

Doc

Breast Cancer Dataset Analysis

Data preprocessing

Dealing with missing data

```
1 print(X_train.isnull().sum())
2 print(y_train.isnull().sum())
3 print(X_test.isnull().sum())
4 print(y_test.isnull().sum())
```



It has been found that there is no missing data.

How the number of samples (aka n_estimators) affects the performance of the algorithm.

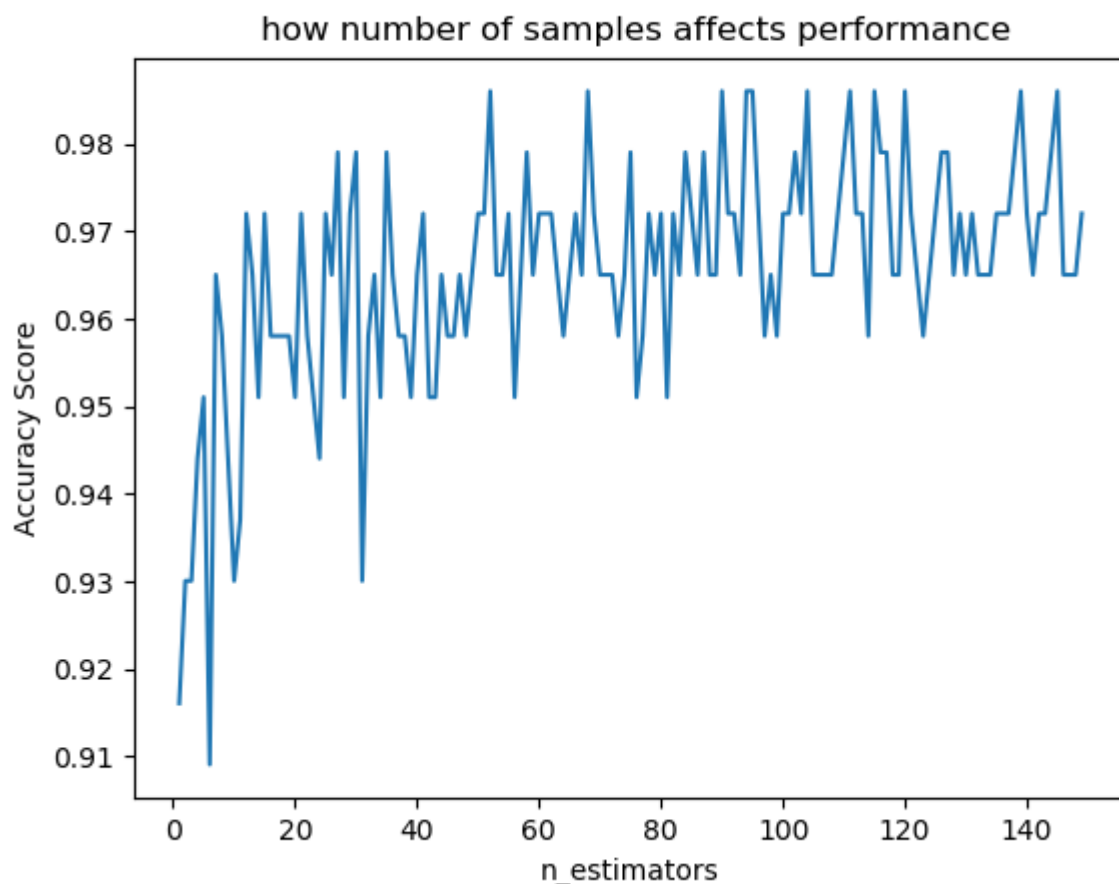
Applying the Random Forests algorithm with the n_estimators argument being in range of 1 to 150:

```
1 # Creating the lists to store the scores and corresponding n_estimator arg
2 n_estimators_score = list()
3 n_estimators_number = list()
4
5 # Calculating the accuracy scores using Random Forests
6 for i in range(1,150):
7     forest = rf.RandomForest(n_estimators = i, random_state = 1)
8     forest.fit(X_train, y_train)
9     y_pred = forest.predict(X_test)
10    score = accuracy_score(y_test, y_pred)
11    n_estimators_score.append(score)
12    n_estimators_number.append(i)
```

Plotting the output to have a better idea

```
1 # Plotting the result
2 fig, ax = plt.subplots()
3 ax.plot(n_estimators_number, n_estimators_score)
4 ax.set(xlabel="n_estimators", ylabel="Accuracy Score", title="how number
5 fig.savefig("n_estimators_performance.png")
6 plt.show()
```

Here is what we got:



It can be seen that starting from around **n_estimators = 25**, we starting to get a high **98% of accuracy score**. Also, the peak has first been approximately found at **n_estimators = 53**. Afterwards, the score has been always **in range of 95% to 99%**. As it can be seen, the value of **n_estimators** does not significantly affect the accuracy score when **n_estimators >= 50**.

Therefore, to enhance our time performance let **n_estimators = 50**. If the size of the dataset is going to be significantly increased, then it is common to make **n_estimators** to be in the range of square root of number of training samples. In this case, it is around 20. But since the dataset is not large, it is affordable to make it a bit larger.

How the maximum depth number affects the performance of the algorithm.

