Feature selection using Wrapper methods in Python

vikashraj luhaniwal Follow
Oct 3, 2019 · 7 min read ★

In today's era of Big data and IoT, we are easily loaded with rich datasets having extremely *high dimensions*. In order to perform any machine learning task or to get insights from such *high dimensional* data, **feature selection** becomes very important. Since some features may be irrelevant or less significant to the dependent variable so their unnecessary inclusion to the model leads to

- Increase in **complexity** of a model and makes it **harder** to **interpret**.
- Increase in **time complexity** for a model to get trained.
- Result in a **dumb model** with inaccurate or less reliable predictions.

Hence, it gives an indispensable need to perform *feature selection*. **Feature selection** is very crucial and must component in machine learning and data science workflows especially while dealing with *high dimensional* datasets.

What is Feature selection?

As the name suggests, it is a process of selecting the most *significant* and *relevant*features from a vast set of features in the given dataset.

For a dataset with ***d*** input features, the ***feature selection*** process results in ***k*** featuressuch that $k < d$, where k is the smallest set of *significant* and *relevant* features.

So ***feature selection*** helps in finding the smallest set of features which results in

- **Training** a machine learning algorithm **faster**.
- Reducing the **complexity** of a model and making it easier to **interpret**.
- Building a **sensible model** with **better prediction power**.
- **Reducing overfitting** by selecting the right set of features.

Feature selection methods

For a dataset with **d features**, if we apply hit and trial method with all possible combinations of features then total **2^d — 1** models need to be evaluated for a *significant* set of features. It is a time-consuming approach, therefore, we use *feature selection* techniques to find out the smallest set of features more efficiently.

There are three types of *feature selection* techniques :

1. Filter methods
2. Wrapper methods
3. Embedded methods

Difference between Filter, Wrapper and Embedded methods

Filter methods	Wrapper methods	Embedded methods
Generic set of methods which do not incorporate a **specific machine learning algorithm.**	Evaluates on a **specific machine learning algorithm** to find optimal features.	Embeds (fix) features during **model building process.** Feature selection is done by observing each iteration of model training phase.
Much **faster** compared to Wrapper methods in terms of time complexity	**High computation time** for a dataset with many features	Sits **between Filter methods and Wrapper methods** in terms of time complexity
Less prone to **over-fitting**	High chances of **over-fitting** because it involves training of machine learning models with different combination of features	Generally used to reduce **over-fitting** by **penalizing** the coefficients of a model being too large.
Examples – **Correlation, Chi-Square test, ANOVA, Information gain** etc.	Examples - **Forward Selection, Backward elimination, Stepwise selection** etc.	Examples - **LASSO, Elastic Net, Ridge Regression** etc.

Filter vs. Wrapper vs. Embedded methods

In this post, we will only discuss *feature selection* using ***Wrapper methods*** in Python.

Wrapper methods

In *wrapper methods,* the *feature selection* process is based on a specific machine learning algorithm that we are trying to fit on a given dataset.

It follows a *greedy search approach* by evaluating all the possible combinations of features against the *evaluation criterion*. The *evaluation criterion* is simply the performance measure which depends on the type of problem, for eg. for *regression*evaluation criterion can be p-values, R-squared, Adjusted R-squared, similarly for *classification* the evaluation criterion can be accuracy, precision, recall, f1-score, etc. Finally, it selects the combination of features that gives the optimal results for the specified machine learning algorithm.

Flow chart — Wrapper methods

Most commonly used techniques under *wrapper methods* are:

1. Forward selection
2. Backward elimination
3. Bi-directional elimination(Stepwise Selection)

Too much theory so far. Now let us discuss *wrapper methods* with an example of ***Boston house prices dataset*** available in sklearn. The dataset contains 506 observations of 14 different features. The dataset can be imported using `load_boston()` function available in `sklearn.datasets` module.

```
from sklearn.datasets import load_boston
boston = load_boston()
print(boston.data.shape)          # for dataset dimension
print(boston.feature_names)       # for feature names
print(boston.target)              # for target variable
print(boston.DESCR)               # for data description
```

