Applying Filter Methods in Python for Feature Selection

By Usman Malik • October 30, 2018 • 3 Comments

Introduction

Machine learning and deep learning algorithms learn from data, which consists of different types of features. The training time and performance of a machine learning algorithm depends heavily on the features in the dataset. Ideally, we should only retain those features in the dataset that actually help our machine learning model learn something.

Unnecessary and redundant features not only slow down the training time of an algorithm, but they also affect the performance of the algorithm. The process of selecting the most suitable features for training the machine learning model is called "feature selection".

There are several advantages of performing feature selection before training machine learning models, some of which have been enlisted below:

- Models with less number of features have higher explainability
- It is easier to implement machine learning models with reduced features
- Fewer features lead to enhanced generalization which in turn reduces overfitting
- Feature selection removes data redundancy
- Training time of models with fewer features is significantly lower
- Models with fewer features are less prone to errors

Several methods have been developed to select the most optimal features for a machine learning algorithm. One category of such methods is called filter methods. In this article, we will study some of the basic filter methods for feature selection.

Filter Methods for Feature Selection

Filters methods belong to the category of feature selection methods that select features independently of the machine learning algorithm model. This is one of the biggest advantages of filter methods. Features selected using filter methods can be used as an input to any machine learning models. Another advantage of filter methods is that they are very fast. Filter methods are generally the first step in any feature selection pipeline.

Filter methods can be broadly categorized into two categories: **Univariate Filter Methods** and **Multivariate filter methods**.

The univariate filter methods are the type of methods where individual features are ranked according to specific criteria. The top N features are then selected. Different types of ranking criteria are used for univariate filter methods, for example fisher score, mutual information, and variance of the feature.

One of the major disadvantage of univariate filter methods is that they may select redundant features because the relationship between individual features is not taken into account while making decisions. Univariate filter methods are ideal for removing constant and quasi-constant features from the data.

Multivariate filter methods are capable of removing redundant features from the data since they take the mutual relationship between the features into account. Multivariate filter methods can be used to remove duplicate and correlated features from the data.

In this article, we will see how we can remove constant, quasi-constant, duplicate, and correlated features from our dataset with the help of Python.

Removing Constant features

Constant features are the type of features that contain only one value for all the outputs in the dataset. Constant features provide no information that can help in classification of the record at hand. Therefore, it is advisable to remove all the constant features from the dataset.

Let's see how we can remove constant features from a dataset. The dataset that we are going to use for this example is the Santandar Customer Satisfaction dataset, that can be downloaded from Kaggle. We will use the file "train.csv". However, I have renamed it to "santandar_data.csv" for readability purpose.

Importing Required Libraries and Dataset

Constant features have values with zero variance since all the values are the same. We can find the constant columns using the `VarianceThreshold` function of Python's Scikit Learn Library. Execute the following script to import the required libraries and the dataset:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import VarianceThreshold

santandar_data = pd.read_csv(r"E:\Datasets\santandar_data.csv", nrows=40000)
santandar_data.shape
```

I filtered the top 40 thousand records. In the output, you should see (40000, 371) which means that we have 40 thousand rows and 371 columns in our dataset.

Splitting Data Into Training and Test Sets

It is important to mention here that, in order to avoid overfitting, feature selection should only be applied to the training set. Let's divide our data into training and test sets. Execute the following script:

```
train_features, test_features, train_labels, test_labels=train_test_split(
    santandar_data.drop(labels=['TARGET'], axis=1),
    santandar_data['TARGET'],
    test_size=0.2,
    random_state=41)
```

Removing Constant Features using Variance Threshold

Now is the time to remove constant features. To do so we will use `VarianceThreshold` function that we imported earlier. The function requires a value for its `threshold` parameter. Passing a value of zero for the parameter will filter all the features with zero variance. Execute the following script to create a filter for constant features.