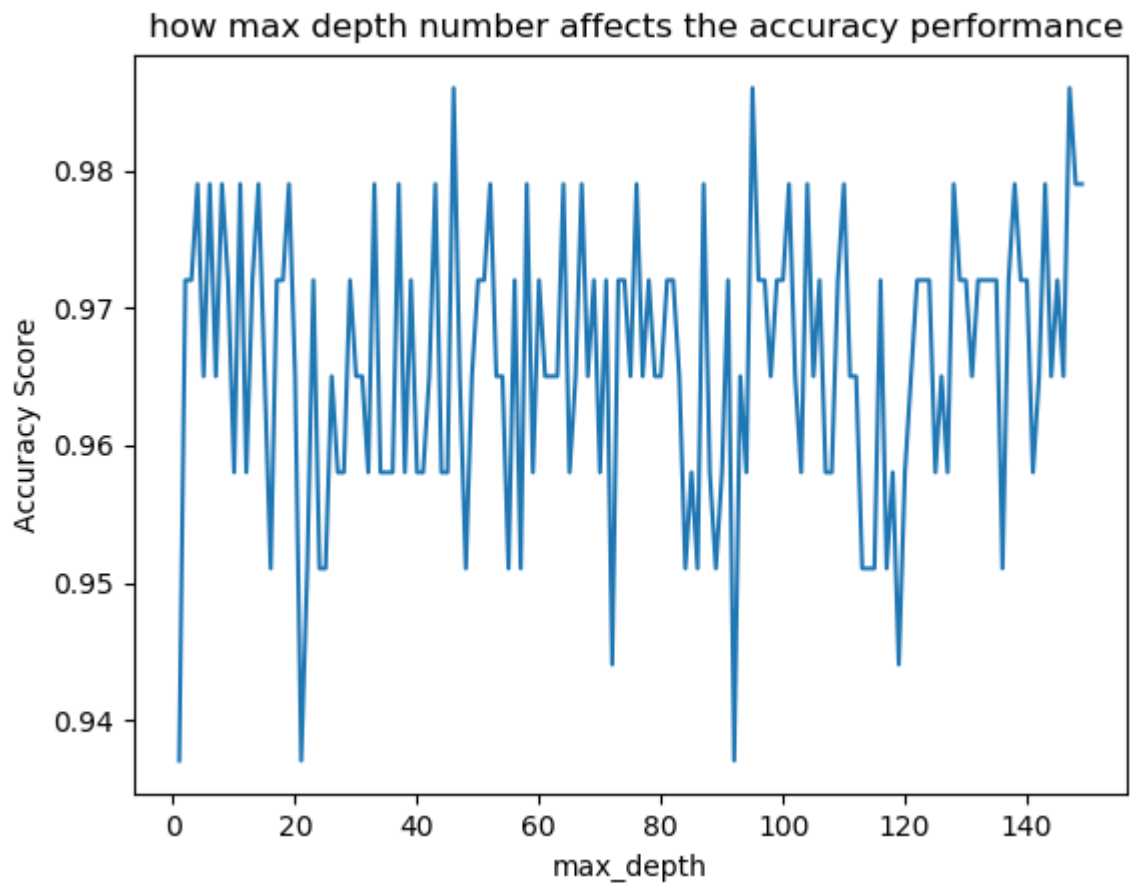
Applying the Random Forests algorithm with the **max_depth** argument being in range of 1 to 150:

```
1 # Creating the lists to store the scores and corresponding max_depth argument
2 max_depth_score = list()
3 max_depth_number = list()
4
5 # Calculating the accuracy scores using Random Forests
6 for i in range(1,150):
7     forest = rf.RandomForest(n_estimators=50, max_depth=i, random_state =
8     forest.fit(X_train, y_train)
9     y_pred = forest.predict(X_test)
10    score = accuracy_score(y_test, y_pred)
11    max_depth_score.append(score)
12    max_depth_number.append(i)
```

Plotting the output to have a better idea

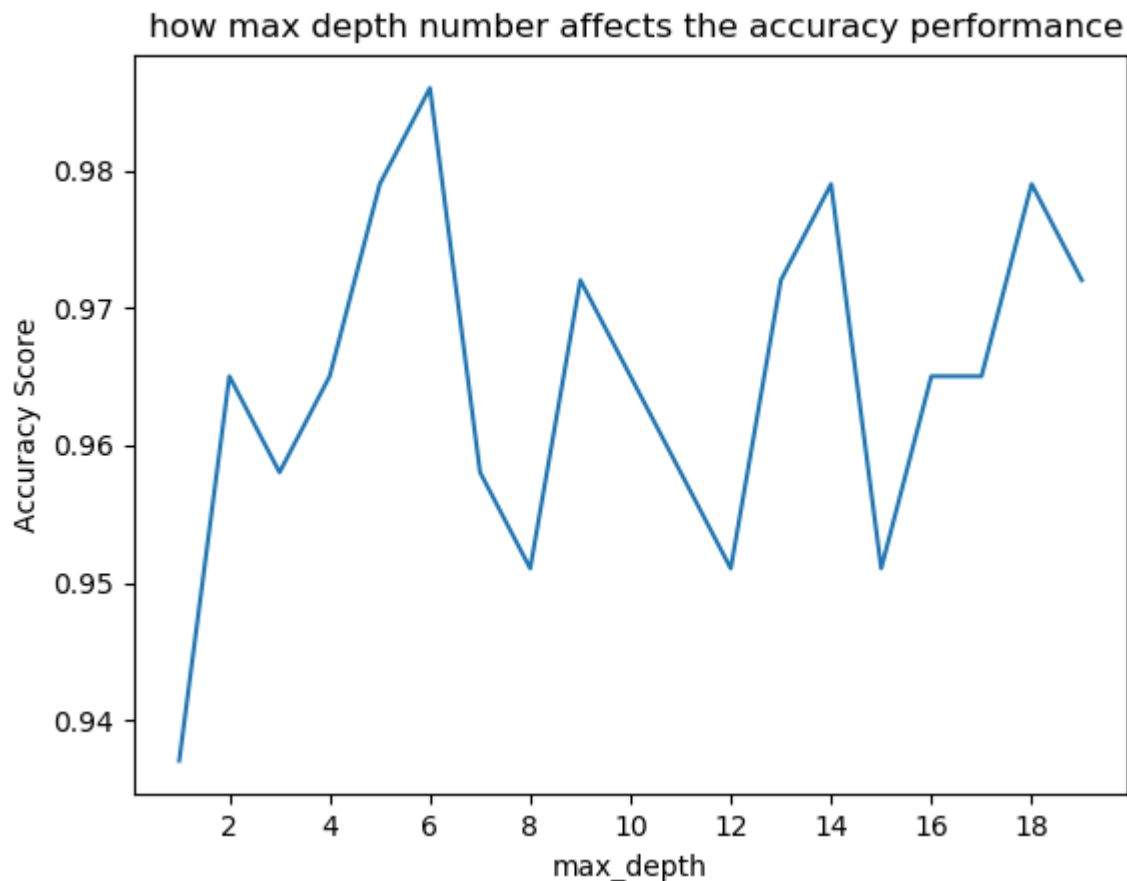
```
1 # Plotting the result
2 fig, ax = plt.subplots()
3 ax.plot(max_depth_number,max_depth_score)
4 ax.set(xlabel="max_depth", ylabel="Accuracy Score", title=" how max depth
5 fig.savefig("max_depth_performance.png")
6 plt.show()
```

Here is what we got:



Although, the highest scores are found when **max_depth > 40**, but there are more good enough scores in the range of 5 to 20. Hence, let's consider the plot with **max_depth** being in that range.

Consider $1 < \text{max_depth} < 20$:



The code is omitted, since it can be trivially edited to get this

The code has been run many times, and I found that most of the time when **max_depth = 6**, the accuracy score is higher than most. Of course, it might be reasonable to change it if we have a larger dataset, but in this case it is appropriate to set **max_depth = 6**.

How the maximum features number affects the performance of the algorithm.

Recall, the total number of features is 30.