Let's convert this raw data into a *data frame* including target variable and actual data along with feature names.

```
import pandas as pd
bos = pd.DataFrame(boston.data, columns = boston.feature_names)
bos['Price'] = boston.target
X = bos.drop("Price", 1)       # feature matrix
y = bos['Price']               # target feature
bos.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	Price
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Here, the target variable is `Price`. We will be fitting a *regression model* to predict `Price` by selecting optimal features through *wrapper methods*.

1. Forward selection

In *forward selection*, we start with a null model and then start fitting the model with each individual feature one at a time and select the feature with the minimum *p-value*. Now fit a model with two features by trying combinations of the earlier selected feature with all other remaining features. Again select the feature with the minimum *p-value*. Now fit a model with three features by trying combinations of two previously selected features with other remaining features. Repeat this process until we have a set of selected features with a *p-value* of individual feature less than the *significance level*.

In short, the steps for ***forward selection*** technique are as follows :

1. Choose a *significance level* (e.g. SL = 0.05 with a 95% confidence).
2. Fit all possible *simple regression models* by considering one feature at a time.
   Total **'n'**models are possible. Select the feature with the lowest *p-value*.

3. Fit all possible models with one extra feature added to the previously selected feature(s).

4. Again, select the feature with minimum *p-value*. if *p_value* < *significance level* then go to Step 3, otherwise terminate the process.

Now let us perform the same on `Boston house price` data.

forward selection code

This above function accepts `data` , `target` variable and `significance level` as arguments and returns the final list of *significant features* based on *p-values* through ***forward selection***.

```
forward_selection(X,y)
```

```
#OUTPUT
['LSTAT',
 'RM',
 'PTRATIO',
 'DIS',
 'NOX',
 'CHAS',
 'B',
```

```
    'ZN',
    'CRIM',
    'RAD',
    'TAX']
```

Implementing Forward selection using built-in functions in Python:

`mlxtend` library contains built-in implementation for most of the *wrapper methods* based feature selection techniques. `SequentialFeatureSelector()` function comes with various combinations of *feature selection* techniques.

```
#importing the necessary libraries
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from sklearn.linear_model import LinearRegression

# Sequential Forward Selection(sfs)
sfs = SFS(LinearRegression(),
           k_features=11,
           forward=True,
           floating=False,
           scoring = 'r2',
           cv = 0)
```

`SequentialFeatureSelector()` function accepts the following major arguments :

- `LinearRegression()` as an `estimator` for the entire process. Similarly, it can be any *classification* based algorithm.
- `k_features` indicates the number of features to be selected. It can be any random value, but the optimal value can be found by analyzing and visualizing the *scores* for different numbers of features.
- `forward` and `floating` arguments for different flavors of *wrapper methods*, here, `forward = True` and `floating = False` are for *forward selection* technique.
- `Scoring` argument specifies the *evaluation criterion* to be used. For *regression*problems, there is only `r2` *score* in default implementation. Similarly for *classification*, it can be accuracy, precision, recall, f1-score, etc.

- `cv` argument is for *k-fold cross-validation*.

Now let's fit the above-defined *feature selector* on `Boston house price` dataset.

```
sfs.fit(X, y)
sfs.k_feature_names_     # to get the final set of features

#OUTPUT

('CRIM',
 'ZN',
 'CHAS',
 'NOX',
 'RM',
 'DIS',
 'RAD',
 'TAX',
 'PTRATIO',
 'B',
 'LSTAT')
```

2. Backward elimination

In *backward elimination*, we start with the full model (including all the independent variables) and then remove the insignificant feature with highest *p-value*(> *significance level*). This process repeats again and again until we have the final set of *significant*features.

In short, the steps involved in *backward elimination* are as follows:

1. Choose a *significance level* (e.g. SL = 0.05 with a 95% confidence).
2. Fit a full model including all the features.
3. Consider the feature with highest *p-value*. If the *p-value* > *significance level* then go to Step 4, otherwise terminate the process.
4. Remove the feature which is under consideration.
5. Fit a model without this *feature*. Repeat the entire process from Step 3.

Now let us perform the same on `Boston house price` data.

This above function returns the final list of *significant features* based on *p-values* through ***backward elimination***.

```
backward_elimination(X,y)
```

```
# OUTPUT
['CRIM',
 'ZN',
 'CHAS',
 'NOX',
 'RM',
 'DIS',
 'RAD',
 'TAX',
 'PTRATIO',
 'B',
 'LSTAT']
```

Implementing Backward elimination using built-in functions in Python:

The same `SequentialFeatureSelector()` function can be used to perform *backward elimination* by disabling `forward` argument.

