputer, MissingIndicator
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.pipeline import FeatureUnion, make_pipeline
>>> from sklearn.tree import DecisionTreeClassifier
>>> X, y = load_iris(return_X_y=True)
>>> mask = np.random.randint(0, 2, size=X.shape).astype(np.bool)
>>> X[mask] = np.nan
>>> X_train, X_test, y_train, _ = train_test_split(X, y, test_size=100,
...                                                 random_state=0)
```

Now we create a FeatureUnion. All features will be imputed using `SimpleImputer`, in order to enable classifiers to work with this data. Additionally, it adds the the indicator variables from `MissingIndicator`.

```
>>> transformer = FeatureUnion(
...     transformer_list=[
...         ('features', SimpleImputer(strategy='mean')),
...         ('indicators', MissingIndicator())])
>>> transformer = transformer.fit(X_train, y_train)
>>> results = transformer.transform(X_test)
>>> results.shape
(100, 8)
```

Of course, we cannot use the transformer to make any predictions. We should wrap this in a Pipeline with a classifier (e.g., a `DecisionTreeClassifier`) to be able to make predictions.

```
>>> clf = make_pipeline(transformer, DecisionTreeClassifier())
>>> clf = clf.fit(X_train, y_train)
>>> results = clf.predict(X_test)
>>> results.shape
(100,)
```

3.5.5 Unsupervised dimensionality 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](#).

PCA: principal component analysis

`decomposition.PCA` looks for a combination of features that capture well the variance of the original features. See [Decomposing signals in components \(matrix factorization problems\)](#).

Examples

- [Faces recognition example using eigenfaces and SVMs](#)

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](#)

Feature agglomeration

`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` may not be able to capture the links between related features. Using a `preprocessing.StandardScaler` can be useful in these settings.

3.5.6 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.

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.

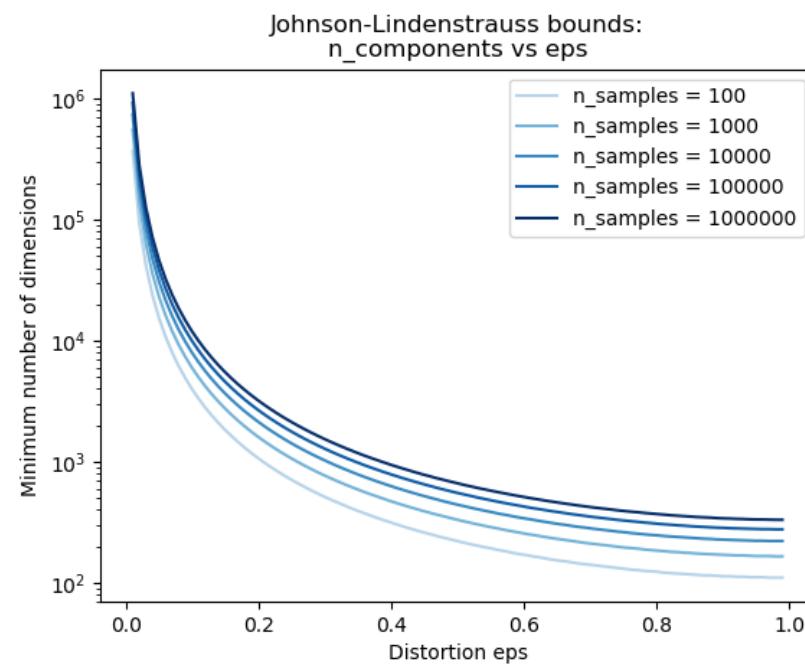
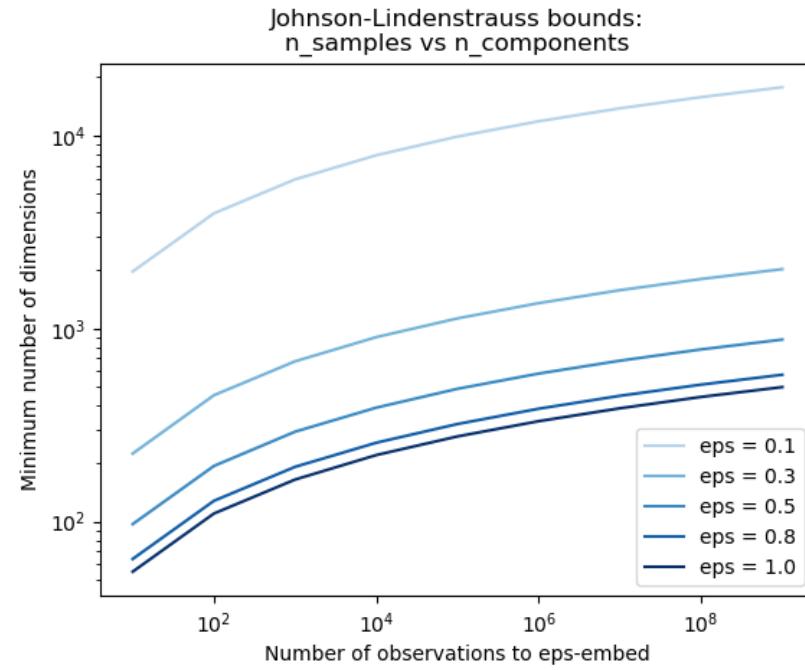
References:

- Sanjoy Dasgupta and Anupam Gupta, 1999. [An elementary proof of the Johnson-Lindenstrauss Lemma](#).

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)
```

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 $s = 1 / \text{density}$, 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.

3.5.7 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.

See also:

Polynomial regression: extending linear models with basis functions for an exact polynomial transformation.

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`.

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(max_iter=5)
>>> clf.fit(X_features, y)
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
              early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
              l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=5,
              n_iter_no_change=5, n_jobs=None, penalty='l2', power_t=0.5,
              random_state=None, shuffle=True, tol=0.001, validation_fraction=0.1,
              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.

Examples:

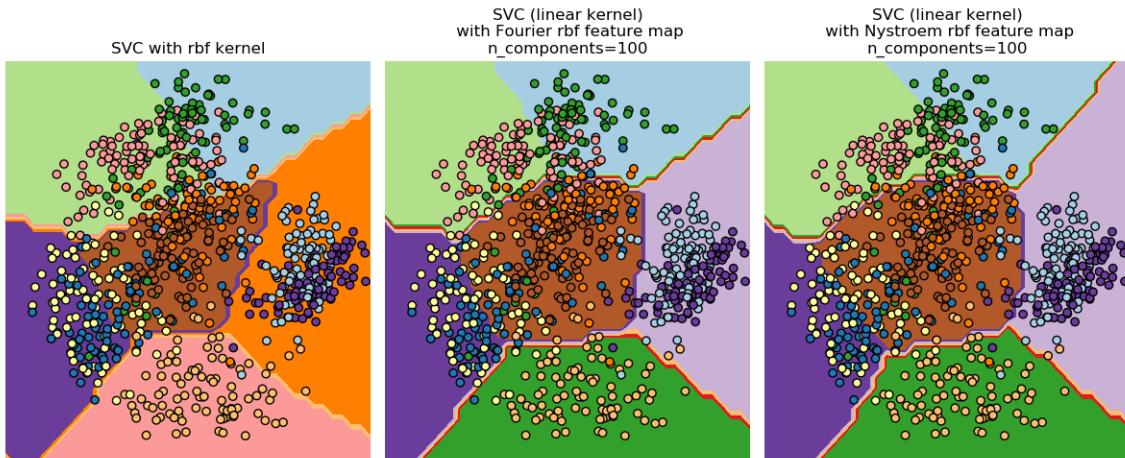


Fig. 3.9: Comparing an exact RBF kernel (left) with the approximation (right)

- *Explicit feature map approximation for RBF kernels*

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`.

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].

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:

3.5.8 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 functions $d(a, b)$ such that $d(a, b) < d(a, c)$ if objects a and b are considered “more similar” than 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. $d(a, b) \geq 0$, 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. $d(a, c) \leq d(a, b) + d(b, c)$, 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” than 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. $S = 1. / (D / np.max(D))$

The distances between the row vectors of X and the row vectors of Y can be evaluated using `pairwise_distances`. If Y is omitted the pairwise distances of the row vectors of X are calculated. Similarly, `pairwise_kernels` can be used to calculate the kernel between X and Y using different kernel functions. See the API reference for more details.

```
>>> import numpy as np
>>> from sklearn.metrics import pairwise_distances
>>> from sklearn.metrics.pairwise import pairwise_kernels
>>> X = np.array([[2, 3], [3, 5], [5, 8]])
>>> Y = np.array([[1, 0], [2, 1]])
>>> pairwise_distances(X, Y, metric='manhattan')
array([[ 4.,  2.],
       [ 7.,  5.],
       [12., 10.]])
>>> pairwise_distances(X, metric='manhattan')
array([[ 0.,  3.,  8.],
       [ 3.,  0.,  5.],
       [ 8.,  5.,  0.]])
>>> pairwise_kernels(X, Y, metric='linear')
array([[ 2.,  7.],
       [ 3., 11.],
       [ 5., 18.]])
```

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:

$$k(x, y) = \frac{xy^\top}{\|x\|\|y\|}$$

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. <https://nlp.stanford.edu/IR-book/html/htmledition/the-vector-space-model-for-scoring-1.html>

Linear kernel

The function `linear_kernel` computes the linear kernel, that is, a special case of `polynomial_kernel` with `degree=1` and `coef0=0` (homogeneous). If x and y are column vectors, their linear kernel is:

$$k(x, y) = x^\top y$$

Polynomial kernel

The function `polynomial_kernel` computes the degree-d polynomial kernel between two vectors. The polynomial kernel represents the similarity between two vectors. Conceptually, the polynomial kernels considers not only the similarity between vectors under the same dimension, but also across dimensions. When used in machine learning algorithms, this allows to account for feature interaction.