Therefore, applying the Random Forests algorithm with the **max_features** argument being in range of 1 to 30:



```

1 # storing all the scores and corresponding max_features arguments in the 1
2 max_features_score = list()
3 max_features_number = list()
4
5 # Calculating the accuracy scores using Random Forests
6 for i in range(1,30):
7     forest = rf.RandomForest(n_estimators=50, max_features=i, random_state
8     forest.fit(X_train, y_train)
9     y_pred = forest.predict(X_test)
10    score = accuracy_score(y_test, y_pred)
11    max_features_score.append(score)
12    max_features_number.append(i)

```

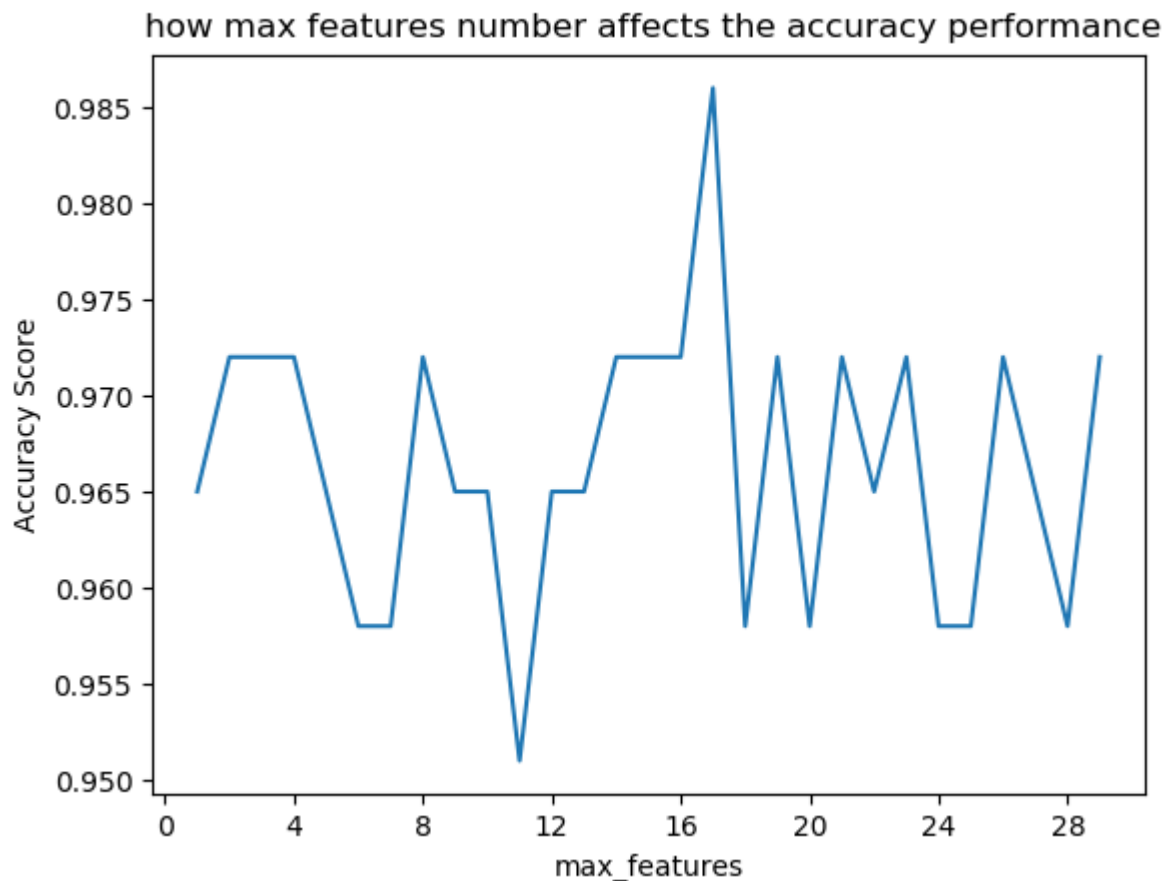
Plotting the output to have a better idea:

```

1 fig, ax = plt.subplots()
2 ax.plot(max_features_number,max_features_score)
3 ax.set(xlabel="max_features", ylabel="Accuracy Score", title=" how max fea
4 ax.xaxis.set_major_locator(MaxNLocator(integer=True))
5 fig.savefig("max_features_performance.png")
6 plt.show()

```

Here is the result:



It can be seen that we are getting very similar result in the range of 14 to 30. Hence, it might be a good idea to perform a Feature Selection.

Feature Selection

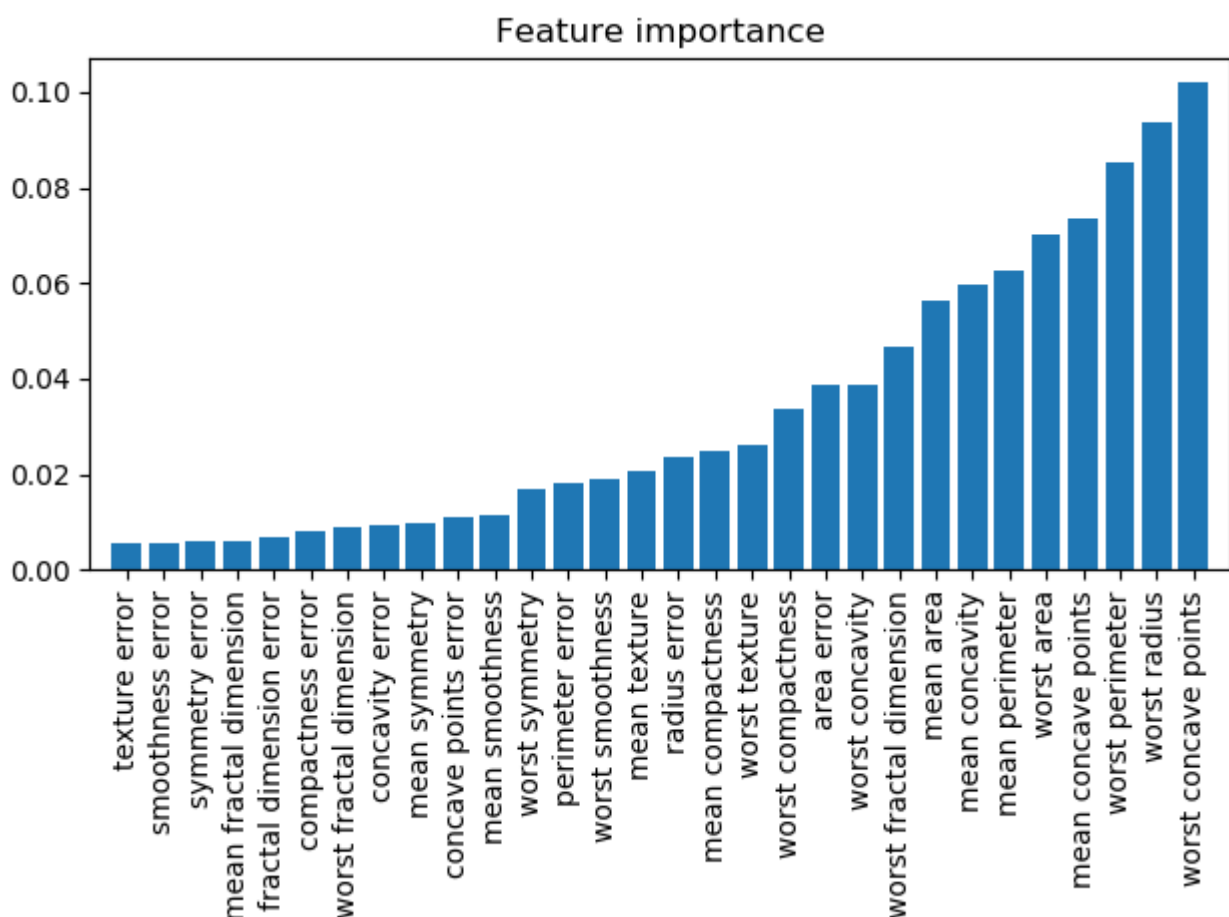
As we know, some of the features might be insignificant, therefore let's select the most significant features and analyze how it affects the accuracy performance.

```
1 # FEATURE SELECTION
2 # Find the most important features
3 feat_labels = X_train.columns[1:]
4 clf = ExtraTreesClassifier(n_estimators=300)
5 clf = clf.fit(X_train, np.ravel(y_train))
6 importances = clf.feature_importances_
7 indices = np.argsort(importances)[::-1]
```


