featimp

May 8, 2021

[1]:
```
import sklearn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%config InlineBackend.figure_format='retina'
from featimp import *
```

1 Introduction to Feature Importance

Feature Importance is well, important, because it enables creating better and faster models. It is vital to understand how feature importance can provide this, by seeing how feature importance fundamentally works, and ultimately what it tells us (and what it cannot tell us).

This guide walks through two fundamental strategies for determining feature importance: **data based**, and **model based**. To conclude, we will look at strategies for picking features in **model selection**.

2 Data Based Feature Importance

Data based strategies contrast Model based strategies because they can define importances solely based on the structure of the data alone. If you had to define feature knowing nothing about machine learning, you would intuitively look for the feature that has the best correlation to the output. If you're able to find a variable that perfectly correlates to the target then that feature is clearly important. At the same time if you find a feature that has no correlation to the output,then it's easy to see how that feature is unimportant. This is what feature importance is doing, by trying to find features that correlate; however once the number of features rise, simple correlation will not suffice.

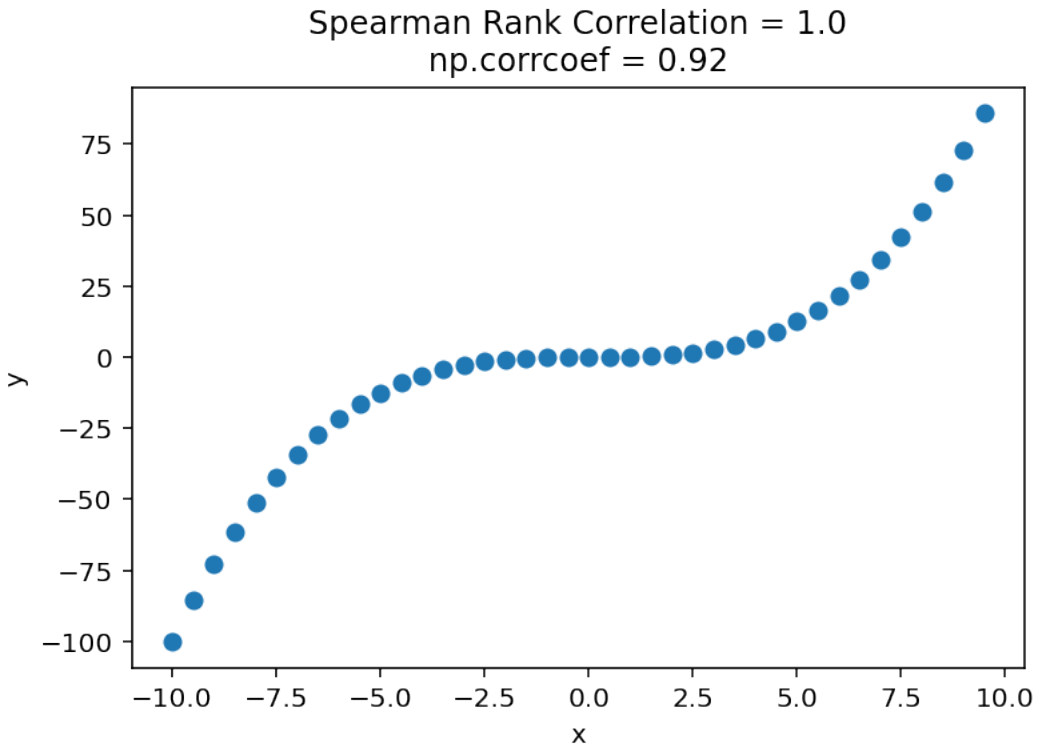
A number of data-based strategies are spearman's rank correlation coefficient, principal component analysis, and minimum redundancy maximum relevance. To begin let's look at correlation coefficient's as this is a feature importance is doing.

2.1 Correlation coefficient as a first approximation

If there is only one feature between X and Y, would you conclude that this feature is important? It obviously does not have a linear relationship but there is information in the fact that as X increases so does Y. That monotonicity is important for determining importance. In the figure title I show both the Spearman rank correlation coefficient and the standard numpy correlation coefficient, also

known as the Pearson correlation coefficient. What is important to realize about the correlation is that it weighs only on the rank, which is why you can see there's a perfect Spearman correlation between X and Y even though the relationship is not linear.