The polynomial kernel is defined as:

$$k(x, y) = (\gamma x^\top y + c_0)^d$$

where:

- x, y are the input vectors
- d is the kernel degree

If $c_0 = 0$ the kernel is said to be homogeneous.

Sigmoid kernel

The function `sigmoid_kernel` computes the sigmoid kernel between two vectors. The sigmoid kernel is also known as hyperbolic tangent, or Multilayer Perceptron (because, in the neural network field, it is often used as neuron activation function). It is defined as:

$$k(x, y) = \tanh(\gamma x^\top y + c_0)$$

where:

- x, y are the input vectors
- γ is known as slope
- c_0 is known as intercept

RBF kernel

The function `rbf_kernel` computes the radial basis function (RBF) kernel between two vectors. This kernel is defined as:

$$k(x, y) = \exp(-\gamma \|x - y\|^2)$$

where x and y are the input vectors. If $\gamma = \sigma^{-2}$ the kernel is known as the Gaussian kernel of variance σ^2 .

Laplacian kernel

The function `laplacian_kernel` is a variant on the radial basis function kernel defined as:

$$k(x, y) = \exp(-\gamma \|x - y\|_1)$$

where x and y are the input vectors and $\|x - y\|_1$ is the Manhattan distance between the input vectors.

It has proven useful in ML applied to noiseless data. See e.g. [Machine learning for quantum mechanics in a nutshell](#).

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.          , 0.36787944, 0.89483932, 0.58364548],
       [0.36787944, 1.          , 0.51341712, 0.83822343],
       [0.89483932, 0.51341712, 1.          , 0.7768366 ],
       [0.58364548, 0.83822343, 0.7768366 , 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 <https://research.microsoft.com/en-us/um/people/manik/projects/trade-off/papers/ZhangIJCV06.pdf>

3.5.9 Transforming the prediction target (`y`)

These are transformers that are not intended to be used on features, only on supervised learning targets. See also *Transforming target in regression* if you want to transform the prediction target for learning, but evaluate the model in the original (untransformed) space.

Label binarization

`LabelBinarizer` is a utility class to help create a label indicator matrix from a list of multi-class labels:

```
>>> from sklearn import preprocessing
>>> 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])
```