Now let's plot the result:

```
1 # plotting
2 fig, ax = plt.subplots()
3 plt.title('Feature importance')
4 plt.bar(range(X_train.shape[1]), importances[indices], align='center')
5 plt.xticks(range(X_train.shape[1]), feat_labels[indices-1], rotation=90)
6 plt.xlim([-1, X_train.shape[1]])
7 plt.tight_layout()
8 fig.savefig("features_importances.png")
9 plt.show()
```

And we get:



We see that not many features are as significant as some of the others. Therefore, let's apply a Select from Model.

Selecting the most significant features for model:

```
1 model = SelectFromModel(clf, prefit=True)
2 X_new = model.transform(X_train)
3 X_new = pd.DataFrame(X_new)
4 X_test_new = model.transform(X_test)
5 X_test_new = pd.DataFrame(X_test_new)
6 print("Number of features selected:", X_new.shape[1])
7 #Number of features selected varies in the range of 10-13
```

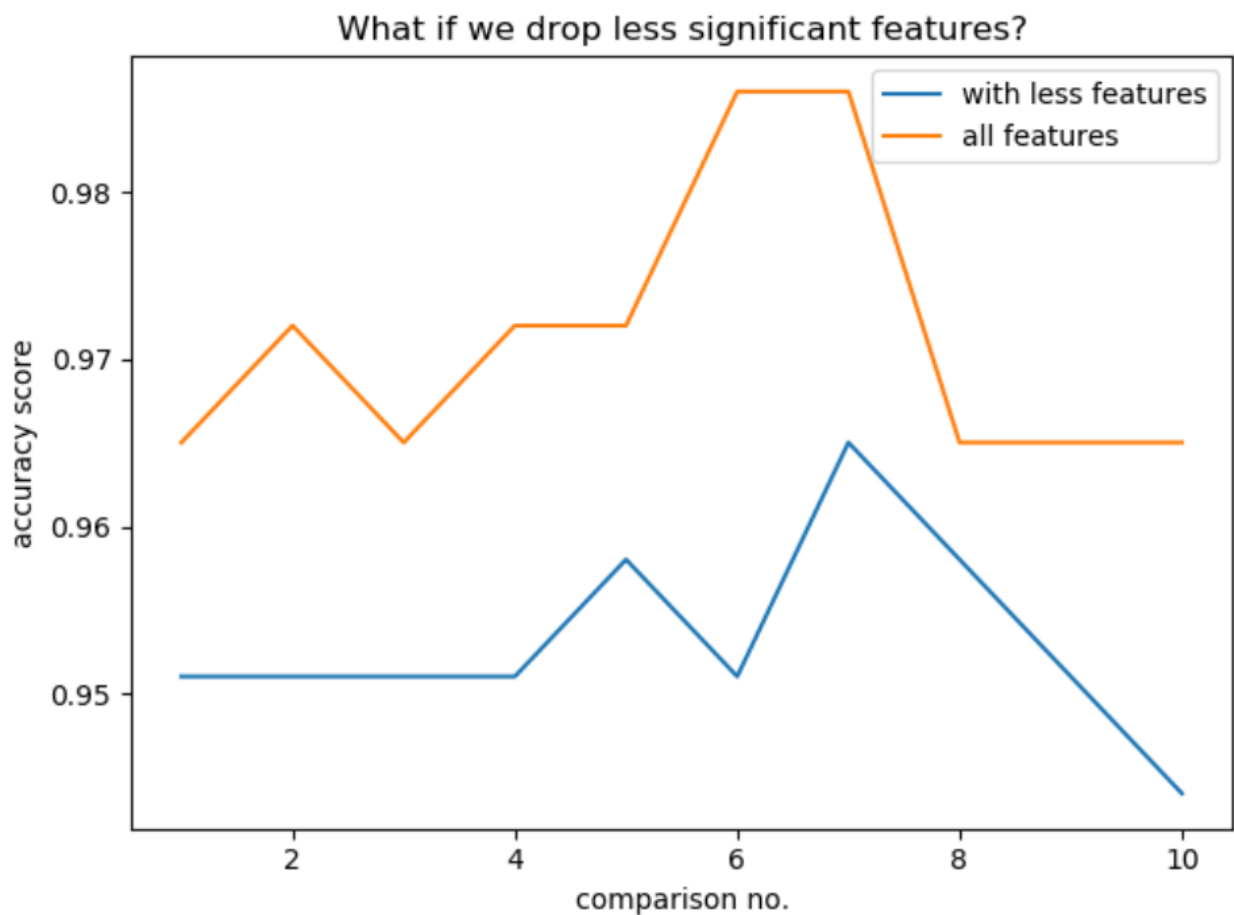
Finally, compare the model with only significant features and the model with all features.

```
1 # Compare the models 10 times
2 scorelist_new = list()
3 scorelist_old = list()
4 comparisons = list()
5 for i in range(10):
6     forest_new = rf.RandomForest(n_estimators=50, random_state=1, max_depth=6)
7     forest_new.fit(X_new, y_train)
8     y_pred_new = forest_new.predict(X_test_new)
9     score_new = accuracy_score(y_test, y_pred_new)
10    scorelist_new.append(score_new)
11
12    forest = rf.RandomForest(n_estimators=50, random_state=1, max_depth=6)
13    forest.fit(X_train, y_train)
14    y_pred = forest.predict(X_test)
15    score = accuracy_score(y_test, y_pred)
16    scorelist_old.append(score)
17
18    comparisons.append(i+1)
19 print(scorelist_new, scorelist_old)
```

Now plot the graph:

```
1 # plotting comparison
2 plt.plot(comparisons, scorelist_new, label='with less features')
3 plt.plot(comparisons, scorelist_old, label='all features')
4 plt.xlabel('comparison no.')
```

```
5 plt.ylabel('accuracy score')
6 plt.title('What if we drop less significant features?')
7 plt.legend()
8 plt.show()
```



Considering the situation, we may decide which approach to use. For our small dataset, using all features might be better, but if we had a larger dataset, we could use the approach with only significant features.

Car Evaluation Dataset Analysis

Data preprocessing

1. Dealing with missing values

main.py

```
1 print(X_train.isnull().sum())
2 print(y_train.isnull().sum())
3 print(X_test.isnull().sum())
4 print(y_test.isnull().sum())
```



It has been found that there is no missing values

2. Mapping Ordinal features

First, let's observe all unique values of features:

main.py

```
1 print('unique features of buying: ', X_train['buying'].unique())
2 print('unique features of maint: ', X_train['maint'].unique())
3 print('unique features of doors: ', X_train['doors'].unique())
4 print('unique features of persons: ', X_train['persons'].unique())
5 print('unique features of lug_boot: ', X_train['lug_boot'].unique())
6 print('unique features of safety: ', X_train['safety'].unique())
```



unique features of buying: ['med' 'high' 'low' 'vhigh']

unique features of maint: ['med' 'high' 'vhigh' 'low']

unique features of doors: ['5more' '3' '4' '2']

unique features of persons: ['4' 'more' '2']

unique features of lug_boot: ['big' 'small' 'med']

unique features of safety: ['high' 'low' 'med']

Using the above result, map the ordinal features:

```
main.py

1  buying_mapping = {'low': 1, 'med': 2, 'high': 3, 'vhigh': 4}
2  maint_mapping = {'low': 1, 'med': 2, 'high': 3, 'vhigh': 4}
3  doors_mapping = {'2': 2, '3': 3, '4': 4, '5more': 5}
4  persons_mapping = {'2': 2, '4': 4, 'more': 5}
5  lug_boot_mapping = {'small': 1, 'med': 2, 'big': 3}
6  safety_mapping = {'low': 1, 'med': 2, 'high': 3}
7
8  x_list = [X_train, X_test]
9  f_list = ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety']
10 m_list = [buying_mapping, maint_mapping, doors_mapping, persons_mapping, lug_boot_mapping, safety_mapping]
11
12 for x in x_list:
13     for i in range(len(f_list)):
14         x[f_list[i]] = x[f_list[i]].map(m_list[i])
```

⚠ Note that there are some too unspecific values like:

- "5more" value of "doors"
- "more" value of "persons"

They were mapped to 5, but we are going to analyze what could be a better choice for that in the future.

3. Label Encoding

First, find possible class labels:

main.py

```
1 print('labels: ', y_train['class'].unique())
```

✓ labels: ['vgood' 'unacc' 'acc' 'good']

Using the above result, perform label encoding:

main.py

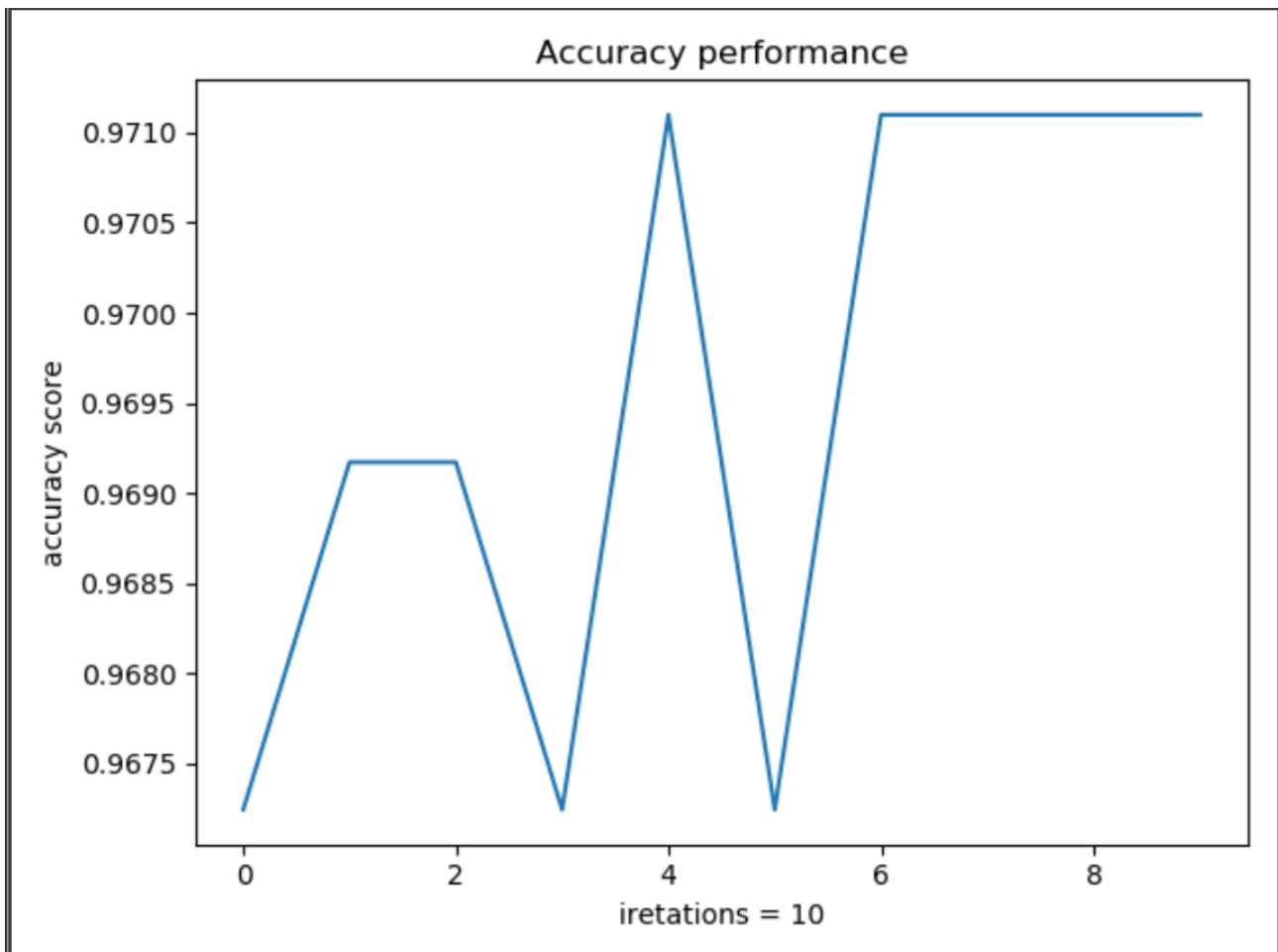
```
1 class_mapping = {'unacc': 0, 'acc': 1, 'good': 2, 'vgood': 3}
2 y_list = [y_train, y_test]
3
4 for y in y_list:
5     y['class'] = y['class'].map(class_mapping)
```

