scikit-learn-contrib / **boruta_py**

Python implementations of the Boruta all-relevant feature selection method.

README.md

boruta_py

This project hosts Python implementations of the Boruta all-relevant feature selection method.

[Related blog post] (http://danielhomola.com/2015/05/08/borutapy-an-all-relevant-feature-selection-method/)

Dependencies

- numpy
- scipy
- scikit-learn

How to use

Download, import and do as you would with any other scikit-learn method:

- fit(X, y)
- transform(X)
- fit_transform(X, y)

Description

Python implementations of the Boruta R package.

This implementation tries to mimic the scikit-learn interface, so use fit, transform or fit_transform, to run the feautre selection.

For more, see the docs of these functions, and the examples below.

Original code and method by: Miron B Kursa, https://m2.icm.edu.pl/boruta/

Boruta is an all relevant feature selection method, while most other are minimal optimal; this means it tries to find all features carrying information usable for prediction, rather than finding a possibly compact subset of features on which some classifier has a minimal error.

Why bother with all relevant feature selection? When you try to understand the phenomenon that made your data, you should care about all factors that contribute to it, not just the bluntest signs of it in context of your methodology (yes, minimal optimal set of features by definition depends on your classifier choice).

What's different in BorutaPy?

It is the original R package recoded in Python with a few added extra features. Some improvements include:

- Faster run times, thanks to scikit-learn
- Scikit-learn like interface
- Compatible with any ensemble method from scikit-learn
- Automatic n_estimator selection
- Ranking of features

For more details, please check the top of the docstring.

We highly recommend using pruned trees with a depth between 3-7.

Also, after playing around a lot with the original code I identified a few areas where the core algorithm could be improved/altered to make it less strict and more applicable to biological data, where the Bonferroni correction might be overly harsh.

Percentile as threshold
The original method uses the maximum of the shadow features as a threshold in deciding which real feature is doing better than the shadow ones. This could be overly harsh.

To control this, I added the perc parameter, which sets the percentile of the shadow features' importances, the algorithm uses as the threshold. The default of 100 which is equivalent to taking the maximum as the R version of Boruta does, but it could be relaxed. Note, since this is the percentile, it changes with the size of the dataset. With several thousands of features it isn't as stringent as with a few dozens at the end of a Boruta run.

Two step correction for multiple testing
The correction for multiple testing was relaxed by making it a two step process, rather than a harsh one step Bonferroni correction.

We need to correct firstly because in each iteration we test a number of features against the null hypothesis (does a feature perform better than expected by random). For this the Bonferroni correction is used in the original code which is known to be too stringent in such scenarios (at least for biological data), and also the original code corrects for n features, even if we are in the 50th iteration where we only have k<<n features left. For this reason the first step of correction is the widely used Benjamini Hochberg FDR.

Following that however we also need to account for the fact that we have been testing the same features over and over again in each iteration with the same test. For this scenario the Bonferroni is perfect, so it is applied by deviding the p-value threshold with the current iteration index.

If this two step correction is not required, the two_step parameter has to be set to False, then (with perc=100) BorutaPy behaves exactly as the R version.

Parameters

estimator : object

> A supervised learning estimator, with a 'fit' method that returns the feature_importances_ attribute. Important features must correspond to high absolute values in the feature_importances_.

n_estimators : int or string, default = 1000

> If int sets the number of estimators in the chosen ensemble method. If 'auto' this is determined automatically based on the size of the dataset. The other parameters of the used estimators need to be set with initialisation.

perc : int, default = 100

> Instead of the max we use the percentile defined by the user, to pick our threshold for comparison between shadow and real features. The max tend to be too stringent. This provides a finer control over this. The lower perc is the more false positives will be picked as relevant but also the less relevant features will be left out. The usual trade-off. The default is essentially the vanilla Boruta corresponding to the max.

alpha : float, default = 0.05

> Level at which the corrected p-values will get rejected in both correction steps.

two_step : Boolean, default = True

> If you want to use the original implementation of Boruta with Bonferroni correction only set this to False.

max_iter : int, default = 100

> The number of maximum iterations to perform.

verbose : int, default=0

> Controls verbosity of output.

Attributes

n_features_ : int

> The number of selected features.

support_ : array of shape [n_features]

> The mask of selected features - only confirmed ones are True.

support_weak_ : array of shape [n_features]

> The mask of selected tentative features, which haven't gained enough support during the max_iter number of iterations..

ranking_ : array of shape [n_features]

> The feature ranking, such that `ranking_[i]` corresponds to the ranking position of the i-th feature. Selected (i.e., estimated best) features are assigned rank 1 and tentative features are assigned rank 2.

Examples

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from boruta import BorutaPy

# load X and y
# NOTE BorutaPy accepts numpy arrays only, hence the .values attribute
X = pd.read_csv('examples/test_X.csv', index_col=0).values
y = pd.read_csv('examples/test_y.csv', header=None, index_col=0).values
y = y.ravel()

# define random forest classifier, with utilising all cores and
# sampling in proportion to y labels
rf = RandomForestClassifier(n_jobs=-1, class_weight='balanced', max_depth=5)

# define Boruta feature selection method
feat_selector = BorutaPy(rf, n_estimators='auto', verbose=2, random_state=1)

# find all relevant features - 5 features should be selected
feat_selector.fit(X, y)

# check selected features - first 5 features are selected
feat_selector.support_

# check ranking of features
feat_selector.ranking_
```

```
# call transform() on X to filter it down to selected features
X_filtered = feat_selector.transform(X)
```

References

1. Kursa M., Rudnicki W., "Feature Selection with the Boruta Package" Journal of Statistical Software, Vol. 36, Issue 11, Sep 2010