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']
```

3.6 Dataset loading utilities

The `sklearn.datasets` package embeds some small toy datasets as introduced in the [Getting Started](#) section.

This package also features helpers to fetch larger datasets commonly used by the machine learning community to benchmark algorithms on data that comes from the ‘real world’.

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.

3.6.1 General dataset API

There are three main kinds of dataset interfaces that can be used to get datasets depending on the desired type of dataset.

The dataset loaders. They can be used to load small standard datasets, described in the [Toy datasets](#) section.

The dataset fetchers. They can be used to download and load larger datasets, described in the [Real world datasets](#) section.

Both loaders and fetchers 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`.

It's also possible for almost all of these function to constrain the output to be a tuple containing only the data and the target, by setting the `return_X_y` parameter to `True`.

The datasets also contain a full description in their `DESCR` attribute and some contain `feature_names` and `target_names`. See the dataset descriptions below for details.

The dataset generation functions. They can be used to generate controlled synthetic datasets, described in the [Generated datasets](#) section.

These functions return a tuple `(X, y)` consisting of a `n_samples * n_features` numpy array `X` and an array of length `n_samples` containing the targets `y`.

In addition, there are also miscellaneous tools to load datasets of other formats or from other locations, described in the [Loading other datasets](#) section.

3.6.2 Toy datasets

scikit-learn comes with a few small standard datasets that do not require to download any file from some external website.

They can be loaded using the following functions:

<code>load_boston([return_X_y])</code>	Load and return the boston house-prices dataset (regression).
<code>load_iris([return_X_y])</code>	Load and return the iris dataset (classification).
<code>load_diabetes([return_X_y])</code>	Load and return the diabetes dataset (regression).
<code>load_digits([n_class, return_X_y])</code>	Load and return the digits dataset (classification).
<code>load_linnerud([return_X_y])</code>	Load and return the linnerud dataset (multivariate regression).
<code>load_wine([return_X_y])</code>	Load and return the wine dataset (classification).
<code>load_breast_cancer([return_X_y])</code>	Load and return the breast cancer wisconsin dataset (classification).

These datasets are useful to quickly illustrate the behavior of the various algorithms implemented in scikit-learn. They are however often too small to be representative of real world machine learning tasks.

Boston house prices dataset

Data Set Characteristics:

Number of Instances 506

Number of Attributes 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.

Attribute Information (in order)

- CRIM per capita crime rate by town
- ZN proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS proportion of non-retail business acres per town
- CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX nitric oxides concentration (parts per 10 million)
- RM average number of rooms per dwelling
- AGE proportion of owner-occupied units built prior to 1940
- DIS weighted distances to five Boston employment centres
- RAD index of accessibility to radial highways
- TAX full-value property-tax rate per \$10,000
- PTRATIO pupil-teacher ratio by town
- B 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT % lower status of the population
- MEDV Median value of owner-occupied homes in \$1000's

Missing Attribute Values None

Creator Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset. <https://archive.ics.uci.edu/ml/machine-learning-databases/housing/>

This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. ‘Hedonic prices and the demand for clean air’, J. Environ. Economics & Management, vol.5, 81-102, 1978. Used in Belsley, Kuh & Welsch, ‘Regression diagnostics ...’, Wiley, 1980. N.B. Various transformations are used in the table on pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression problems.

References

- Belsley, Kuh & Welsch, ‘Regression diagnostics: Identifying Influential Data and Sources of Collinearity’, Wiley, 1980. 244-261.
- Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.

Iris plants dataset

Data Set Characteristics:

Number of Instances 150 (50 in each of three classes)

Number of Attributes 4 numeric, predictive attributes and the class

Attribute Information

- sepal length in cm
- sepal width in cm
- petal length in cm
- petal width in cm
- **class:**
 - Iris-Setosa
 - Iris-Versicolour
 - Iris-Virginica

Summary Statistics

sepal length:	4.3	7.9	5.84	0.83	0.7826
sepal width:	2.0	4.4	3.05	0.43	-0.4194
petal length:	1.0	6.9	3.76	1.76	0.9490 (high!)
petal width:	0.1	2.5	1.20	0.76	0.9565 (high!)

Missing Attribute Values None

Class Distribution 33.3% for each of 3 classes.

Creator R.A. Fisher

Donor Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)

Date July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken from Fisher's paper. Note that it's the same as in R, but not as in the UCI Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the pattern recognition literature. Fisher's paper is a classic in the field and is referenced frequently to this day. (See Duda & Hart, for example.) The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other.

References

- Fisher, R.A. "The use of multiple measurements in taxonomic problems" Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to Mathematical Statistics" (John Wiley, NY, 1950).
- Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis. (Q327.D83) John Wiley & Sons. ISBN 0-471-22361-1. See page 218.
- Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System Structure and Classification Rule for Recognition in Partially Exposed Environments". IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-2, No. 1, 67-71.

- Gates, G.W. (1972) “The Reduced Nearest Neighbor Rule”. IEEE Transactions on Information Theory, May 1972, 431-433.
- See also: 1988 MLC Proceedings, 54-64. Cheeseman et al’s AUTOCLASS II conceptual clustering system finds 3 classes in the data.
- Many, many more ...

Diabetes dataset

Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of $n = 442$ diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline.

Data Set Characteristics:

Number of Instances 442

Number of Attributes First 10 columns are numeric predictive values

Target Column 11 is a quantitative measure of disease progression one year after baseline

Attribute Information

- Age
- Sex
- Body mass index
- Average blood pressure
- S1
- S2
- S3
- S4
- S5
- S6

Note: Each of these 10 feature variables have been mean centered and scaled by the standard deviation times $n_samples$ (i.e. the sum of squares of each column totals 1).

Source URL: <https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html>

For more information see: Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004) “Least Angle Regression,” Annals of Statistics (with discussion), 407-499. (https://web.stanford.edu/~hastie/Papers/LARS/LeastAngle_2002.pdf)

Optical recognition of handwritten digits dataset

Data Set Characteristics:

Number of Instances 5620

Number of Attributes 64

Attribute Information 8x8 image of integer pixels in the range 0..16.

Missing Attribute Values None

Creator

5. Alpaydin (alpaydin '@' boun.edu.tr)

Date July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets <https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits>

The data set contains images of hand-written digits: 10 classes where each class refers to a digit.

Preprocessing programs made available by NIST were used to extract normalized bitmaps of handwritten digits from a preprinted form. From a total of 43 people, 30 contributed to the training set and different 13 to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of 4x4 and the number of on pixels are counted in each block. This generates an input matrix of 8x8 where each element is an integer in the range 0..16. This reduces dimensionality and gives invariance to small distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G. T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C. L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469, 1994.

References

- C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their Applications to Handwritten Digit Recognition, MSc Thesis, Institute of Graduate Studies in Science and Engineering, Bogazici University.
- 5. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.
- Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin. Linear dimensionality reduction using relevance weighted LDA. School of Electrical and Electronic Engineering Nanyang Technological University. 2005.
- Claudio Gentile. A New Approximate Maximal Margin Classification Algorithm. NIPS. 2000.

Linnerud dataset

Data Set Characteristics:

Number of Instances 20

Number of Attributes 3

Missing Attribute Values None

The Linnerud dataset contains two small datasets:

- **physiological - CSV containing 20 observations on 3 exercise variables:** Weight, Waist and Pulse.
- **exercise - CSV containing 20 observations on 3 physiological variables:** Chins, Situps and Jumps.

References

- Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris: Editions Technic.

Wine recognition dataset

Data Set Characteristics:

Number of Instances 178 (50 in each of three classes)

Number of Attributes 13 numeric, predictive attributes and the class

Attribute Information

- Alcohol
- Malic acid
- Ash
- Alcalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols
- Proanthocyanins
- Color intensity
- Hue
- OD280/OD315 of diluted wines
- Proline
- class:
 - class_0
 - class_1
 - class_2

Summary Statistics

Alcohol:	11.0	14.8	13.0	0.8
Malic Acid:	0.74	5.80	2.34	1.12
Ash:	1.36	3.23	2.36	0.27
Alcalinity of Ash:	10.6	30.0	19.5	3.3
Magnesium:	70.0	162.0	99.7	14.3
Total Phenols:	0.98	3.88	2.29	0.63
Flavanoids:	0.34	5.08	2.03	1.00
Nonflavanoid Phenols:	0.13	0.66	0.36	0.12
Proanthocyanins:	0.41	3.58	1.59	0.57
Colour Intensity:	1.3	13.0	5.1	2.3
Hue:	0.48	1.71	0.96	0.23
OD280/OD315 of diluted wines:	1.27	4.00	2.61	0.71
Proline:	278	1680	746	315

Missing Attribute Values None

Class Distribution class_0 (59), class_1 (71), class_2 (48)

Creator R.A. Fisher

Donor Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)

Date July, 1988

This is a copy of UCI ML Wine recognition datasets. <https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data>

The data is the results of a chemical analysis of wines grown in the same region in Italy by three different cultivators. There are thirteen different measurements taken for different constituents found in the three types of wine.

Original Owners:

Forina, M. et al, PARVUS - An Extendible Package for Data Exploration, Classification and Correlation. Institute of Pharmaceutical and Food Analysis and Technologies, Via Brigata Salerno, 16147 Genoa, Italy.

Citation:

Lichman, M. (2013). UCI Machine Learning Repository [<https://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

References

(1) S. Aeberhard, D. Coomans and O. de Vel, Comparison of Classifiers in High Dimensional Settings, Tech. Rep. no. 92-02, (1992), Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland. (Also submitted to Technometrics).

The data was used with many others for comparing various classifiers. The classes are separable, though only RDA has achieved 100% correct classification. (RDA : 100%, QDA 99.4%, LDA 98.9%, 1NN 96.1% (z-transformed data)) (All results using the leave-one-out technique)

(2) S. Aeberhard, D. Coomans and O. de Vel, “THE CLASSIFICATION PERFORMANCE OF RDA” Tech. Rep. no. 92-01, (1992), Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland. (Also submitted to Journal of Chemometrics).

Breast cancer wisconsin (diagnostic) dataset

Data Set Characteristics:

Number of Instances 569

Number of Attributes 30 numeric, predictive attributes and the class

Attribute Information

- radius (mean of distances from center to points on the perimeter)
- texture (standard deviation of gray-scale values)
- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness (perimeter² / area - 1.0)
- concavity (severity of concave portions of the contour)
- concave points (number of concave portions of the contour)
- symmetry
- fractal dimension (“coastline approximation” - 1)

The mean, standard error, and “worst” or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. For instance, field 3 is Mean Radius, field 13 is Radius SE, field 23 is Worst Radius.

- **class:**
 - WDBC-Malignant
 - WDBC-Benign

Summary Statistics

radius (mean):	6.981	28.11
texture (mean):	9.71	39.28
perimeter (mean):	43.79	188.5
area (mean):	143.5	2501.0
smoothness (mean):	0.053	0.163
compactness (mean):	0.019	0.345
concavity (mean):	0.0	0.427
concave points (mean):	0.0	0.201
symmetry (mean):	0.106	0.304
fractal dimension (mean):	0.05	0.097
radius (standard error):	0.112	2.873
texture (standard error):	0.36	4.885
perimeter (standard error):	0.757	21.98
area (standard error):	6.802	542.2
smoothness (standard error):	0.002	0.031
compactness (standard error):	0.002	0.135
concavity (standard error):	0.0	0.396
concave points (standard error):	0.0	0.053
symmetry (standard error):	0.008	0.079
fractal dimension (standard error):	0.001	0.03
radius (worst):	7.93	36.04
texture (worst):	12.02	49.54
perimeter (worst):	50.41	251.2
area (worst):	185.2	4254.0
smoothness (worst):	0.071	0.223
compactness (worst):	0.027	1.058
concavity (worst):	0.0	1.252
concave points (worst):	0.0	0.291
symmetry (worst):	0.156	0.664
fractal dimension (worst):	0.055	0.208

Missing Attribute Values None

Class Distribution 212 - Malignant, 357 - Benign

Creator Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian

Donor Nick Street

Date November, 1995

This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets. <https://goo.gl/U2Uwz2>

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.

Separating plane described above was obtained using Multisurface Method-Tree (MSM-T) [K. P. Bennett, “Decision Tree Construction Via Linear Programming.” Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society, pp. 97-101, 1992], a classification method which uses linear programming to construct a decision tree. Relevant features were selected using an exhaustive search in the space of 1-4 features and 1-3 separating planes.

The actual linear program used to obtain the separating plane in the 3-dimensional space is that described in: [K. P. Bennett and O. L. Mangasarian: “Robust Linear Programming Discrimination of Two Linearly Inseparable Sets”, Optimization Methods and Software 1, 1992, 23-34].

This database is also available through the UW CS ftp server:

```
ftp ftp.cs.wisc.edu cd math-prog/cpo-dataset/machine-learn/WDBC/
```

References

- W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature extraction for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on Electronic Imaging: Science and Technology, volume 1905, pages 861-870, San Jose, CA, 1993.
- O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosis and prognosis via linear programming. Operations Research, 43(4), pages 570-577, July-August 1995.
- W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning techniques to diagnose breast cancer from fine-needle aspirates. Cancer Letters 77 (1994) 163-171.

3.6.3 Real world datasets

scikit-learn provides tools to load larger datasets, downloading them if necessary.

They can be loaded using the following functions:

<code>fetch_olivetti_faces([data_home, shuffle, ...])</code>	Load the Olivetti faces data-set from AT&T (classification).
<code>fetch_20newsgroups([data_home, subset, ...])</code>	Load the filenames and data from the 20 newsgroups dataset (classification).
<code>fetch_20newsgroups_vectorized([subset, ...])</code>	Load the 20 newsgroups dataset and vectorize it into token counts (classification).
<code>fetch_lfw_people([data_home, funneled, ...])</code>	Load the Labeled Faces in the Wild (LFW) people dataset (classification).
<code>fetch_lfw_pairs([subset, data_home, ...])</code>	Load the Labeled Faces in the Wild (LFW) pairs dataset (classification).
<code>fetch_covtype([data_home, ...])</code>	Load the covtype dataset (classification).
<code>fetch_rcv1([data_home, subset, ...])</code>	Load the RCV1 multilabel dataset (classification).
<code>fetch_kddcup99([subset, data_home, shuffle, ...])</code>	Load the kddcup99 dataset (classification).
<code>fetch_california_housing([data_home, ...])</code>	Load the California housing dataset (regression).

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 `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).

Data Set Characteristics:

Classes	40
Samples total	400
Dimensionality	4096
Features	real, between 0 and 1

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.

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.CountVectorizer` 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.

Data Set Characteristics:

Classes	20
Samples total	18846
Dimensionality	1
Features	text

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_files` 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([ 7,  4,  4,  1, 14, 16, 13,  3,  2,  4])
```

It is possible to load only a sub-selection of the categories by passing the list of the categories to load to the `sklearn.datasets.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([0, 1, 1, 1, 0, 1, 1, 0, 0, 0])
```

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])
159.01327...
```

`sklearn.datasets.fetch_20newsgroups_vectorized` is a function which returns ready-to-use token counts features instead of file names.

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)
MultinomialNB(alpha=0.01, class_prior=None, fit_prior=True)

>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred, average='macro')
0.88213...
```

(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: edu it and in you that is of to the
comp.graphics: edu in graphics it is for and of to the
sci.space: edu it that is in and space to of the
talk.religion.misc: not it you in is that and to of the
```

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, average='macro')
0.77310...
```

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 = MultinomialNB(alpha=.01)
>>> clf.fit(vectors, newsgroups_train.target)
MultinomialNB(alpha=0.01, class_prior=None, fit_prior=True)
```

```
>>> vectors_test = vectorizer.transform(newsgroups_test.data)
>>> pred = clf.predict(vectors_test)
>>> metrics.f1_score(newsgroups_test.target, pred, average='macro')
0.76995...
```

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](#)

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.

Data Set Characteristics:

Classes	5749
Samples total	13233
Dimensionality	5828
Features	real, between 0 and 255

Usage

scikit-learn provides two loaders that will automatically download, cache, parse the metadata files, decode the jpeg and convert the interesting slices into memmapped 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 memmapped 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 `sklearn.datasets.fetch_lfw_people` and `sklearn.datasets.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 `sklearn.datasets.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.

Examples

Faces recognition example using eigenfaces and SVMs

Forest covtypes

The samples in this dataset correspond to 30x30m 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 covtypes, 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.

Data Set Characteristics:

Classes	7
Samples total	581012
Dimensionality	54
Features	int

`sklearn.datasets.fetch_covtype` will load the covtype 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.

RCV1 dataset

Reuters Corpus Volume I (RCV1) is an archive of over 800,000 manually categorized newswire stories made available by Reuters, Ltd. for research purposes. The dataset is extensively described in¹.

Data Set Characteristics:

Classes	103
Samples total	804414
Dimensionality	47236
Features	real, between 0 and 1

`sklearn.datasets.fetch_rcv1` will load the following version: RCV1-v2, vectors, full sets, topics multilabels:

```
>>> from sklearn.datasets import fetch_rcv1
>>> rcv1 = fetch_rcv1()
```

It returns a dictionary-like object, with the following attributes:

`data`: The feature matrix is a scipy CSR sparse matrix, with 804414 samples and 47236 features. Non-zero values contains cosine-normalized, log TF-IDF vectors. A nearly chronological split is proposed in¹: The first 23149 samples are the training set. The last 781265 samples are the testing set. This follows the official LYRL2004 chronological split. The array has 0.16% of non zero values:

```
>>> rcv1.data.shape
(804414, 47236)
```

`target`: The target values are stored in a scipy CSR sparse matrix, with 804414 samples and 103 categories. Each sample has a value of 1 in its categories, and 0 in others. The array has 3.15% of non zero values:

```
>>> rcv1.target.shape
(804414, 103)
```

`sample_id`: Each sample can be identified by its ID, ranging (with gaps) from 2286 to 810596:

```
>>> rcv1.sample_id[:3]
array([2286, 2287, 2288], dtype=uint32)
```

¹ Lewis, D. D., Yang, Y., Rose, T. G., & Li, F. (2004). RCV1: A new benchmark collection for text categorization research. *The Journal of Machine Learning Research*, 5, 361-397.

`target_names`: The target values are the topics of each sample. Each sample belongs to at least one topic, and to up to 17 topics. There are 103 topics, each represented by a string. Their corpus frequencies span five orders of magnitude, from 5 occurrences for ‘GMIL’, to 381327 for ‘CCAT’:

```
>>> rcv1.target_names[:3].tolist()
['E11', 'ECAT', 'M11']
```

The dataset will be downloaded from the [rcv1](#) homepage if necessary. The compressed size is about 656 MB.

References

Kddcup 99 dataset

The KDD Cup ‘99 dataset was created by processing the tcpdump portions of the 1998 DARPA Intrusion Detection System (IDS) Evaluation dataset, created by MIT Lincoln Lab [1]. The artificial data (described on the [dataset’s homepage](#)) was generated using a closed network and hand-injected attacks to produce a large number of different types of attack with normal activity in the background. As the initial goal was to produce a large training set for supervised learning algorithms, there is a large proportion (80.1%) of abnormal data which is unrealistic in real world, and inappropriate for unsupervised anomaly detection which aims at detecting ‘abnormal’ data, ie

1. qualitatively different from normal data
2. in large minority among the observations.

We thus transform the KDD Data set into two different data sets: SA and SF.

-SA is obtained by simply selecting all the normal data, and a small proportion of abnormal data to gives an anomaly proportion of 1%.

-SF is obtained as in [2] by simply picking up the data whose attribute `logged_in` is positive, thus focusing on the intrusion attack, which gives a proportion of 0.3% of attack.

-http and smtp are two subsets of SF corresponding with third feature equal to ‘http’ (resp. to ‘smtp’)

General KDD structure :

Samples total	4898431
Dimensionality	41
Features	discrete (int) or continuous (float)
Targets	str, ‘normal.’ or name of the anomaly type

SA structure :

Samples total	976158
Dimensionality	41
Features	discrete (int) or continuous (float)
Targets	str, ‘normal.’ or name of the anomaly type

SF structure :

Samples total	699691
Dimensionality	4
Features	discrete (int) or continuous (float)
Targets	str, ‘normal.’ or name of the anomaly type

http structure :

Samples total	619052
Dimensionality	3
Features	discrete (int) or continuous (float)
Targets	str, ‘normal.’ or name of the anomaly type

smtp structure :

Samples total	95373
Dimensionality	3
Features	discrete (int) or continuous (float)
Targets	str, ‘normal.’ or name of the anomaly type

`sklearn.datasets.fetch_kddcup99` will load the kddcup99 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.

California Housing dataset

Data Set Characteristics:

Number of Instances 20640

Number of Attributes 8 numeric, predictive attributes and the target

Attribute Information

- MedInc median income in block
- HouseAge median house age in block
- AveRooms average number of rooms
- AveBedrms average number of bedrooms
- Population block population
- AveOccup average house occupancy
- Latitude house block latitude
- Longitude house block longitude

Missing Attribute Values

None

This dataset was obtained from the StatLib repository. <http://lib.stat.cmu.edu/datasets/>

The target variable is the median house value for California districts.

This dataset was derived from the 1990 U.S. census, using one row per census block group. A block group is the smallest geographical unit for which the U.S. Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people).

It can be downloaded/loaded using the `sklearn.datasets.fetch_california_housing` function.

References

- Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions, Statistics and Probability Letters, 33 (1997) 291-297

3.6.4 Generated datasets

In addition, scikit-learn includes various random sample generators that can be used to build artificial datasets of controlled size and complexity.

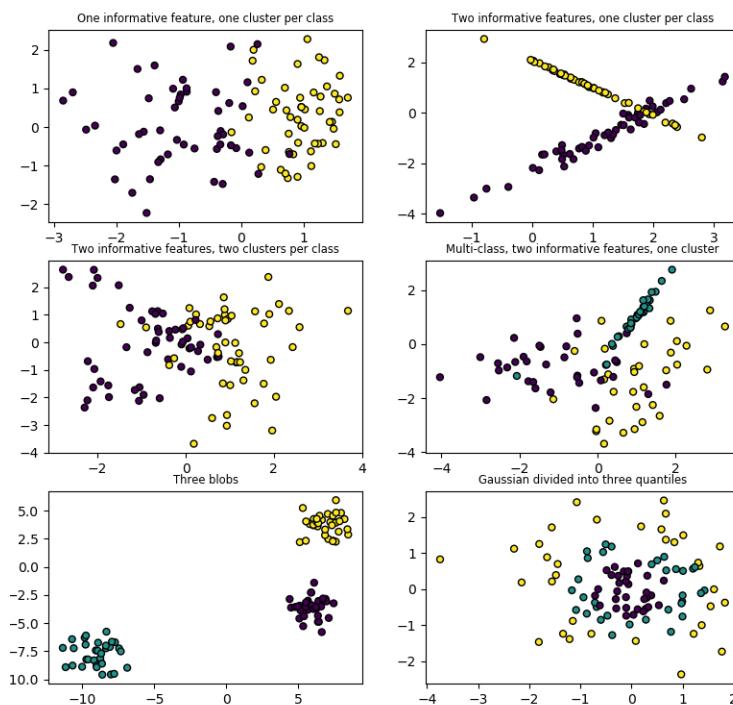
Generators for classification and clustering

These generators produce a matrix of features and corresponding discrete targets.

Single label

Both `make_blobs` and `make_classification` create multiclass datasets by allocating each class one or more normally-distributed clusters of points. `make_blobs` provides greater control regarding the centers and standard deviations of each cluster, and is used to demonstrate clustering. `make_classification` specialises in introducing noise by way of: correlated, redundant and uninformative features; multiple Gaussian clusters per class; and linear transformations of the feature space.

`make_gaussian_quantiles` divides a single Gaussian cluster into near-equal-size classes separated by concentric hyperspheres. `make_hastie_10_2` generates a similar binary, 10-dimensional problem.