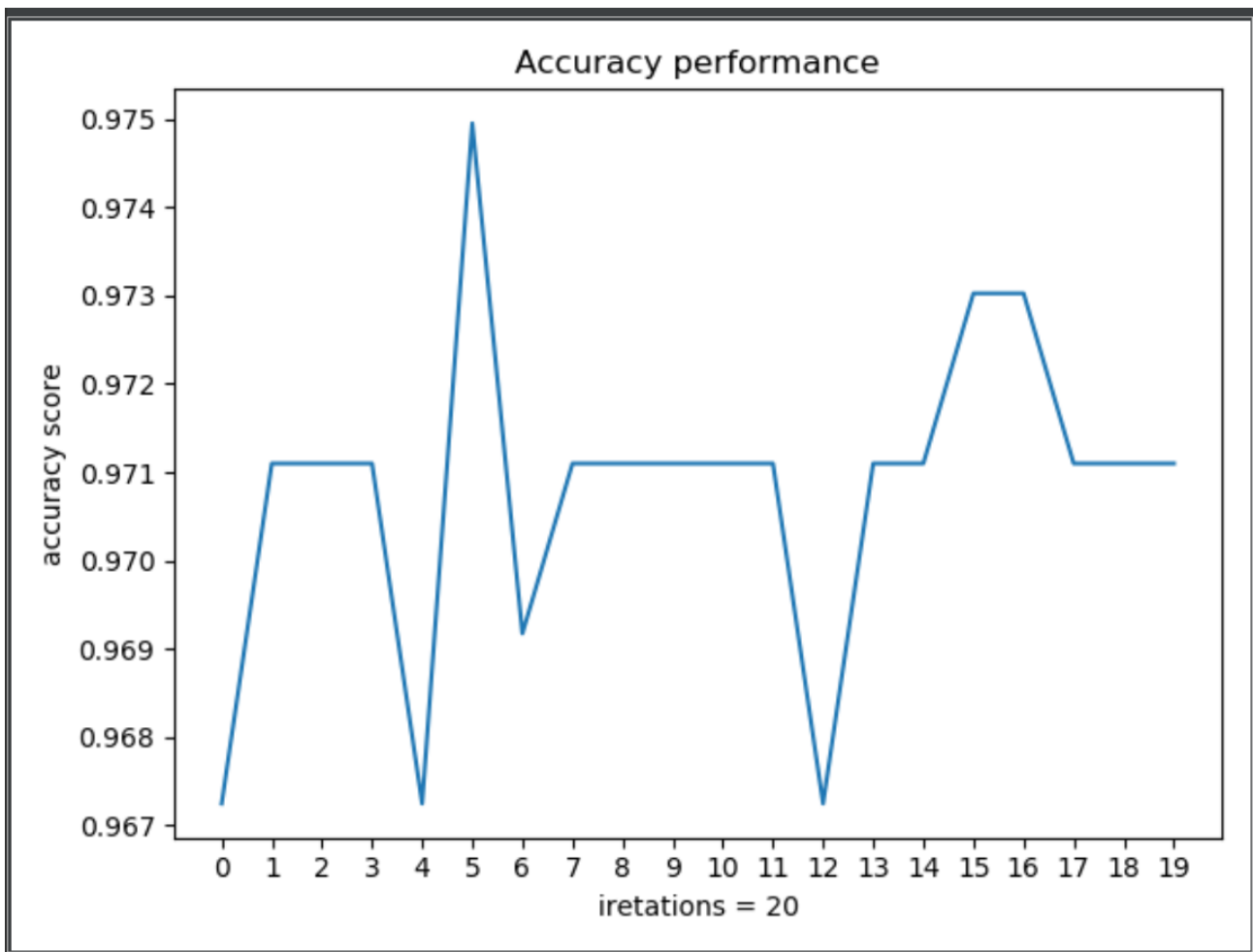
Now, let's run the Random Forests to get the accuracy score.

```
1 # Applying Random Forests 10 times
2 scorelist = list()
3 iterations = list()
4 for i in range(10):
5     forest = rf.RandomForest(random_state=1)
6     forest.fit(X_train, y_train)
7     y_pred = forest.predict(X_test)
8     score = accuracy_score(y_test, y_pred)
9     scorelist.append(score)
10    iterations.append(i)
```

And, plot it:

```
1 # plotting the result
2 plt.plot(iterations, scorelist)
3 plt.xlabel('iterations = 10')
4 plt.xticks(iterations)
5 plt.ylabel('accuracy score')
6 plt.title('Accuracy performance')
7 plt.show()
```





It can be seen that the accuracy score varies in the range of 0.965 to 0.975. It is a good result, however let's see if we can do better.

4. Choosing the right mapping values

Recall, "5more" value of "doors" and "more" value of "persons" were mapped to 5, which might be not accurate. Therefore, consider different mapping values and compare them.

1. Mapping the "5more" value of "doors" feature

- i 5 cases were compared with each other, mapping from 5 to 9. The Random Forest run 10 times, and the mean value was taken to compare the cases. Here is the code implementation:

main.py

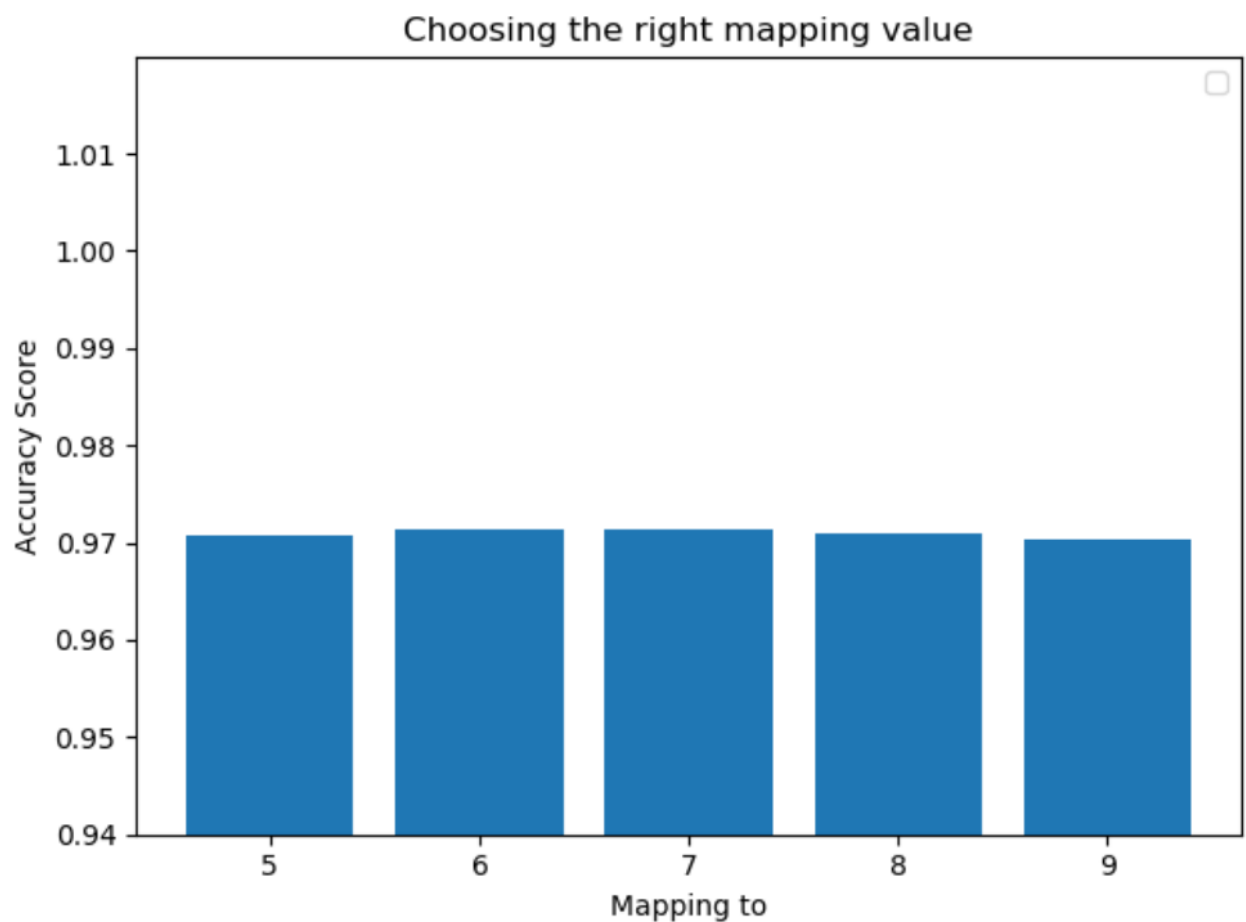
```
1  # Case-1: 5more: 5
2  doors_mapping_5 = {2: 2, 3: 3, 4: 4, 5: 5}
3  X_train['doors'] = X_train['doors'].map(doors_mapping_5)
4  X_test['doors'] = X_test['doors'].map(doors_mapping_5)
5  scorelist_5 = list()
6
7  for i in range(10):
8      forest = rf.RandomForest(random_state=1)
9      forest.fit(X_train, y_train)
10     y_pred = forest.predict(X_test)
11     score = accuracy_score(y_test, y_pred)
12     scorelist_5.append(score)
13
14 mean_5 = np.mean(scorelist_5)
15
16 # Case-1: 5more: 6
17 doors_mapping_6 = {2: 2, 3: 3, 4: 4, 5: 6}
18 X_train['doors'] = X_train['doors'].map(doors_mapping_6)
19 X_test['doors'] = X_test['doors'].map(doors_mapping_6)
20 scorelist_6 = list()
21
22 for i in range(10):
23     forest = rf.RandomForest(random_state=1)
24     forest.fit(X_train, y_train)
25     y_pred = forest.predict(X_test)
26     score = accuracy_score(y_test, y_pred)
27     scorelist_6.append(score)
28
29 mean_6 = np.mean(scorelist_6)
30
31 # Case-1: 5more: 7
32 doors_mapping_7 = {2: 2, 3: 3, 4: 4, 6: 7}
33 X_train['doors'] = X_train['doors'].map(doors_mapping_7)
34 X_test['doors'] = X_test['doors'].map(doors_mapping_7)
35 scorelist_7 = list()
36
37 for i in range(10):
38     forest = rf.RandomForest(random_state=1)
39     forest.fit(X_train, y_train)
40     y_pred = forest.predict(X_test)
41     score = accuracy_score(y_test, y_pred)
42     scorelist_7.append(score)
43
44 mean_7 = np.mean(scorelist_6)
45
46 # Case-1: 5more: 8
47 doors_mapping_8 = {2: 2, 3: 3, 4: 4, 7: 8}
48 X_train['doors'] = X_train['doors'].map(doors_mapping_8)
```

```

49 X_test['doors'] = X_test['doors'].map(doors_mapping_8)
50 scorelist_8 = list()
51
52 for i in range(10):
53     forest = rf.RandomForest(random_state=1)
54     forest.fit(X_train, y_train)
55     y_pred = forest.predict(X_test)
56     score = accuracy_score(y_test, y_pred)
57     scorelist_8.append(score)
58
59 mean_8 = np.mean(scorelist_8)
60
61 # Case-1: 5more: 9
62 doors_mapping_9 = {2: 2, 3: 3, 4: 4, 8: 9}
63 X_train['doors'] = X_train['doors'].map(doors_mapping_9)
64 X_test['doors'] = X_test['doors'].map(doors_mapping_9)
65 scorelist_9 = list()
66
67 for i in range(10):
68     forest = rf.RandomForest(random_state=1)
69     forest.fit(X_train, y_train)
70     y_pred = forest.predict(X_test)
71     score = accuracy_score(y_test, y_pred)
72     scorelist_9.append(score)
73
74 mean_9 = np.mean(scorelist_9)
75
76 mean_scores = [mean_5, mean_6, mean_7, mean_8, mean_9]
77 values = [5,6,7,8,9]
78
79 # plotting comparison
80 plt.bar(values, mean_scores)
81 plt.xticks(values)
82 plt.ylim(bottom=0.94)
83 plt.xlabel("Mapping to")
84 plt.ylabel('Accuracy Score')
85 plt.title('Choosing the right mapping value')
86 plt.legend()
87 plt.show()