[2]:
```
from scipy import stats
x = np.arange(-10,10, .5)
y = x**3/10
plt.scatter(x,y)
plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Spearman Rank Correlation = {stats.spearmanr(x,y)[0]}\nnp.corrcoef␣
 ↪= {np.corrcoef(x,y)[0,1]:.2f}')
plt.show()
```



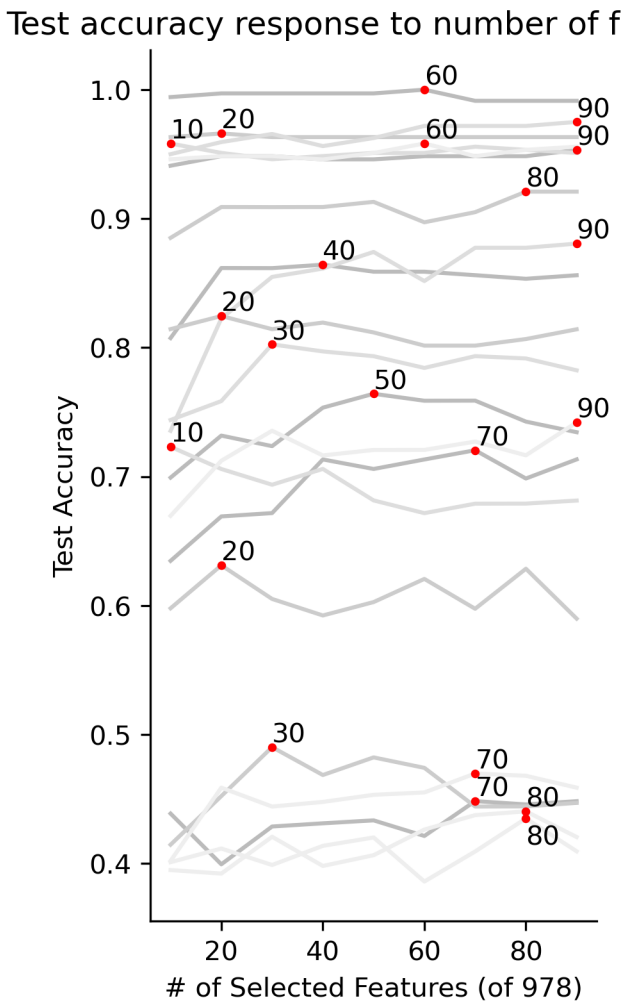
2.2 Principal Component Analysis

PCA is a technique from linear algebra, which can take the data, interpreted as a matrix and run singular value decomposition on it which is a matrix factorization technique that decomposes The matrix into eigenvectors and eigenvalues which are projected vectors along the axis of least variance. Because of this, PCA can be helpful because the first in components will often contain much information about the data, allowing a compression of redundant features into new PCA space. There are a number of pros and cons to this, since the data no longer lives in original feature space; but sometimes this is actually beneficial if you want to anonymize data.

2.3 Minimum Redundancy, Maximum Relevancy (mRMR)

The final data based feature important strategy I want to talk about is minimum redundancy, maximum relevancy. This technique can work at scale, choose the best k features, and lead to increased model performance and accuracy. To see a comparison of mRMR across 21 different problems, I evaluated a number of models across a range of the selected features to demonstrate that setting models with the maximum number of features is not always the best strategy. In fact, when feature space starts to rise in tabular data to the thousands, mRMR can be a useful technique as demonstrated by the uber team in this paper. The specific example shown below is cancer detection data using 978 features. I swept the top K features, and plot the test accuracy. Each annotation highlights the **max** test score and labels the **number of features** in that maximum score. There are a few points worth mentioning:

1. The top test accuracy model for each problem selects much less features than the total number of features available: 90 « 978
2. The difficulty of the problem (lower test score) does not highly correlate to the number of features to find the best score.
3. Some models can perform excellently (>%95) selecting less than 20 features from ~1000. This is of course problem dependent, but at scale, if you can reduce your feature set by 2 orders of magnitude, that is a substantial difference.



3 Model Based strategies:

There are three strategies I'll discuss which require a model. The reason they require a model is because the importance of each feature column is based off of baseline at the model versus any differences that changing that data and retraining the model may produce. Retraining is not always necessary as we will see with importance, but that more information that is the model and computation training is required in order to determine importance scores using model based strategies.