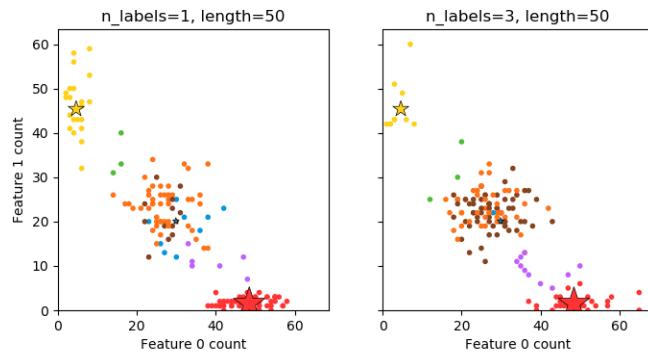


`make_circles` and `make_moons` generate 2d binary classification datasets that are challenging to certain algorithms (e.g. centroid-based clustering or linear classification), including optional Gaussian noise. They are useful for visualisation. `make_circles` produces Gaussian data with a spherical decision boundary for binary classification, while `make_moons` produces two interleaving half circles.

Multilabel

`make_multilabel_classification` generates random samples with multiple labels, reflecting a bag of words drawn from a mixture of topics. The number of topics for each document is drawn from a Poisson distribution, and the topics themselves are drawn from a fixed random distribution. Similarly, the number of words is drawn from Poisson, with words drawn from a multinomial, where each topic defines a probability distribution over words. Simplifications with respect to true bag-of-words mixtures include:

- Per-topic word distributions are independently drawn, where in reality all would be affected by a sparse base distribution, and would be correlated.
- For a document generated from multiple topics, all topics are weighted equally in generating its bag of words.
- Documents without labels words at random, rather than from a base distribution.



Biclustering

<code>make_biclusters(shape, n_clusters[, noise, ...])</code>	Generate an array with constant block diagonal structure for biclustering.
<code>make_checkerboard(shape, n_clusters[, ...])</code>	Generate an array with block checkerboard structure for biclustering.

Generators for regression

`make_regression` produces regression targets as an optionally-sparse random linear combination of random features, with noise. Its informative features may be uncorrelated, or low rank (few features account for most of the variance).

Other regression generators generate functions deterministically from randomized features. `make_sparse_uncorrelated` produces a target as a linear combination of four features with fixed coefficients. Others encode explicitly non-linear relations: `make_friedman1` is related by polynomial and sine transforms; `make_friedman2` includes feature multiplication and reciprocation; and `make_friedman3` is similar with an arctan transformation on the target.

Generators for manifold learning

<code>make_s_curve([n_samples, noise, random_state])</code>	Generate an S curve dataset.
<code>make_swiss_roll([n_samples, noise, random_state])</code>	Generate a swiss roll dataset.

Generators for decomposition

<code>make_low_rank_matrix([n_samples, ...])</code>	Generate a mostly low rank matrix with bell-shaped singular values
<code>make_sparse_coded_signal(n_samples, ...[, ...])</code>	Generate a signal as a sparse combination of dictionary elements.
<code>make_spd_matrix(n_dim[, random_state])</code>	Generate a random symmetric, positive-definite matrix.
<code>make_sparse_spd_matrix([dim, alpha, ...])</code>	Generate a sparse symmetric definite positive matrix.

3.6.5 Loading other datasets

Sample images

Scikit-learn also embed a couple of sample JPEG images published under Creative Commons license by their authors. Those images can be useful to test algorithms and pipeline on 2D data.

<code>load_sample_images()</code>	Load sample images for image manipulation.
<code>load_sample_image(image_name)</code>	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 `matplotlib.pyplot.imshow` don't forget to scale to the range 0 - 1 as done in the following example.

Examples:

- *Color Quantization using K-Means*

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: <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets>

Faster API-compatible implementation: <https://github.com/mblondel/svmlight-loader>

Downloading datasets from the openml.org repository

`openml.org` is a public repository for machine learning data and experiments, that allows everybody to upload open datasets.

The `sklearn.datasets` package is able to download datasets from the repository using the function `sklearn.datasets.fetch_openml`.

For example, to download a dataset of gene expressions in mice brains:

```
>>> from sklearn.datasets import fetch_openml
>>> mice = fetch_openml(name='miceprotein', version=4)
```

To fully specify a dataset, you need to provide a name and a version, though the version is optional, see *Dataset Versions* below. The dataset contains a total of 1080 examples belonging to 8 different classes:

```
>>> mice.data.shape
(1080, 77)
>>> mice.target.shape
(1080,)
>>> np.unique(mice.target)
array(['c-CS-m', 'c-CS-s', 'c-SC-m', 'c-SC-s', 't-CS-m', 't-CS-s', 't-SC-m',
       't-SC-s'], dtype=object)
```

You can get more information on the dataset by looking at the `DESCR` and `details` attributes:

```
>>> print(mice.DESCR)
**Author**: Clara Higuera, Katheleen J. Gardiner, Krzysztof J. Cios
**Source**: [UCI] (https://archive.ics.uci.edu/ml/datasets/Mice+Protein+Expression) - 2015
**Please cite**: Higuera C, Gardiner KJ, Cios KJ (2015) Self-Organizing
Feature Maps Identify Proteins Critical to Learning in a Mouse Model of Down
Syndrome. PLoS ONE 10(6): e0129126...
>>> mice.details
{'id': '40966', 'name': 'MiceProtein', 'version': '4', 'format': 'ARFF',
 'upload_date': '2017-11-08T16:00:15', 'licence': 'Public',
```

```
'url': 'https://www.openml.org/data/v1/download/17928620/MiceProtein.arff',
'file_id': '17928620', 'default_target_attribute': 'class',
'row_id_attribute': 'MouseID',
'ignore_attribute': ['Genotype', 'Treatment', 'Behavior'],
'tag': ['OpenML-CC18', 'study_135', 'study_98', 'study_99'],
'vesibility': 'public', 'status': 'active',
'md5_checksum': '3c479a6885bfa0438971388283a1ce32'}
```

The DESCR contains a free-text description of the data, while details contains a dictionary of meta-data stored by openml, like the dataset id. For more details, see the [OpenML documentation](#) The data_id of the mice protein dataset is 40966, and you can use this (or the name) to get more information on the dataset on the openml website:

```
>>> mice.url
'https://www.openml.org/d/40966'
```

The data_id also uniquely identifies a dataset from OpenML:

```
>>> mice = fetch_openml(data_id=40966)
>>> mice.details
{'id': '4550', 'name': 'MiceProtein', 'version': '1', 'format': 'ARFF',
'creator': ...,
'upload_date': '2016-02-17T14:32:49', 'licence': 'Public', 'url':
'https://www.openml.org/data/v1/download/1804243/MiceProtein.ARFF', 'file_id':
'1804243', 'default_target_attribute': 'class', 'citation': 'Higuera C,
Gardiner KJ, Cios KJ (2015) Self-Organizing Feature Maps Identify Proteins
Critical to Learning in a Mouse Model of Down Syndrome. PLoS ONE 10(6):
e0129126. [Web Link] journal.pone.0129126', 'tag': ['OpenML100', 'study_14',
'study_34'], 'visibility': 'public', 'status': 'active', 'md5_checksum':
'3c479a6885bfa0438971388283a1ce32'}
```

Dataset Versions

A dataset is uniquely specified by its data_id, but not necessarily by its name. Several different “versions” of a dataset with the same name can exist which can contain entirely different datasets. If a particular version of a dataset has been found to contain significant issues, it might be deactivated. Using a name to specify a dataset will yield the earliest version of a dataset that is still active. That means that `fetch_openml(name="miceprotein")` can yield different results at different times if earlier versions become inactive. You can see that the dataset with data_id 40966 that we fetched above is the version 1 of the “miceprotein” dataset:

```
>>> mice.details['version']
'1'
```

In fact, this dataset only has one version. The iris dataset on the other hand has multiple versions:

```
>>> iris = fetch_openml(name="iris")
>>> iris.details['version']
'1'
>>> iris.details['id']
'61'

>>> iris_61 = fetch_openml(data_id=61)
>>> iris_61.details['version']
'1'
>>> iris_61.details['id']
'61'
```

```
>>> iris_969 = fetch_openml(data_id=969)
>>> iris_969.details['version']
'3'
>>> iris_969.details['id']
'969'
```

Specifying the dataset by the name “iris” yields the lowest version, version 1, with the `data_id` 61. To make sure you always get this exact dataset, it is safest to specify it by the dataset `data_id`. The other dataset, with `data_id` 969, is version 3 (version 2 has become inactive), and contains a binarized version of the data:

```
>>> np.unique(iris_969.target)
array(['N', 'P'], dtype=object)
```

You can also specify both the name and the version, which also uniquely identifies the dataset:

```
>>> iris_version_3 = fetch_openml(name="iris", version=3)
>>> iris_version_3.details['version']
'3'
>>> iris_version_3.details['id']
'969'
```

References:

- Vanschoren, van Rijn, Bischl and Torgo “OpenML: networked science in machine learning”, ACM SIGKDD Explorations Newsletter, 15(2), 49-60, 2014.

Loading from external datasets

scikit-learn works on any numeric data stored as numpy arrays or scipy sparse matrices. Other types that are convertible to numeric arrays such as pandas DataFrame are also acceptable.