```
#Sequential backward selection(sbs)
sbs = SFS(LinearRegression(),
          k_features=11,
          forward=False,
```

```
          floating=False,
          cv=0)
sbs.fit(X, y)
sbs.k_feature_names_

# OUTPUT

('CRIM',
 'ZN',
 'CHAS',
 'NOX',
 'RM',
 'DIS',
 'RAD',
 'TAX',
 'PTRATIO',
 'B',
 'LSTAT')
```

Additional Note

Here we are directly using the optimal value of `k_features` argument in both *forward selection* and *backward elimination*. In order to find out the optimal number of *significant*features, we can use *hit and trial* method for different value of `k_features` and make the final decision by plotting it against the model performance.

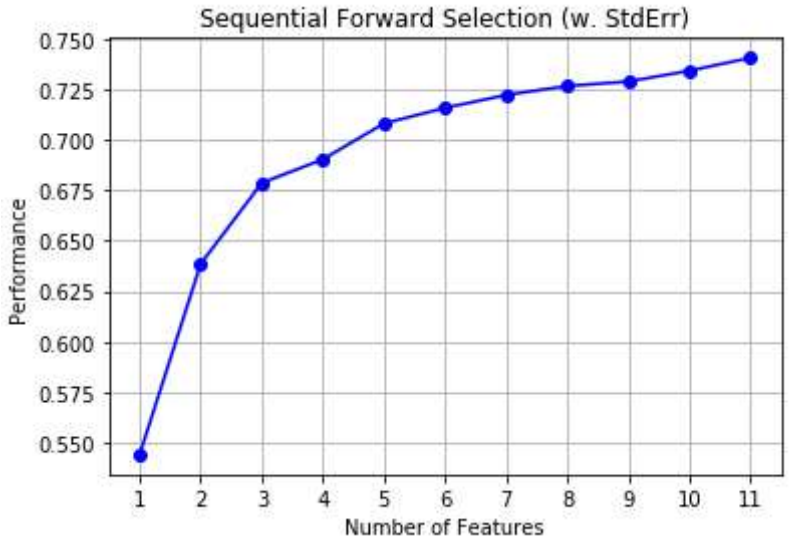
```
sfs1 = SFS(LinearRegression(),
          k_features=(3,11),
          forward=True,
          floating=False,
          cv=0)
sfs1.fit(X, y)
```

The same visualization can be achieved through `plot_sequential_feature_selection()` function available in `mlxtend.plotting` module.

```
from mlxtend.plotting import plot_sequential_feature_selection as
plot_sfs
import matplotlib.pyplot as plt
```

```
fig1 = plot_sfs(sfs1.get_metric_dict(), kind='std_dev')

plt.title('Sequential Forward Selection (w. StdErr)')
plt.grid()
plt.show()
```



Here, on the y-axis, the performance label indicates the *R-squared* values for the different numbers of features.

3. Bi-directional elimination(Stepwise Selection)

It is similar to *forward selection* but the difference is while adding a new feature it also checks the *significance* of already added features and if it finds any of the already selected features insignificant then it simply removes that particular feature through *backward elimination*.

Hence, It is a combination of *forward selection* and *backward elimination*.

In short, the steps involved in *bi-directional elimination* are as follows:

1. Choose a *significance level* to enter and exit the model (e.g. SL_in = 0.05 and SL_out = 0.05 with 95% confidence).
2. Perform the next step of *forward selection* (newly added feature must have *p-value* < SL_in to enter).

3. Perform all steps of *backward elimination* (any previously added feature with *p-value*>SL_out is ready to exit the model).

4. Repeat step 2 and 3 until we get a final *optimal* set of features.

Let us perform the same on `Boston house price` data.

This above function returns the final list of *significant* features based on *p-values* through *bi-directional elimination*.

```
stepwise_selection(X,y)
```

```
  # OUTPUT
```

```
['LSTAT',
 'RM',
 'PTRATIO',
 'DIS',
 'NOX',
 'CHAS',
 'B',
 'ZN',
 'CRIM',
 'RAD',
 'TAX']
```

Implementing bi-directional elimination using built-in functions in Python:

The same `SequentialFeatureSelector()` function can be used to perform *backward elimination* by enabling `forward` and `floating` argument.

```
# Sequential Forward Floating Selection(sffs)
sffs = SFS(LinearRegression(),
           k_features=(3,11),
           forward=True,
           floating=True,
           cv=0)
sffs.fit(X, y)
sffs.k_feature_names_

# OUTPUT

('CRIM',
 'ZN',
 'CHAS',
 'NOX',
 'RM',
 'DIS',
 'RAD',
 'TAX',
 'PTRATIO',
 'B',
 'LSTAT')
```

End Notes

In this article, we saw different kinds of Wrapper methods for feature selection with implementation using `mlxtend` library in Python.