```
constant_filter = VarianceThreshold(threshold=0)
```

Next, we need to simply apply this filter to our training set as shown in the following example:

```
constant_filter.fit(train_features)
```

Now to get all the features that are not constant, we can use the `get_support()` method of the filter that we created. Execute the following script to see the number of non-constant features.

```
len(train_features.columns[constant_filter.get_support()])
```

In the output, you should see 320, which means that out of 370 features in the training set 320 features are not constant.

Similarly, you can find the number of constant features with the help of the following script:

```
constant_columns = [column for column in train_features.columns
                    if column not in train_features.columns[constant_filter.get_support()]]

print(len(constant_columns))
```

To see all the constant columns, execute the following script:

```
for column in constant_columns:
    print(column)
```

The outpu looks likes this:

```
ind_var2_0
ind_var2
ind_var18_0
ind_var18
ind_var27_0
ind_var28_0
ind_var28
ind_var27
ind_var34_0
ind_var34
ind_var41
ind_var46_0
ind_var46
num_var18_0
num_var18
num_var27_0
num_var28_0
num_var28
num_var27
num_var34_0
num_var34
num_var41
num_var46_0
num_var46
saldo_var18
saldo_var28
saldo_var27
saldo_var34
saldo_var41
saldo_var46
delta_imp_amort_var18_1y3
delta_imp_amort_var34_1y3
imp_amort_var18_hace3
imp_amort_var18_ult1
imp_amort_var34_hace3
imp_amort_var34_ult1
imp_reemb_var13_hace3
imp_reemb_var17_hace3
imp_reemb_var33_hace3
imp_trasp_var17_out_hace3
imp_trasp_var33_out_hace3
num_var2_0_ult1
num_var2_ult1
num_reemb_var13_hace3
num_reemb_var17_hace3
num_reemb_var33_hace3
num_trasp_var17_out_hace3
num_trasp_var33_out_hace3
saldo_var2_ult1
saldo_medio_var13_medio_hace3
```

Finally, to remove constant features from training and test sets, we can use the `transform()` method of the `constant_filter`. Execute the following script to do so:

```
train_features = constant_filter.transform(train_features)
test_features = constant_filter.transform(test_features)

train_features.shape, test_features.shape
```

If you execute the above script, you will see that both our training and test sets will now contain 320 columns, since the 50 constant columns have been removed.

Removing Quasi-Constant features

Quasi-constant features, as the name suggests, are the features that are almost constant. In other words, these features have the same values for a very large subset of the outputs. Such features are not very useful for making predictions. There is no rule as to what should be the threshold for the variance of quasi-constant features. However, as a rule of thumb, remove those quasi-constant features that have more than 99% similar values for the output observations.

In this section, we will create a quasi-constant filter with the help of `VarianceThreshold` function. However, instead of passing 0 as the value for the `threshold` parameter, we will pass 0.01, which means that if the variance of the values in a column is less than 0.01, remove that column. In other words, remove feature column where approximately 99% of the values are similar.

The steps are quite similar to the previous section. We will import the dataset and libraries, will perform train-test split and will remove the constant features first.

Importing Required Libraries and Dataset

Execute the following script to import the dataset and desired libraries:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import VarianceThreshold

santandar_data = pd.read_csv(r"E:\Datasets\santandar_data.csv", nrows=40000)
santandar_data.shape
```

Splitting Data Into Training and Test Sets

```
train_features, test_features, train_labels, test_labels = train_test_split(
    santandar_data.drop(labels=['TARGET'], axis=1),
    santandar_data['TARGET'],
    test_size=0.2,
    random_state=41)
```

Removing Constant Features using Variance Threshold

Before we can remove quasi-constant features, we should first remove the constant features. Execute the following script to do so:

```
constant_filter = VarianceThreshold(threshold=0)
constant_filter.fit(train_features)

len(train_features.columns[constant_filter.get_support()])

constant_columns = [column for column in train_features.columns
                    if column not in train_features.columns[constant_filter.get_support()]]

train_features.drop(labels=constant_columns, axis=1, inplace=True)
test_features.drop(labels=constant_columns, axis=1, inplace=True)
```

