

Feature Importance Permutation

A function to estimate the feature importance of classifiers and regressors based on *permutation importance*.

```
from mlxtend.evaluate import feature_importance_permutation
```

Overview

The *permutation importance* is an intuitive, model-agnostic method to estimate the feature importance for classifier and regression models. The approach is relatively simple and straightforward:

1. Take a model that was fit to the training dataset
2. Estimate the predictive performance of the model on an independent dataset (e.g., validation dataset) and record it as the baseline performance
3. For each feature i :
4. randomly permute feature column i in the original dataset
5. record the predictive performance of the model on the dataset with the permuted column
6. compute the feature importance as the difference between the baseline performance (step 2) and the performance on the permuted dataset

Permutation importance is generally considered as a relatively efficient technique that works well in practice [1], while a drawback is that the importance of correlated features may be overestimated [2].

References

- [1] Terence Parr, Kerem Turgutlu, Christopher Csiszar, and Jeremy Howard. *Beware Default Random Forest Importances* (<http://parrt.cs.usfca.edu/doc/rf-importance/index.html>)
- [2] Strobl, C., Boulesteix, A. L., Kneib, T., Augustin, T., & Zeileis, A. (2008). Conditional variable importance for random forests. *BMC bioinformatics*, 9(1), 307.

Example 1 -- Feature Importance for Classifiers

The following example illustrates the feature importance estimation via permutation importance based for classification models.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from mlxtend.evaluate import feature_importance_permutation
```

Generate a toy dataset

```

from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier

# Build a classification task using 3 informative features
X, y = make_classification(n_samples=10000,
                           n_features=10,
                           n_informative=3,
                           n_redundant=0,
                           n_repeated=0,
                           n_classes=2,
                           random_state=0,
                           shuffle=False)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1, stratify=y)

```

Feature importance via random forest

First, we compute the feature importance directly from the random forest via *mean impurity decrease* (described after the code section):

```

forest = RandomForestClassifier(n_estimators=250,
                               random_state=0)

forest.fit(X_train, y_train)

print('Training accuracy:', np.mean(forest.predict(X_train) == y_train)*100)
print('Test accuracy:', np.mean(forest.predict(X_test) == y_test)*100)

importance_vals = forest.feature_importances_
print(importance_vals)

```

```

Training accuracy: 100.0
Test accuracy: 95.0666666667
[ 0.283357  0.30846795 0.24204291 0.02229767 0.02364941 0.02390578
 0.02501543 0.0234225  0.02370816 0.0241332 ]

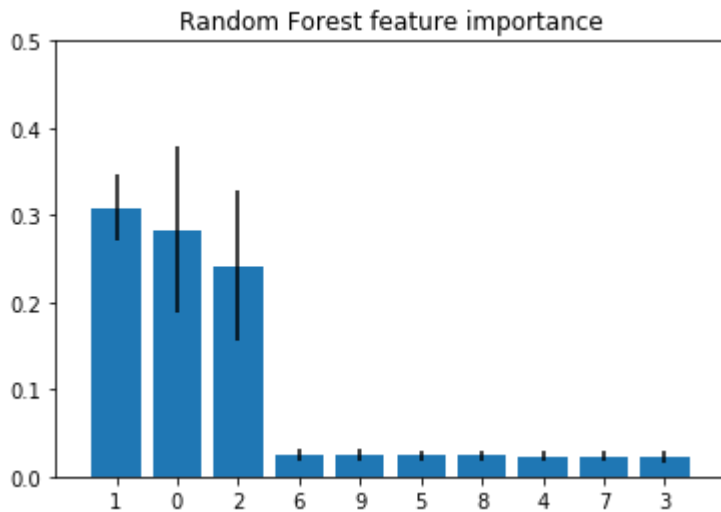
```

There are several strategies for computing the feature importance in random forest. The method implemented in scikit-learn (used in the next code example) is based on the Breiman and Friedman's CART (Breiman, Friedman, "Classification and regression trees", 1984), the so-called *mean impurity decrease*. Here, the importance value of a features is computed by averaging the impurity decrease for that feature, when splitting a parent node into two child nodes, across all the trees in the ensemble. Note that the impurity decrease values are weighted by the number of samples that are in the respective nodes. This process is repeated for all features in the dataset, and the feature importance values are then normalized so that they sum up to 1. In CART, the authors also note that this fast way of computing feature importance values is relatively consistent with the permutation importance.

Next, let's visualize the feature importance values from the random forest including a measure of the *mean impurity decrease* variability (here: standard deviation):

```
std = np.std([tree.feature_importances_ for tree in forest.estimators_],
             axis=0)
indices = np.argsort(importance_vals)[::-1]

# Plot the feature importances of the forest
plt.figure()
plt.title("Random Forest feature importance")
plt.bar(range(X.shape[1]), importance_vals[indices],
        yerr=std[indices], align="center")
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.ylim([0, 0.5])
plt.show()
```



As we can see, the features 1, 0, and 2 are estimated to be the most informative ones for the random forest classifier. Next, let's compute the feature importance via the permutation importance approach.

Permutation Importance

```
imp_vals, _ = feature_importance_permutation(
    predict_method=forest.predict,
    X=X_test,
    y=y_test,
    metric='accuracy',
    num_rounds=1,
    seed=1)
```

imp_vals

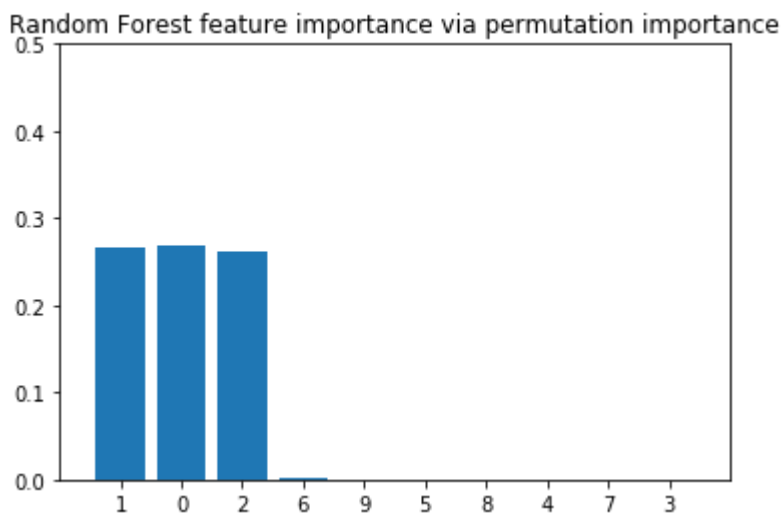
```
array([ 0.26833333,  0.26733333,  0.261        , -0.002        , -0.00033333,
        0.00066667,  0.00233333,  0.00066667,  0.00066667, -0.00233333])
```

Note that the `feature_importance_permutation` returns two arrays. The first array (here: `imp_vals`) contains the actual importance values we are interested in. If `num_rounds > 1`, the permutation is repeated multiple times (with different random seeds), and in this case the first

array contains the average value of the importance computed from the different runs. The second array (here, assigned to `_`, because we are not using it) then contains all individual values from these runs (more about that later).

Now, let's also visualize the importance values in a barplot:

```
indices = np.argsort(imp_vals)[::-1]
plt.figure()
plt.title("Random Forest feature importance via permutation importance")
plt.bar(range(X.shape[1]), imp_vals[indices])
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.ylim([0, 0.5])
plt.show()
```



As we can see, also here, features 1, 0, and 2 are predicted to be the most important ones, which is consistent with the feature importance values that we computed via the *mean impurity decrease* method earlier.

(Note that in the context of random forests, the feature importance via permutation importance is typically computed using the out-of-bag samples of a random forest, whereas in this implementation, an independent dataset is used.)

Previously, it was mentioned that the permutation is repeated multiple times if `num_rounds > 1`. In this case, the second array returned by the `feature_importance_permutation` contains the importance values for these individual runs

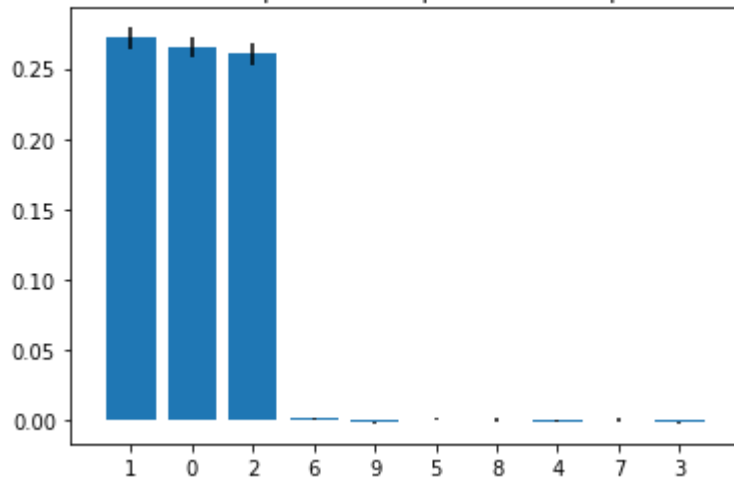
(the array has shape [num_features, num_rounds]), which we can use to compute some sort of variability between these runs.

```
imp_vals, imp_all = feature_importance_permutation(
    predict_method=forest.predict,
    X=X_test,
    y=y_test,
    metric='accuracy',
    num_rounds=10,
    seed=1)

std = np.std(imp_all, axis=1)
indices = np.argsort(imp_vals)[::-1]

plt.figure()
plt.title("Random Forest feature importance via permutation importance w. std. dev.")
plt.bar(range(X.shape[1]), imp_vals[indices],
        yerr=std[indices])
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.show()
```

Random Forest feature importance via permutation importance w. std. dev.



It shall be noted that the feature importance values do not sum up to one, since they are not normalized (you can normalize them if you'd like, by dividing these by the sum of importance values). Here, the main point is to look at the importance values relative to each other and not to over-interpret the absolute values.

Support Vector Machines

While the permutation importance approach yields results that are generally consistent with the *mean impurity decrease* feature importance values from a random forest, it's a method that is model-agnostic and can be used with any kind of classifier or regressor. The example below applies the `feature_importance_permutation` function to a support vector machine:

```
from sklearn.svm import SVC

svm = SVC(C=1.0, kernel='rbf')
svm.fit(X_train, y_train)

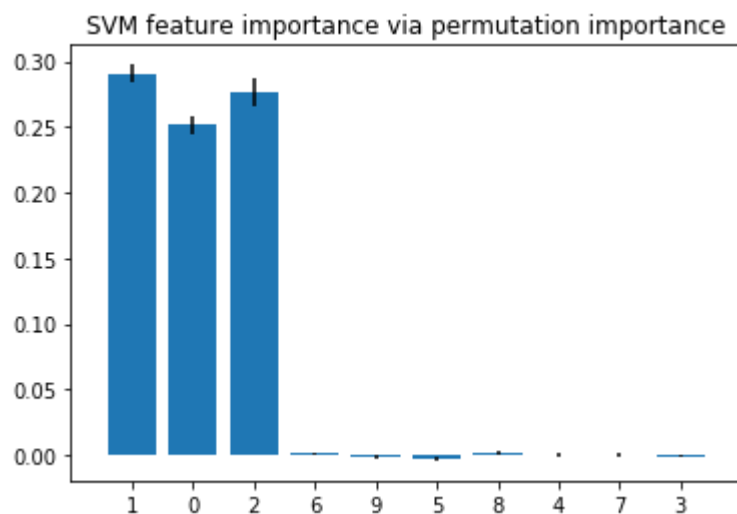
print('Training accuracy', np.mean(svm.predict(X_train) == y_train)*100)
print('Test accuracy', np.mean(svm.predict(X_test) == y_test)*100)
```

```
Training accuracy 95.0857142857
Test accuracy 94.9666666667
```

```
imp_vals, imp_all = feature_importance_permutation(
    predict_method=svm.predict,
    X=X_test,
    y=y_test,
    metric='accuracy',
    num_rounds=10,
    seed=1)

std = np.std(imp_all, axis=1)
indices = np.argsort(imp_vals)[::-1]

plt.figure()
plt.title("SVM feature importance via permutation importance")
plt.bar(range(X.shape[1]), imp_vals[indices],
        yerr=std[indices])
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.show()
```

Example 1 -- Feature Importance for Regressors

```
import numpy as np
import matplotlib.pyplot as plt
from mlxtend.evaluate import feature_importance_permutation
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression
from sklearn.svm import SVR
```

```
X, y = make_regression(n_samples=1000,
                      n_features=5,
                      n_informative=2,
                      n_targets=1,
                      random_state=123,
                      shuffle=False)
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=123)
```

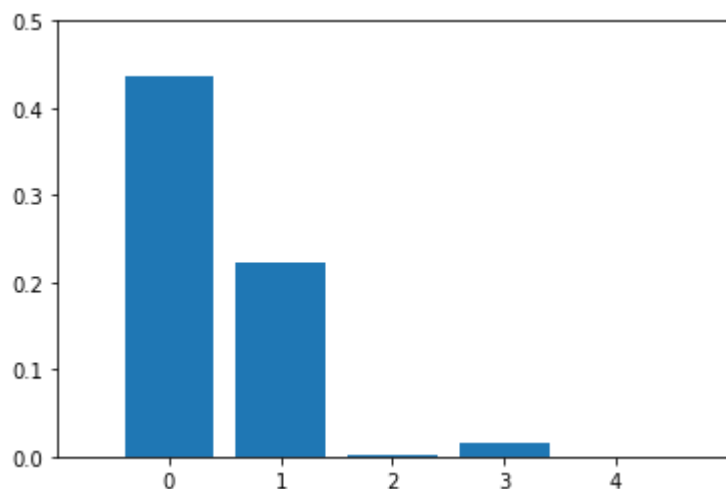
```
svm = SVR(kernel='rbf')
svm.fit(X_train, y_train)
```

```
imp_vals, _ = feature_importance_permutation(
    predict_method=svm.predict,
    X=X_test,
    y=y_test,
    metric='r2',
    num_rounds=1,
    seed=1)
```

```
imp_vals
```

```
array([ 0.43676245,  0.22231268,  0.00146906,  0.01611528, -0.00522067])
```

```
plt.figure()
plt.bar(range(X.shape[1]), imp_vals)
plt.xticks(range(X.shape[1]))
plt.xlim([-1, X.shape[1]])
plt.ylim([0, 0.5])
plt.show()
```



API

feature_importance_permutation(X, y, predict_method, metric, num_rounds=1, seed=None)

Feature importance imputation via permutation importance

Parameters

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

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

- **y** : NumPy array, shape = [n_samples]

Target values.

- **predict_method** : prediction function

A callable function that predicts the target values from X.

- **metric** : str, callable

The metric for evaluating the feature importance through permutation. By default, the strings 'accuracy' is recommended for classifiers and the string 'r2' is recommended for regressors. Optionally, a custom scoring function (e.g., `metric=scoring_func`) that accepts two arguments, `y_true` and `y_pred`, which have similar shape to the `y` array.

- `num_rounds` : int (default=1)

Number of rounds the feature columns are permuted to compute the permutation importance.

- `seed` : int or None (default=None)

Random seed for permuting the feature columns.

Returns

- `mean_importance_vals`, `all_importance_vals` : NumPy arrays.

The first array, `mean_importance_vals` has shape `[n_features,]` and contains the importance values for all features. The shape of the second array is `[n_features, num_rounds]` and contains the feature importance for each repetition. If `num_rounds=1`, it contains the same values as the first array, `mean_importance_vals`.

Examples

For usage examples, please see

http://rasbt.github.io/mlxtend/user_guide/evaluate/feature_importance_permutation/
(http://rasbt.github.io/mlxtend/user_guide/evaluate/feature_importance_permutation/)

Copyright © 2014-2019 Sebastian Raschka (<http://sebastianraschka.com>)

Documentation built with MkDocs (<http://www.mkdocs.org/>).