Here are some recommended ways to load standard columnar data into a format usable by scikit-learn:

- `pandas.io` provides tools to read data from common formats including CSV, Excel, JSON and SQL. DataFrames may also be constructed from lists of tuples or dicts. Pandas handles heterogeneous data smoothly and provides tools for manipulation and conversion into a numeric array suitable for scikit-learn.
- `scipy.io` specializes in binary formats often used in scientific computing context such as .mat and .arff
- `numpy/routines.io` for standard loading of columnar data into numpy arrays
- scikit-learn’s `datasets.load_svmlight_file` for the svmlight or libSVM sparse format
- scikit-learn’s `datasets.load_files` for directories of text files where the name of each directory is the name of each category and each file inside of each directory corresponds to one sample from that category

For some miscellaneous data such as images, videos, and audio, you may wish to refer to:

- `skimage.io` or `Imageio` for loading images and videos into numpy arrays
- `scipy.io.wavfile.read` for reading WAV files into a numpy array

Categorical (or nominal) features stored as strings (common in pandas DataFrames) will need converting to numerical features using `sklearn.preprocessing.OneHotEncoder` or `sklearn.preprocessing.OrdinalEncoder` or similar. See `Preprocessing data`.

Note: if you manage your own numerical data it is recommended to use an optimized file format such as HDF5 to reduce data load times. Various libraries such as H5Py, PyTables and pandas provides a Python interface for reading and writing data in that format.

3.7 Computing with scikit-learn

3.7.1 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.

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 a 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

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.

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 stateful 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.

Incremental learning

Finally, for 3. we have a number of options inside scikit-learn. Although not all algorithms can 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 tuning¹.

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`
- `sklearn.neural_network.MLPClassifier`

- **Regression**

- `sklearn.linear_model.SGDRegressor`
- `sklearn.linear_model.PassiveAggressiveRegressor`
- `sklearn.neural_network.MLPRegressor`

- **Clustering**

- `sklearn.cluster.MiniBatchKMeans`
- `sklearn.cluster.Birch`

- **Decomposition / feature Extraction**

- `sklearn.decomposition.MiniBatchDictionaryLearning`
- `sklearn.decomposition.IncrementalPCA`
- `sklearn.decomposition.LatentDirichletAllocation`

- **Preprocessing**

- `sklearn.preprocessing.StandardScaler`
- `sklearn.preprocessing.MinMaxScaler`
- `sklearn.preprocessing.MaxAbsScaler`

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.

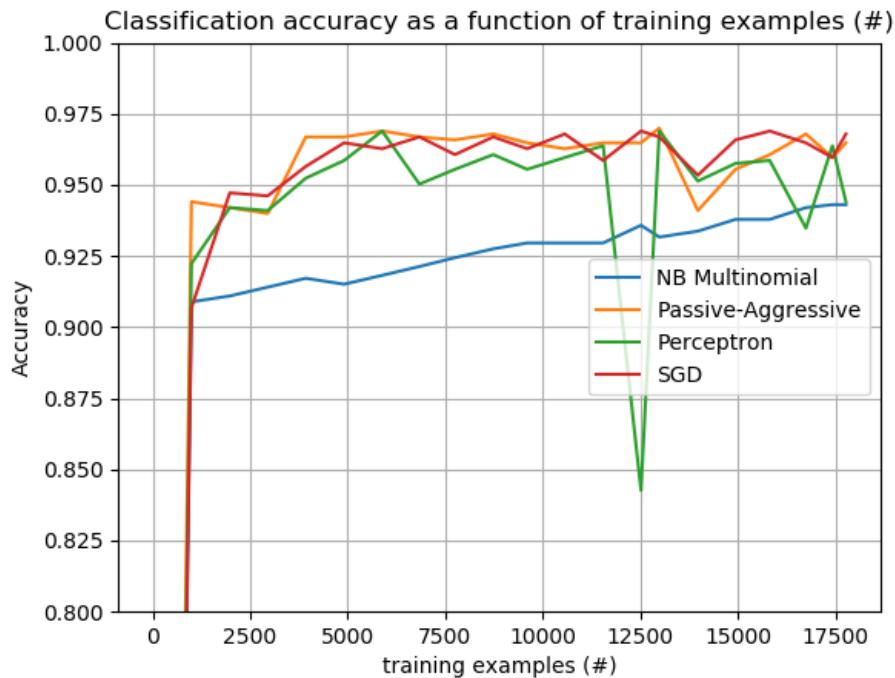
Another aspect to consider when choosing a proper algorithm is that not all of them put the same importance on each example over time. Namely, the Perceptron is still sensitive to badly labeled examples even after many examples whereas the SGD* and PassiveAggressive* families are more robust to this kind of artifacts. Conversely, the latter 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.

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.

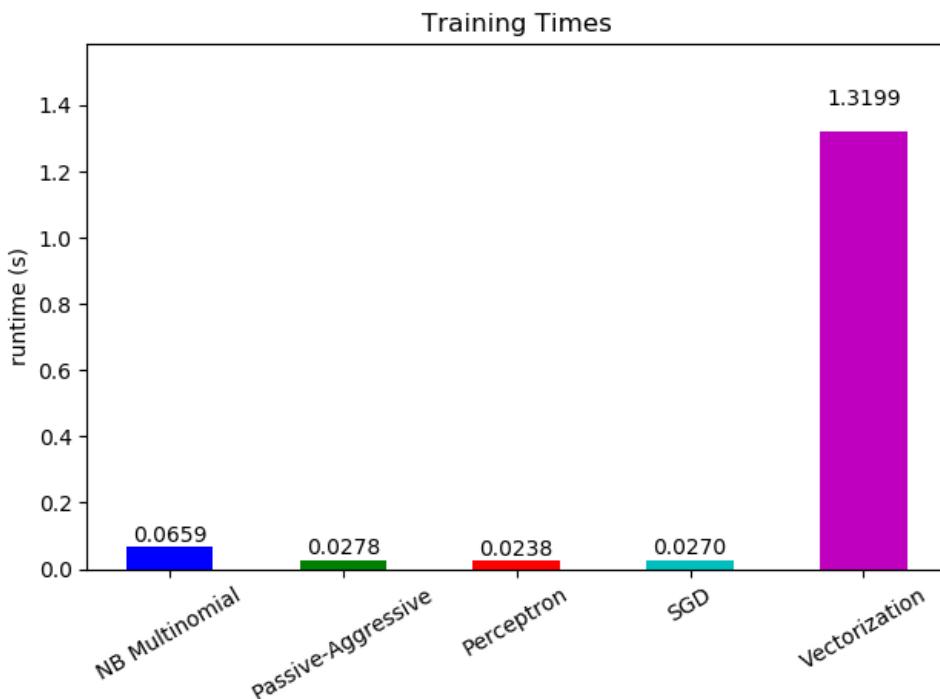
Examples

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).



Notes

3.7.2 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.