[3]:
```
from sklearn.ensemble           import RandomForestClassifier
from sklearn.datasets           import load_wine, load_iris
```

3.1 Drop Column Importance

A simple way to use a model to determine the importance of a feature is to drop the feature from a retrained model and see how it performs on test data as compared to the baseline. This is precisely what **Drop Column Importance** does. The code is simple to implement and the essence of the algorithm is summed up in these lines:

```
for col in x_train.columns:
    if method == 'drop':
        model.fit(x_train.drop(col, axis =1), y_train)
        y_ht = model.predict(x_test.drop(col, axis = 1))
```

The model is fit on the dropped column and `y_ht` is used to determine the impact of dropping that particular column. One main downside to the drop column technique is the cost to retrain for each feature. With hundreds of features, this can quickly become costly.

[4]:
```
# Drop Column Importance
X,y =load_wine(as_frame = True, return_X_y= True)
col_importance(RandomForestClassifier(), X,y,'perm')
```

[4]:

	Column Name	Feature Importance
0	proline	0.142977
1	color_intensity	0.075251
2	hue	0.022575
3	flavanoids	0.022575
4	od280/od315_of_diluted_wines	0.015050
5	alcohol	0.007525
6	malic_acid	0.000000
7	ash	0.000000
8	alcalinity_of_ash	0.000000
9	magnesium	0.000000
10	total_phenols	0.000000
11	nonflavanoid_phenols	0.000000
12	proanthocyanins	0.000000

3.2 Permutation Column Importance

In order to address the training cost for drop-column, **Permutation Importance** simplifies the algorithm to avoid retraining. The code is also simple to implement. The essence of the algorithm is summed up in these lines:

```
for col in x_train.columns:
    if method == 'perm':
        perm_x_test = x_test.copy()
        perm_x_test[col] = np.random.permutation(x_test[col].values)
        y_ht = model.predict(perm_x_test)
```

The model is retested after the selected column is randomly shuffled and `y_ht` is used to determine the impact of shuffling that particular column. Permutation Column is faster due to not needing to retrain models, and permutations can be repeated `n` times for more reliable results.

[5]:
```
# Permutation Importance
X,y =load_wine(as_frame = True, return_X_y= True)
col_importance(RandomForestClassifier(), X,y,'perm')
```

[5]:

	Column Name	Feature Importance
0	alcohol	0.052676
1	color_intensity	0.045151
2	flavanoids	0.037625
3	hue	0.022575
4	od280/od315_of_diluted_wines	0.022575
5	proline	0.022575
6	malic_acid	0.000000
7	ash	0.000000
8	alcalinity_of_ash	0.000000
9	magnesium	0.000000
10	total_phenols	0.000000
11	nonflavanoid_phenols	0.000000
12	proanthocyanins	0.000000

3.3 Gini Drop Importance Metric

The final Feature Importance metric to look at is Gini Impurity Drop. It is a measurement of the gini purity drop for each feature across all features and all trees in an ensembled forest. The is calculated automatically in sklearn and can be called with:

`model.feature_importances_`

[6]:
```
X,y =load_wine(as_frame = True, return_X_y= True)
gini_importance(RandomForestClassifier(), X,y)
```

[6]:

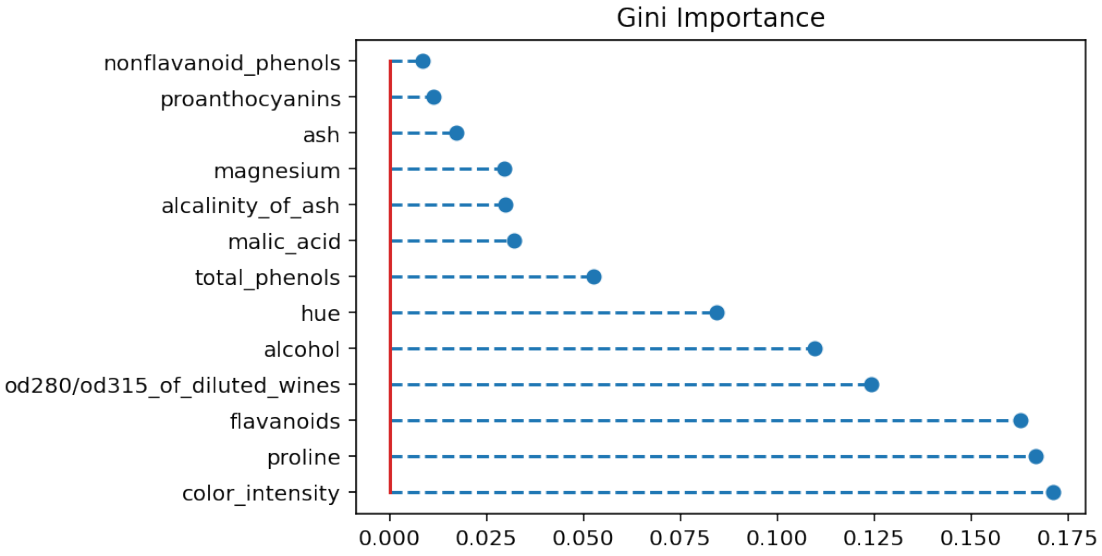
	Column Name	Feature Importance
0	color_intensity	0.174654
1	proline	0.169352
2	flavanoids	0.146240
3	alcohol	0.118238
4	od280/od315_of_diluted_wines	0.096350
5	hue	0.075431
6	total_phenols	0.074721
7	magnesium	0.034786
8	alcalinity_of_ash	0.029059
9	proanthocyanins	0.027061
10	malic_acid	0.026640
11	ash	0.015739
12	nonflavanoid_phenols	0.011730