Removing Quasi-Constant Features Using Variance Threshold

Let's create our quasi-constant filter. Execute the following script to do so:

```
qconstant_filter = VarianceThreshold(threshold=0.01)
```

The rest of the steps are the same. We need to apply the filter to our training set using `fit()` method as shown below.

```
qconstant_filter.fit(train_features)
```

Let's check the number of our non-quasi-constant columns. Execute the following script:

```
len(train_features.columns[qconstant_filter.get_support()])
```

In the output, you should see 265 which means that out of 320 columns that we achieved after removing constant features, 55 are quasi-constant.

To verify the number of quasi-constant columns, execute the following script:

```
qconstant_columns = [column for column in train_features.columns
                    if column not in train_features.columns[qconstant_filter.get_support()]]

print(len(qconstant_columns))
```

You should see 55 in the output.

Let's now print the names of all the quasi-constant columns. Execute the following script:

```
for column in qconstant_columns:
    print(column)
```

In the output, you should see the following column names:

```
ind_var1
ind_var6_0
ind_var6
ind_var13_largo
ind_var13_medio_0
ind_var13_medio
ind_var14
ind_var17_0
ind_var17
ind_var19
ind_var20_0
ind_var20
ind_var29_0
ind_var29
ind_var30_0
ind_var31_0
ind_var31
ind_var32_cte
ind_var32_0
ind_var32
ind_var33_0
ind_var33
ind_var40
ind_var39
ind_var44_0
ind_var44
num_var6_0
num_var6
num_var13_medio_0
num_var13_medio
num_op_var40_hace3
num_var29_0
num_var29
delta_imp_aport_var33_1y3
delta_num_aport_var33_1y3
ind_var7_emit_ult1
ind_var7_recib_ult1
num_aport_var33_hace3
num_aport_var33_ult1
num_var7_emit_ult1
num_meses_var13_medio_ult3
num_meses_var17_ult3
num_meses_var29_ult3
num_meses_var33_ult3
num_meses_var44_ult3
num_reemb_var13_ult1
num_reemb_var17_ult1
num_reemb_var33_ult1
num_trasp_var17_in_hace3
num_trasp_var17_in_ult1
num_trasp_var17_out_ult1
num_trasp_var33_in_hace3
num_trasp_var33_in_ult1
num_trasp_var33_out_ult1
num_venta_var44_hace3
```

Finally, to see if our training and test sets only contains the non-constant and non-quasi-constant columns, we can use the `transform()` method of the `qconstant_filter`. Execute the following script to do so:

```
train_features = qconstant_filter.transform(train_features)
test_features = qconstant_filter.transform(test_features)

train_features.shape, test_features.shape
```


If you execute the above script, you will see that both our training and test sets will now contain 265 columns, since the 50 constant, and 55 quasi-constant columns have been removed from a total of default 370 columns.

Removing Duplicate Features

Duplicate features are the features that have similar values. Duplicate features do not add any value to algorithm training, rather they add overhead and unnecessary delay to the training time. Therefore, it is always recommended to remove the duplicate features from the dataset before training.

Importing Required Libraries and Dataset

Execute the following script to import the dataset and desired libraries:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import VarianceThreshold

santandar_data = pd.read_csv(r"E:\Datasets\santandar_data.csv", nrows=20000)
santandar_data.shape
```

Removing duplicate columns can be computationally costly since we have to take the transpose of the data matrix before we can remove duplicate features. Therefore, in the above script, we only import the first 20 thousand records from the santandar customer satisfaction data that we have been using in this article.

Splitting Data Into Training and Test Sets

```
train_features, test_features, train_labels, test_labels = train_test_split(
    santandar_data.drop(labels=['TARGET'], axis=1),
    santandar_data['TARGET'],
    test_size=0.2,
    random_state=41)
```

Removing Duplicate Features using Transpose

Unlike constant and quasi-constant features, we have no built-in Python method that can remove duplicate features. However, we have a method that can help us identify duplicate rows in a pandas dataframe. We will use this method to first take a transpose of our dataset as shown below:

```
train_features_T = train_features.T
train_features_T.shape
```

In the script above we take the transpose of our training data and store it in the `train_features_T` dataframe. Our initial training set contains 16000 rows and 370 columns, if you take a look at the shape of the transposed training set, you will see that it contains 370 rows and 16000 columns.

Luckily, in pandas we have `duplicated()` method which can help us find duplicate rows from the dataframe. Remember, the rows of the transposed dataframe are actually the columns or the features of the actual dataframe.

Let's find the total number of duplicate features in our dataset using the `sum()` method, chained with the `duplicated()` method as shown below.

```
print(train_features_T.duplicated().sum())
```

In the output, you should see 94.

Finally, we can drop the duplicate rows using the `drop_duplicates()` method. If you pass the string value `first` to the `keep` parameter of the `drop_duplicates()` method, all the duplicate rows will be dropped except the first copy. In the next step we will remove all the duplicate rows and will take transpose of the transposed training set to get the original training set that doesn't contain any duplicate column. Execute the following script:

```
unique_features = train_features_T.drop_duplicates(keep='first').T
```

Now, let's print the shape of our new training set without duplicate features:

```
unique_features.shape
```

In the output, you should see (16000,276), you can see that after removing 94 duplicate columns, the size of our feature set has significantly reduced.

To see the names of the duplicate columns, execute this script:


```
duplicated_features = [dup_col for dup_col in train_features.columns if dup_col not in unique_features.columns]
duplicated_features
```

In the output, you should see the following columns:

```
['ind_var2',
 'ind_var13_medio',
 'ind_var18_0',
 'ind_var18',
 'ind_var26',
 'ind_var25',
 'ind_var27_0',
 'ind_var28_0',
 'ind_var28',
 'ind_var27',
 'ind_var29_0',
 'ind_var29',
 'ind_var32',
 'ind_var34_0',
 'ind_var34',
 'ind_var37',
 'ind_var40_0',
 'ind_var40',
 'ind_var41',
 'ind_var39',
 'ind_var46_0',
 'ind_var46',
 'num_var13_medio',
 'num_var18_0',
 'num_var18',
 'num_var26',
 'num_var25',
 'num_op_var40_hace3',
 'num_op_var39_hace3',
 'num_var27_0',
 'num_var28_0',
 'num_var28',
 'num_var27',
 'num_var29_0',
 'num_var29',
 'num_var32',
 'num_var34_0',
 'num_var34',
 'num_var37',
 'num_var40_0',
 'num_var40',
 'num_var41',
 'num_var39',
 'num_var46_0',
 'num_var46',
 'saldo_var18',
 'saldo_var28',
 'saldo_var27',
 'saldo_var29',
 'saldo_var34',
 'saldo_var40',
 'saldo_var41',
 'saldo_var46',
 'delta_imp_amort_var18_1y3',
 'delta_imp_amort_var34_1y3',
 'delta_imp_reemb_var33_1y3',
 'delta_imp_trasp_var17_out_1y3',
 'delta_imp_trasp_var33_out_1y3',
 'delta_num_reemb_var13_1y3',
 'delta_num_reemb_var17_1y3',
 'delta_num_reemb_var33_1y3',
 'delta_num_trasp_var17_in_1y3',
 'delta_num_trasp_var17_out_1y3',
 'delta_num_trasp_var33_in_1y3',
 'delta_num_trasp_var33_out_1y3',
 'imp_amort_var18_hace3',
 'imp_amort_var18_ult1',
 'imp_amort_var34_hace3',
 'imp_amort_var34_ult1',
 'imp_var7_emit_ult1',
 'imp_reemb_var13_hace3',
 'imp_reemb_var17_hace3',
 'imp_reemb_var33_hace3',
 'imp_reemb_var33_ult1',
 'imp_trasp_var17_out_hace3',
 'imp_trasp_var17_out_ult1',
 'imp_trasp_var33_in_hace3',
 'imp_trasp_var33_out_hace3',
 'ind_var7_emit_ult1',
 'num_var2_0_ult1',
 'num_var2_ult1',
 'num_var7_emit_ult1',
 'num_reemb_var13_hace3',
 'num_reemb_var17_hace3',
 'num_reemb_var33_hace3',
 'num_reemb_var33_ult1',
 'num_trasp_var17_out_hace3',
 'num_trasp_var17_out_ult1',
 'num_trasp_var33_in_hace3',
 'num_trasp_var33_out_hace3',
 'saldo_var2_ult1',
 'saldo_medio_var13_medio_hace3',
 'saldo_medio_var13_medio_ult1',
 'saldo_medio_var29_hace3']
```

Removing Correlated Features

In addition to the duplicate features, a dataset can also contain correlated features. Two or more than two features are correlated if they are close to each other in the linear space.

Take the example of the feature set for a fruit basket, the weight of the fruit basket is normally correlated with the price. The more the weight, the higher the price.

Correlation between the output observations and the input features is very important and such features should be retained. However, if two or more than two features are mutually correlated, they convey redundant information to the model and hence only one of the correlated features should be retained to reduce the number of features.

The dataset we are going to be used for this section is the BNP Paribas Cardif Claims Management dataset, that can be downloaded from Kaggle. Follow these steps to find and remove the correlated features from the dataset.

Importing Required Libraries and Dataset

Execute the following script to import the dataset and desired libraries:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import VarianceThreshold

paribas_data = pd.read_csv(r"E:\Datasets\paribas_data.csv", nrows=20000)
paribas_data.shape
```

In the script above, I imported the dataset along with the required libraries. Next, we printed the shape of our dataframe. In the output, you should see (20000, 133) which means that our dataset contains 20 thousand rows and 133 features.

To find the correlation, we only need the numerical features in our dataset. In order to filter out all the

features, except the numeric ones, we need to preprocess our data.

Data Preprocessing

Execute the following script, to remove non-numeric features from the dataset.

```
num_colums = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
numerical_columns = list(paribas_data.select_dtypes(include=num_colums).columns)
paribas_data = paribas_data[numerical_columns]
```

In the first line of the script above, we define a list that contains the data types of the columns that we want to retain in our dataset. Next, we call the `select_dtypes()` method on our dataset and pass it the `num_colums` list that contains the type of columns that we want to retain. The `select_dtypes()` method will return the names of the specified numeric columns, which we store in the list `numeric_columns`. Next, we filter our columns from `paribas_data` dataframe with the help of the `numerical_colums` list. Let's print the shape of the `paribas_data` dataframe to see how many numeric columns do we have, execute the following script:

```
paribas_data.shape
```

In the output, you should see (20000, 114) which means that now our dataset contains 20 thousand records and 114 features. Remember, previously we had 133 features.

Splitting Data Into Training and Test Sets

As usual, we need to split our data into training and testing set before removing any correlated features, execute the following script to divide the data into training and test sets:

```
train_features, test_features, train_labels, test_labels = train_test_split(
    paribas_data.drop(labels=['target', 'ID'], axis=1),
    paribas_data['target'],
    test_size=0.2,
    random_state=41)
```

In the above script, we divide our data into 80% training and 20% test set.

Removing Correlated Features using corr() Method

To remove the correlated features, we can make use of the `corr()` method of the pandas dataframe. The `corr()` method returns a correlation matrix containing correlation between all the columns of the dataframe. We can then loop through the correlation matrix and see if the correlation between two columns is greater than threshold correlation, add that column to the set of correlated columns. We can remove that set of columns from the actual dataset.

Let's first create correlation matrix for the columns in the dataset and an empty set that will contain all the correlated features. Execute the following script to do so:

```
correlated_features = set()
correlation_matrix = paribas_data.corr()
```

In the script above, we create correlation matrix `correlation_matrix` for all the columns in our dataset. We also created a set `correlated_features` which will contain names of all the correlated features.

Next, we will loop through all the columns in the `correlation_matrix` and will add the columns with a correlation value of 0.8 to the `correlated_features` set as shown below. You can set any threshold value for the correlation.

```
for i in range(len(correlation_matrix .columns)):
    for j in range(i):
        if abs(correlation_matrix.iloc[i, j]) > 0.8:
            colname = correlation_matrix.columns[i]
            correlated_features.add(colname)
```

Let's see the total number of columns in our dataset, with correlation value of greater than 0.8 with at least 1 other column. Execute the following script:

```
len(correlated_features)
```

You should see 55 in the output, which is almost 40% of the original features in the dataset. You can see how much redundant information does our dataset contain. Execute the following script to see the names of these features:

```
print(correlated_features)
```

The output looks like this:

```
{'v55', 'v105', 'v130', 'v12', 'v60', 'v67', 'v63', 'v46', 'v53', 'v43', 'v68', 'v123', 'v95', 'v103', 'v44', 'v
108', 'v89', 'v104', 'v109', 'v83', 'v115', 'v21', 'v101', 'v93', 'v40', 'v78', 'v54', 'v118', 'v124', 'v73', 'v
96', 'v121', 'v77', 'v114', 'v48', 'v116', 'v87', 'v86', 'v65', 'v122', 'v64', 'v81', 'v128', 'v49', 'v37', 'v8
4', 'v98', 'v111', 'v41', 'v25', 'v106', 'v32', 'v126', 'v76', 'v100'}
```

The names of the features have been masked by the bank since they contain sensitive information, however, you can see the code names for the features. These correlated columns convey similar information to the learning algorithm and therefore, should be removed.

The following script removes these columns from the dataset:

```
train_features.drop(labels=correlated_features, axis=1, inplace=True)
test_features.drop(labels=correlated_features, axis=1, inplace=True)
```

Conclusion

Feature selection plays a vital role in the performance and training of any machine learning model. Different types of methods have been proposed for feature selection for machine learning algorithms. In this article, we studied different types of filter methods for feature selection using Python.

We started our discussion by removing constant and quasi-constant features followed by removing duplicate features. Finally, we studied how to remove correlated features from our dataset.

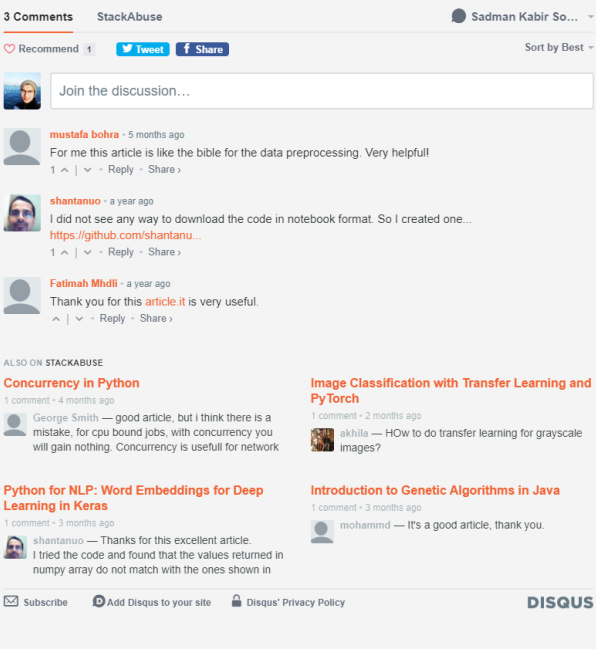
In the next article, we will take a look at some of the other types of feature selection methods. Till then, happy coding!

python, machine learning

About Usman Malik

Paris (France) Twitter

Programmer | Blogger | Data Science Enthusiast | PhD To Be | Arsenal FC for Life



‹ Previous Post : Guide to Spring Data JPA

Next Post : Java's Object Methods: clone() ›