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

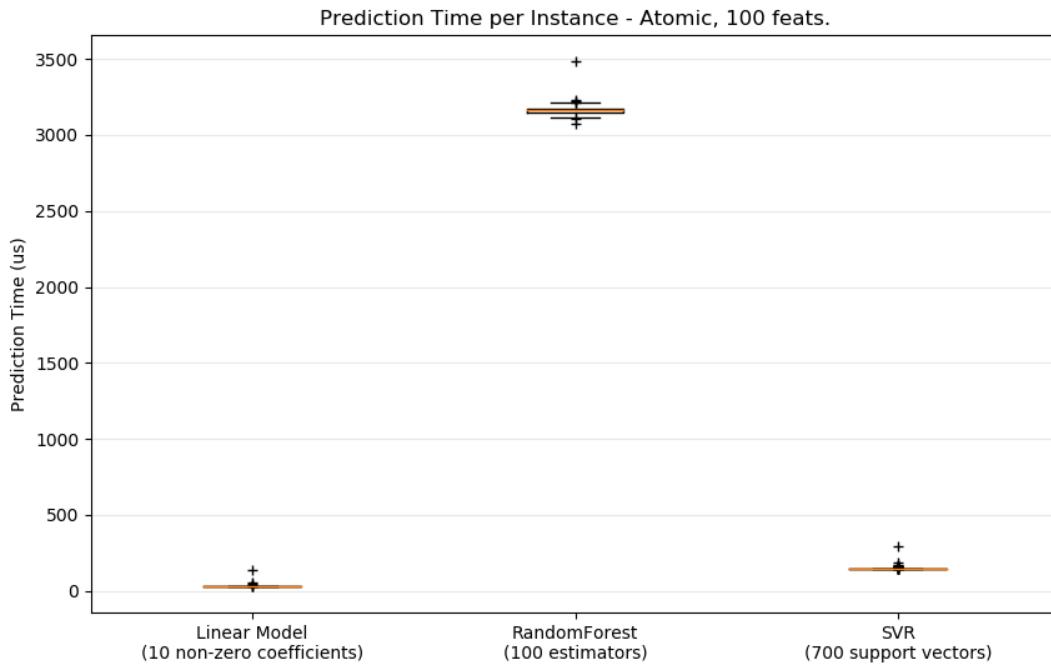
1. Number of features

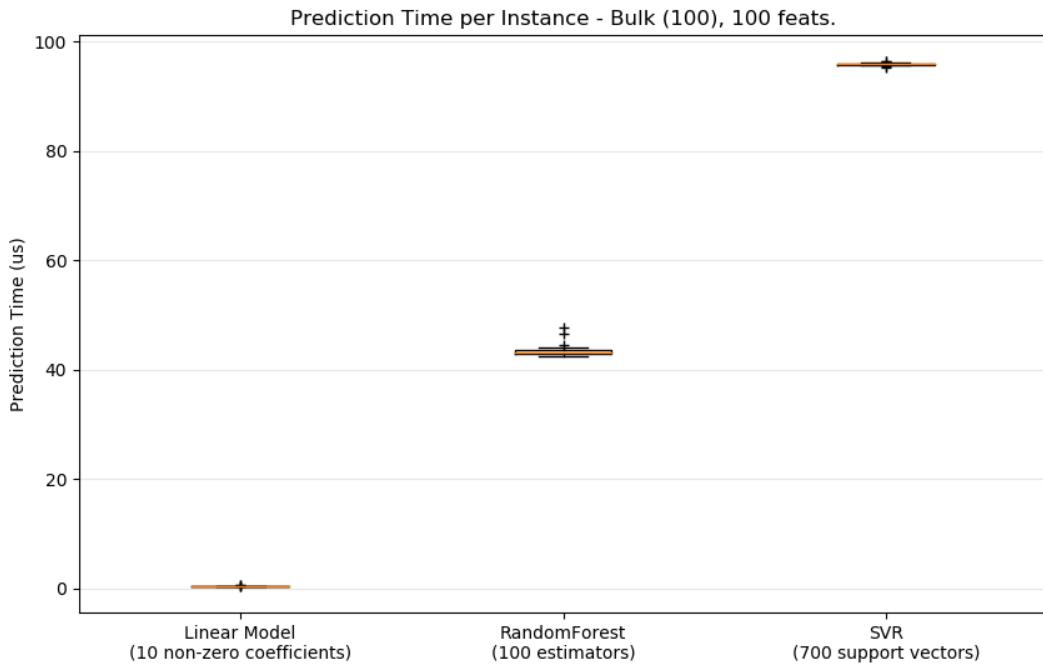
2. Input data representation and sparsity
3. Model complexity
4. Feature extraction

A last major parameter is also the possibility to do predictions in bulk or one-at-a-time mode.

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.

Configuring Scikit-learn for reduced validation overhead

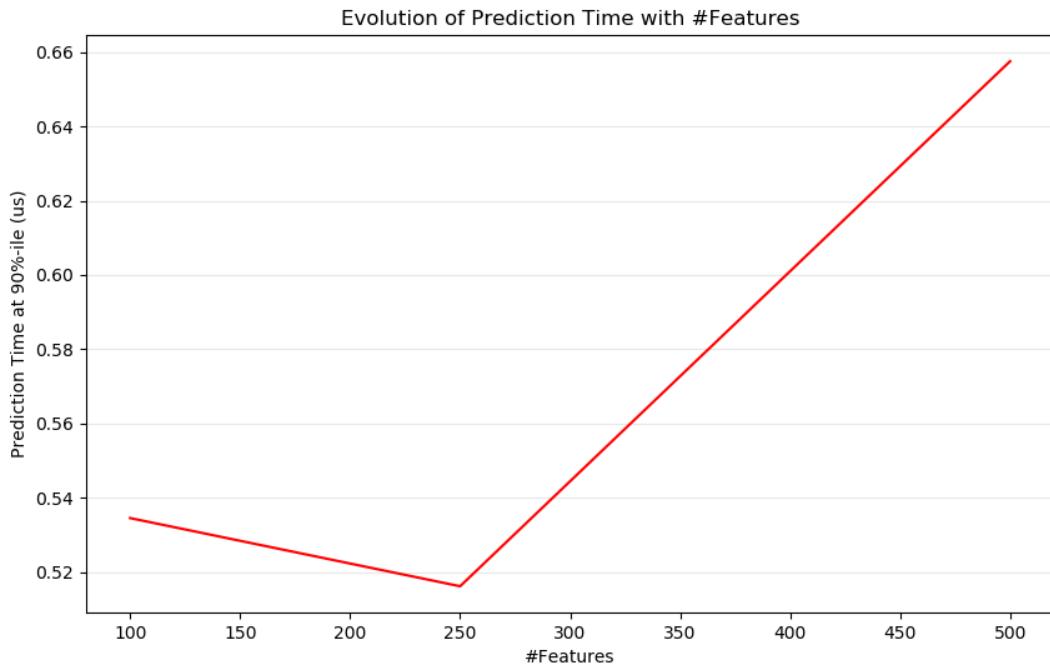
Scikit-learn does some validation on data that increases the overhead per call to `predict` and similar functions. In particular, checking that features are finite (not NaN or infinite) involves a full pass over the data. If you ensure that your data is acceptable, you may suppress checking for finiteness by setting the environment variable `SKLEARN_ASSUMEFINITE` to a non-empty string before importing scikit-learn, or configure it in Python with `sklearn.set_config`. For more control than these global settings, a `config_context` allows you to set this configuration within a specified context:

```
>>> import sklearn
>>> with sklearn.config_context(assume_finite=True):
...     pass # do learning/prediction here with reduced validation
```

Note that this will affect all uses of `sklearn.utils.assert_all_finite` within the context.

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).

Influence of the Input Data Representation

Scipy provides sparse matrix data structures 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))
```

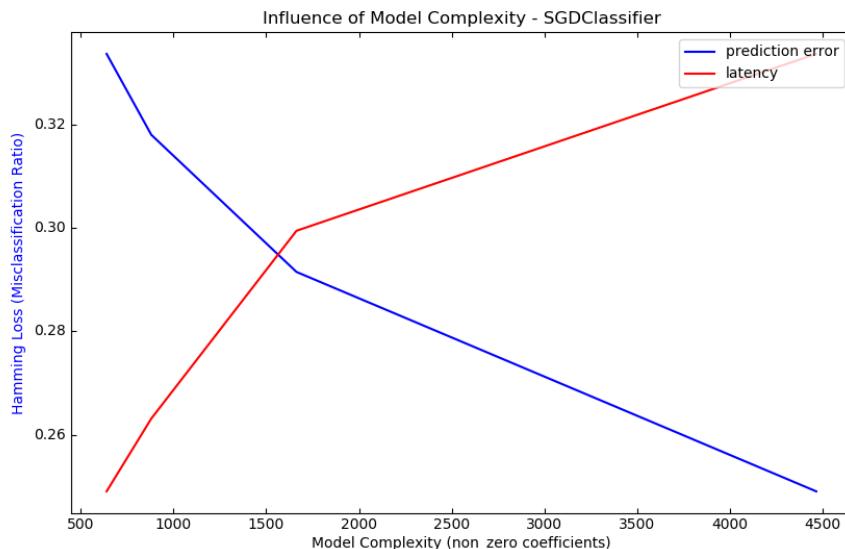
As a rule of thumb you can consider that if the sparsity ratio is greater than 90% you can probably benefit from sparse formats. Check Scipy's sparse matrix formats [documentation](#) for more information on how to build (or convert your data to) sparse matrix formats. Most of the time the CSR and CSC formats work best.

Influence of the Model Complexity

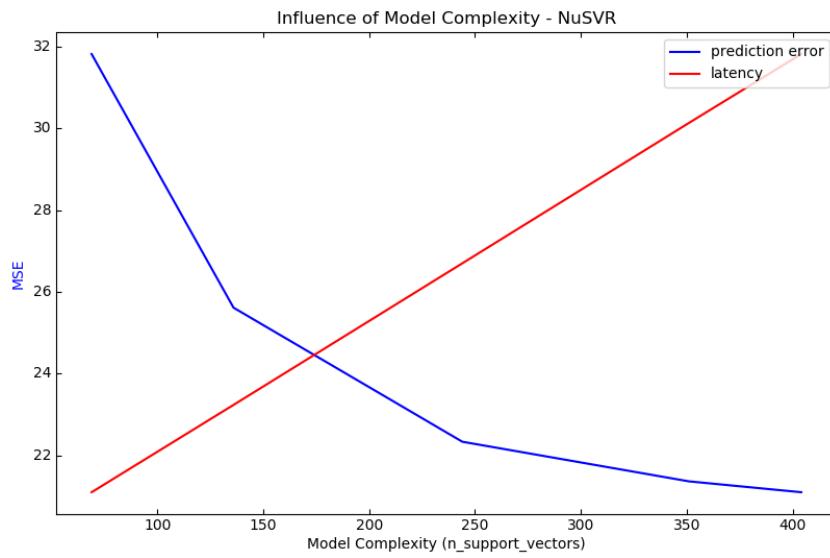
Generally speaking, when model complexity increases, predictive power and latency are supposed to increase. Increasing predictive power is usually interesting, but for many applications we would better not increase prediction latency too much. We will now review this idea for different families of supervised models.

For `sklearn.linear_model` (e.g. Lasso, ElasticNet, SGDClassifier/Regressor, Ridge & RidgeClassifier, PassiveAggressiveClassifier/Regressor, LinearSVC, LogisticRegression...) the decision function that is applied at prediction time is the same (a dot product), so latency should be equivalent.

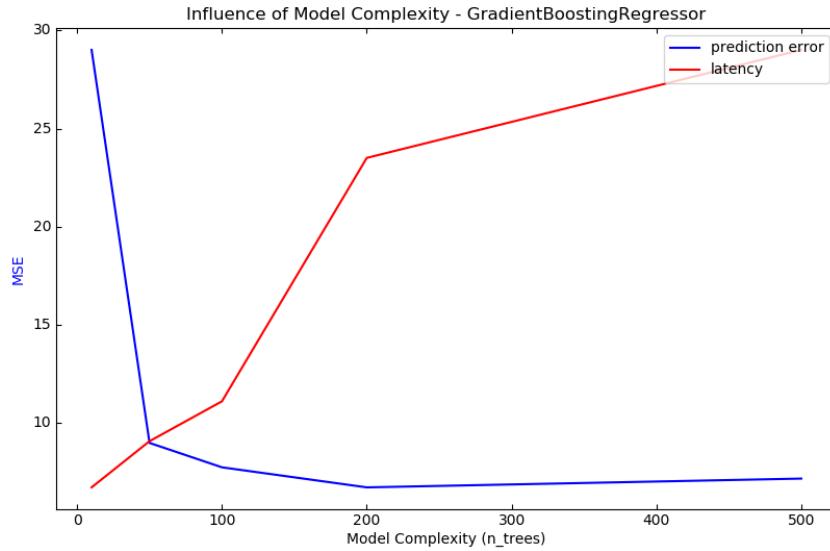
Here is an example using `sklearn.linear_model.stochastic_gradient.SGDClassifier` with the `elasticnet` penalty. The regularization strength is globally controlled by the `alpha` parameter. With a sufficiently high `alpha`, one can then increase the `l1_ratio` parameter of `elasticnet` to enforce various 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.



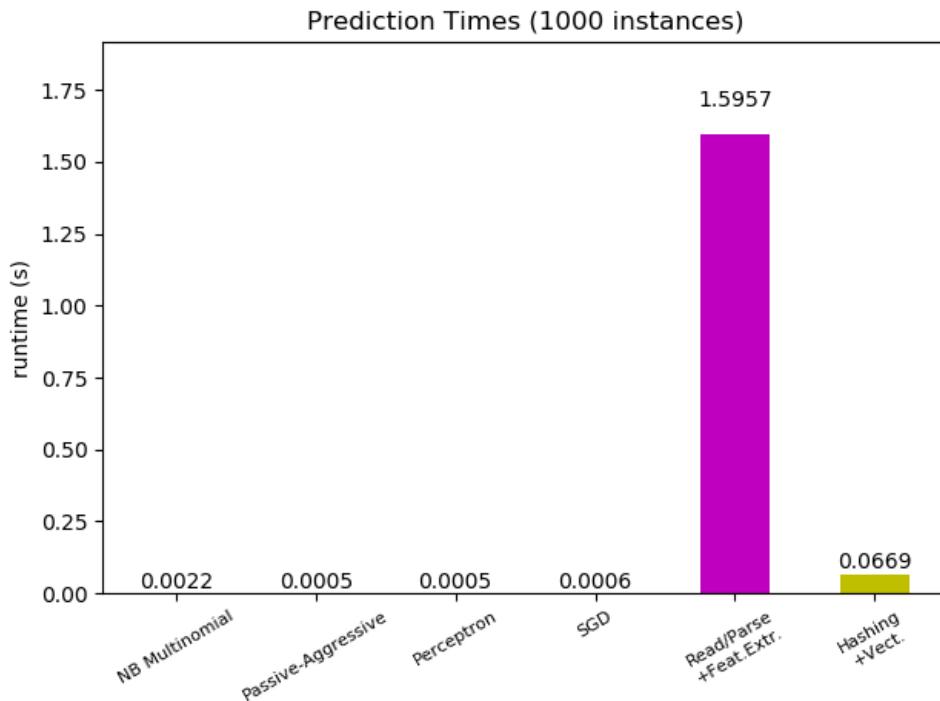
For `sklearn.ensemble` of trees (e.g. RandomForest, GBT, ExtraTrees etc) the number of trees and their depth play the most important role. Latency and throughput should scale linearly with the number of trees. In this case we used directly the `n_estimators` parameter of `sklearn.ensemble.GradientBoostingRegressor`.



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.

Feature Extraction Latency

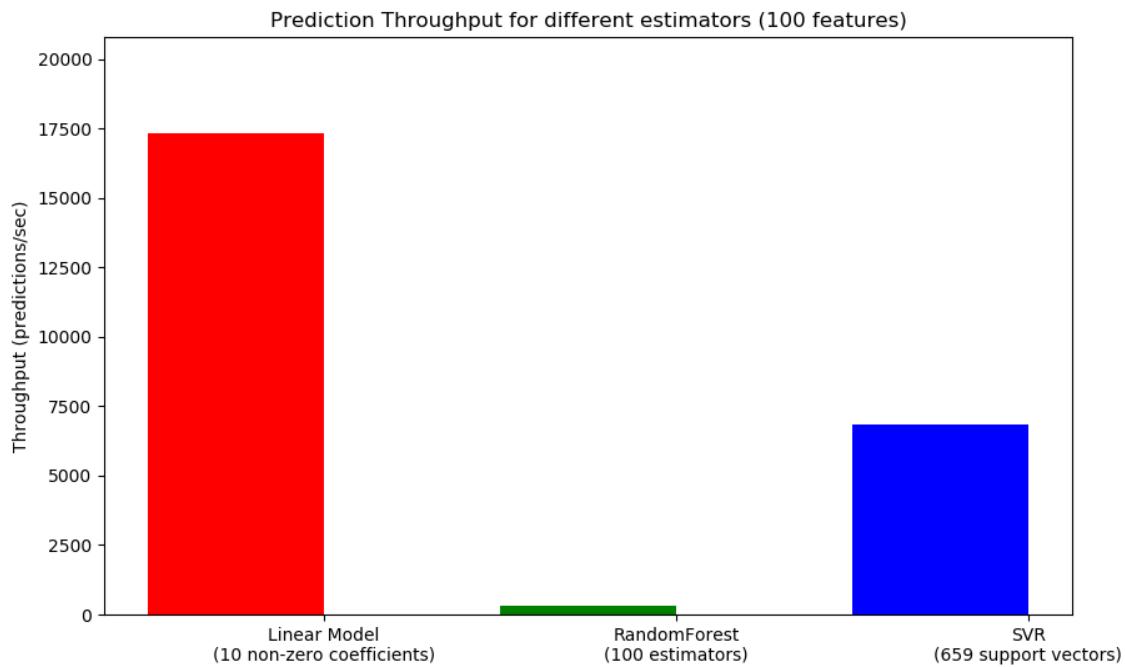
Most scikit-learn models are usually pretty fast as they are implemented either with compiled Cython extensions or optimized computing libraries. On the other hand, in many real world applications the feature extraction process (i.e. turning raw data like database rows or network packets into numpy arrays) 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.

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.

Tips and Tricks

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.

Limiting Working Memory

Some calculations when implemented using standard numpy vectorized operations involve using a large amount of temporary memory. This may potentially exhaust system memory. Where computations can be performed in fixed-memory chunks, we attempt to do so, and allow the user to hint at the maximum size of this working memory (defaulting to 1GB) using `sklearn.set_config` or `config_context`. The following suggests to limit temporary working memory to 128 MiB:

```
>>> import sklearn
>>> with sklearn.config_context(working_memory=128):
...     pass # do chunked work here
```

An example of a chunked operation adhering to this setting is `metric.pairwise_distances_chunked`, which facilitates computing row-wise reductions of a pairwise distance matrix.

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.

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.

Links

- [scikit-learn developer performance documentation](#)
- [Scipy sparse matrix formats documentation](#)

3.7.3 Parallelism, resource management, and configuration

Parallel and distributed computing

Scikit-learn uses the `joblib` library to enable parallel computing inside its estimators. See the `joblib` documentation for the switches to control parallel computing.

Note that, by default, scikit-learn uses its embedded (vendored) version of `joblib`. A configuration switch (documented below) controls this behavior.

Configuration switches

Python runtime

`sklearn.set_config` controls the following behaviors:

- assume_finite** used to skip validation, which enables faster computations but may lead to segmentation faults if the data contains NaNs.
- working_memory** the optimal size of temporary arrays used by some algorithms.

Environment variables

These environment variables should be set before importing scikit-learn.

SKLEARN_SITE_JOBLIB When this environment variable is set to a non zero value, scikit-learn uses the site `joblib` rather than its vendored version. Consequently, `joblib` must be installed for scikit-learn to run. Note that using the site `joblib` is at your own risks: the versions of scikit-learn and `joblib` need to be compatible. Currently, `joblib` 0.11+ is supported. In addition, dumps from `joblib.Memory` might be incompatible, and you might lose some caches and have to redownload some datasets.

Deprecated since version 0.21: As of version 0.21 this parameter has no effect, vendored `joblib` was removed and site `joblib` is always used.

SKLEARN_ASSUME_FINITE Sets the default value for the `assume_finite` argument of `sklearn.set_config`.

SKLEARN_WORKING_MEMORY Sets the default value for the `Limiting Working Memory` argument of `sklearn.set_config`.

SKLEARN_SEED Sets the seed of the global random generator when running the tests, for reproducibility.

SKLEARN_SKIP_NETWORK_TESTS When this environment variable is set to a non zero value, the tests that need network access are skipped.