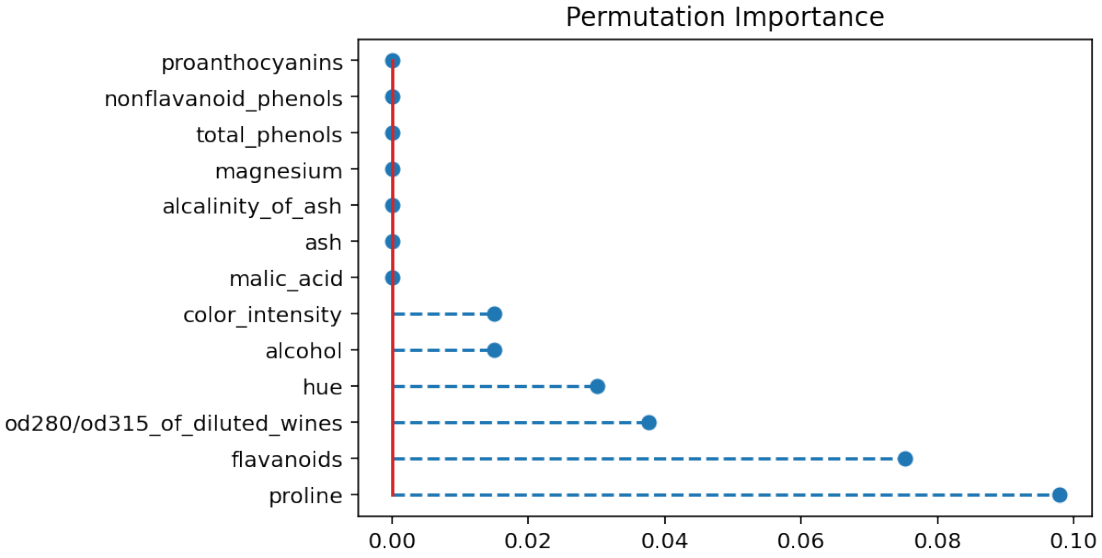
3.4 Comparing Model Based Approaches

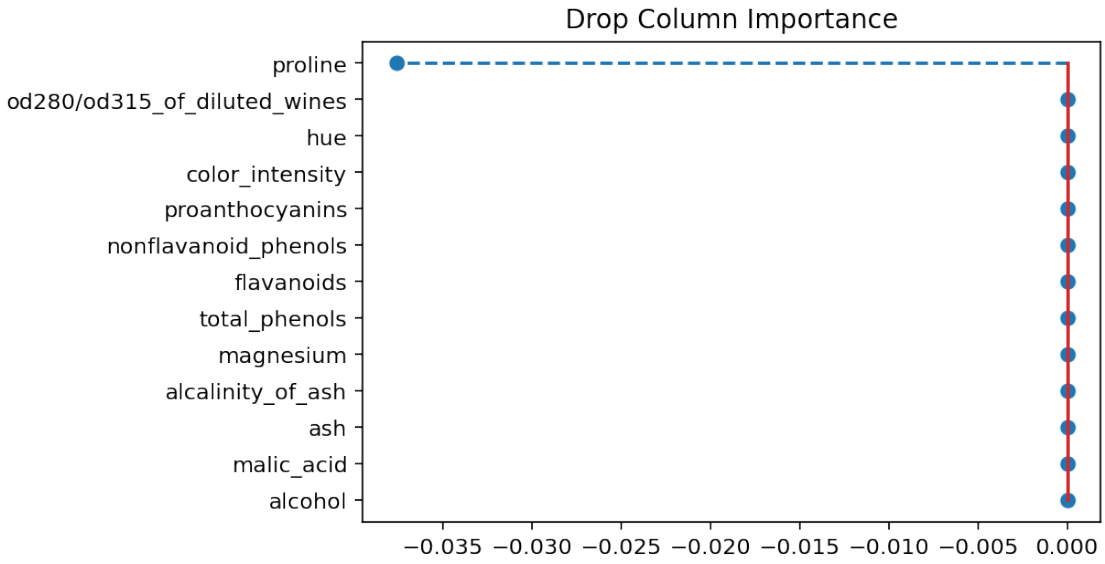
Each model based approach has different tendencies and will produce (sometimes drastically) different results.

Below you can compare the differences between 'Gini Importance', 'Permutation Importance', 'Drop Column Importance' as they have been visualized together.

[7]:
```
X,y =load_wine(as_frame = True, return_X_y= True)
plot_imps(gini_importance(RandomForestClassifier(), X,y), 'Gini Importance')
plot_imps(col_importance(RandomForestClassifier(), X,y,'perm'), 'Permutation␣
↪Importance')
plot_imps(col_importance(RandomForestClassifier(), X,y,'drop'), 'Drop Column␣
↪Importance')
```







4 Automatic Feature Selection

There are many ways to create systems that automatically selects the proper number of features. I use the mRMR feature selector to choose the top features with minimum Redundancy and used these features as a parameter to GridSearchCV. By logging all the models and test results, I can systematically filter out the best model with the ideal subset of features.

Below is a screenshot of the top results from a number of different modeling datasets.

	pid	n_features	params	mean_test_score
2	0	50	{'max_depth': 5, 'n_estimators': 1300}	0.764235
1	0	50	{'max_depth': 5, 'n_estimators': 1000}	0.761496
2	0	70	{'max_depth': 5, 'n_estimators': 1300}	0.758904
5	0	50	{'max_depth': 10, 'n_estimators': 1300}	0.758830

	pid	n_features	params	mean_test_score
0	1	70	{'max_depth': 5, 'n_estimators': 800}	0.448419
3	1	140	{'max_depth': 10, 'n_estimators': 800}	0.448359
3	1	90	{'max_depth': 10, 'n_estimators': 800}	0.448359
3	1	100	{'max_depth': 10, 'n_estimators': 800}	0.448359

	pid	n_features	params	mean_test_score
1	2	70	{'max_depth': 5, 'n_estimators': 1000}	0.720446
2	2	70	{'max_depth': 5, 'n_estimators': 1300}	0.715507
3	2	130	{'max_depth': 10, 'n_estimators': 800}	0.713339
3	2	150	{'max_depth': 10, 'n_estimators': 800}	0.713339

	pid	n_features	params	mean_test_score
5	3	150	{'max_depth': 10, 'n_estimators': 1300}	0.953327
5	3	100	{'max_depth': 10, 'n_estimators': 1300}	0.953327
2	3	120	{'max_depth': 5, 'n_estimators': 1300}	0.953327
1	3	120	{'max_depth': 5, 'n_estimators': 1000}	0.953327

In order to grab the top model for each dataset, it is a simple filtering with pd:

```
top_params = all_test_results[all_test_results.pid == pid]
                    .sort_values(by= ['mean_test_score'], ascending = False)
                    .iloc[0]
```

The original plot showing the mRMR results is a product of this procedure. Results are plotted along the x axis as the number of features selected from mRMR, and the y-axis shows the results. As noted, the algorithm is able to find the best feature set using much less than the original number of features.

[ ]: