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.

1.13.1. Removing features with low variance

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

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

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

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

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

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

1.13.2. Univariate feature selection

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

- `SelectKBest` removes all but the k highest scoring features
- `SelectPercentile` removes all but a user-specified highest scoring percentage of features
- using common univariate statistical tests for each feature: false positive rate `SelectFpr`, false discovery rate `SelectFdr`, or family wise error `SelectFwe`.
- `GenericUnivariateSelect` allows to perform univariate feature selection with a configurable strategy. This allows to select the best univariate selection strategy with hyper-parameter search estimator.

For instance, we can use a F-test to retrieve the two best features for a dataset as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import f_classif
>>> X, y = load_iris(return_X_y=True)
>>> X.shape
(150, 4)
>>> X_new = SelectKBest(f_classif, 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: `r_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. Note that the χ^2-test should only be applied to non-negative features, such as frequencies.

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

1.13.3. Recursive feature elimination

Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), the goal of 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 any specific attribute (such as `coef_`, `feature_importances_`) or callable. 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.

1.13.4. Feature selection using SelectFromModel

`SelectFromModel` is a meta-transformer that can be used alongside any estimator that assigns importance to each feature through a specific attribute (such as `coef_`, `feature_importances_`) or via an `importance_getter` callable after fitting. The features are considered unimportant and removed if the corresponding importance of the feature 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”. In combination with the `threshold` criteria, one can use the `max_features` parameter to set a limit on the number of features to select.

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

Examples

- Model-based and sequential feature selection

1.13.4.1. 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 `SelectFromModel` to select the non-zero coefficients. In particular, sparse estimators useful for this purpose are the `Lasso` for regression, and of `LogisticRegression` and `LinearSVC` for classification:

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectFromModel
>>> X, y = load_iris(return_X_y=True)
>>> 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:

- Lasso on dense and sparse data.

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

Toggle Menu

of non-zero coefficients, and the structure of the design matrix X. In addition, the design matrix must display certain specific properties, such as not being too correlated.

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

Reference Richard G. Baraniuk "Compressive Sensing", IEEE Signal Processing Magazine [120] July 2007
http://users.isr.ist.utl.pt/~aguiar/CS_notes.pdf

1.13.4.2. 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 impurity-based feature importances, which in turn can be used to discard irrelevant features (when coupled with the **`SelectFromModel`** meta-transformer):

```
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectFromModel
>>> X, y = load_iris(return_X_y=True)
>>> 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 a forest of trees: example on synthetic data showing the recovery of the actually meaningful features.
- Pixel importances with a parallel forest of trees: example on face recognition data.

1.13.5. Sequential Feature Selection

Sequential Feature Selection [sfs] (SFS) is available in the **`SequentialFeatureSelector`** transformer. SFS can be either forward or backward:

Forward-SFS is a greedy procedure that iteratively finds the best new feature to add to the set of selected features. Concretely, we initially start with zero features and find the one feature that maximizes a cross-validated score when an estimator is trained on this single feature. Once that first feature is selected, we repeat the procedure by adding a new feature to the set of selected features. The procedure stops when the desired number of selected features is reached, as determined by the `n_features_to_select` parameter.

Backward-SFS follows the same idea but works in the opposite direction: instead of starting with no features and greedily adding features, we start with *all* the features and greedily *remove* features from the set. The `direction` parameter controls whether forward or backward SFS is used.

In general, forward and backward selection do not yield equivalent results. Also, one may be much faster than the other depending on the requested number of selected features: if we have 10 features and ask for 7 selected features, forward selection would need to perform 7 iterations while backward selection would only need to perform 3.

SFS differs from **`RFE`** and **`SelectFromModel`** in that it does not require the underlying model to expose a `coef_` or `feature_importances_` attribute. It may however be slower considering that more models need to be evaluated, compared to the other approaches. For example in backward selection, the iteration going from `m` features to `m - 1` features using k-fold cross-validation requires fitting `m * k` models, while **`RFE`** would require only a single fit, and **`SelectFromModel`** always just does a single fit and requires no iterations.

Examples

- Model-based and sequential feature selection

References:

[sfs]
Ferri et al, Comparative study of techniques for large-scale feature selection.

1.13.6. Feature selection as part of a pipeline

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

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

In this snippet we make use of a **`LinearSVC`** coupled with **`SelectFromModel`** to evaluate feature importances and select the most relevant features. Then, a **`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 **`Pipeline`** examples for more details.