```

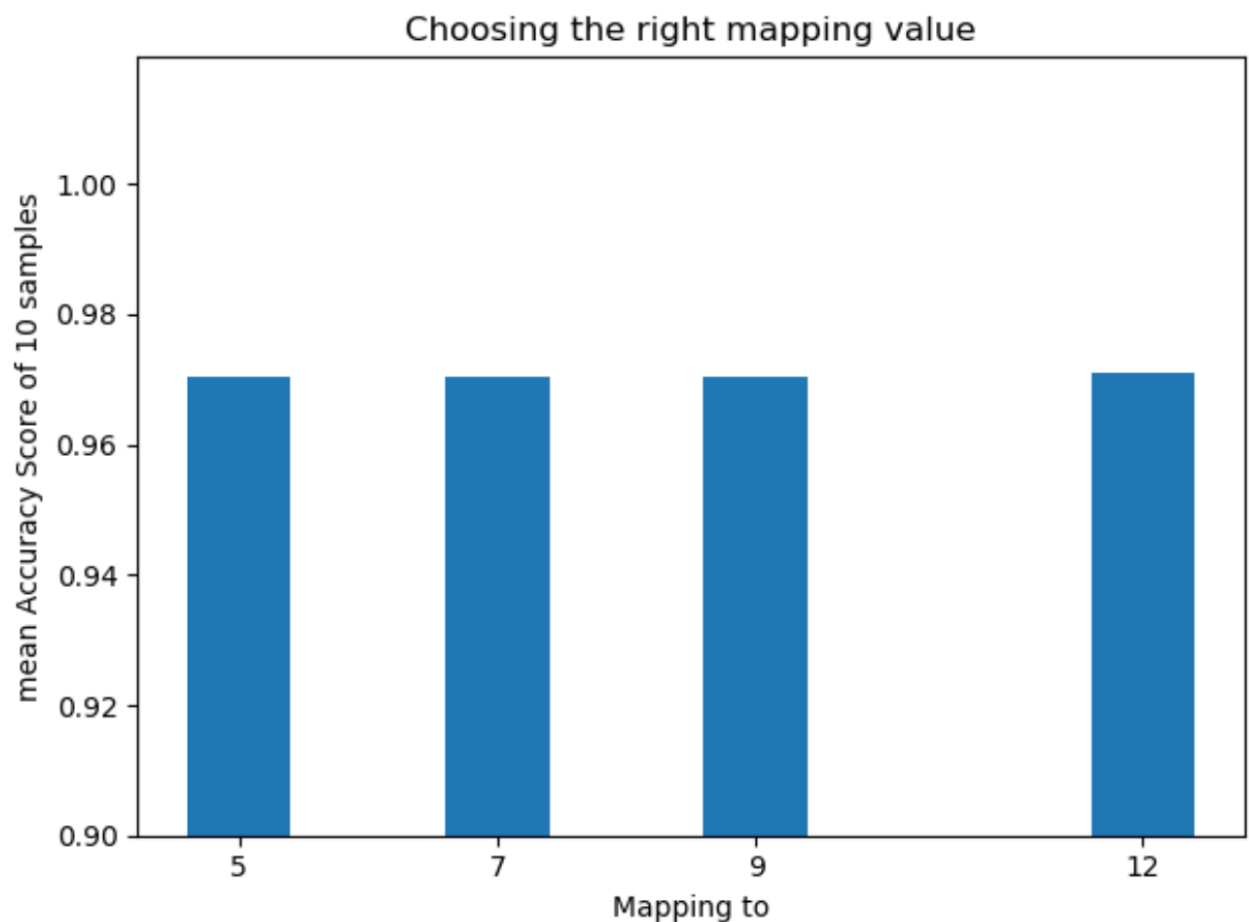
The bar chart:



We can see that the scores are very similar, hence just leave it as it was: map to 5.

1. Mapping the "more" value of "persons" feature

 The code is omitted as it is very similar to previous part. You can find the code for this section in source code.



Interestingly, the scores are very similar here as well. Therefore, let's map the "more" value of persons feature to integer 5 and proceed to further analysis.

How the number of samples (aka `n_estimators`) affects the performance of the algorithm.

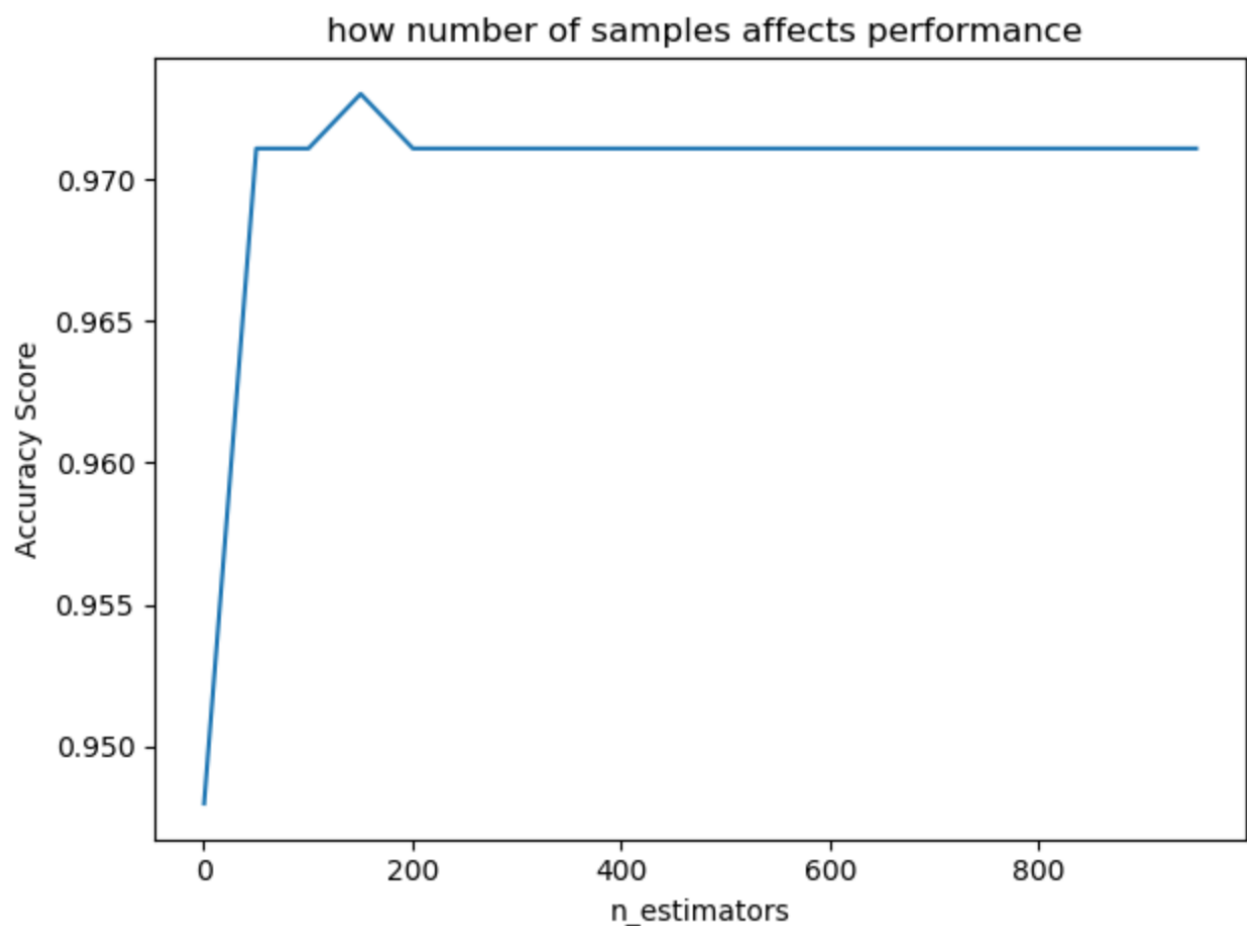
Applying the Random Forests algorithm with the `n_estimators` argument being in range of 1 to 1000 with the step of 50:

```
1 # storing all the scores and corresponding n_estimator arguments in the li
2 n_estimators_score = list()
3 n_estimators_number = list()
4
5 # Calculating the accuracy scores using Random Forests
```


```

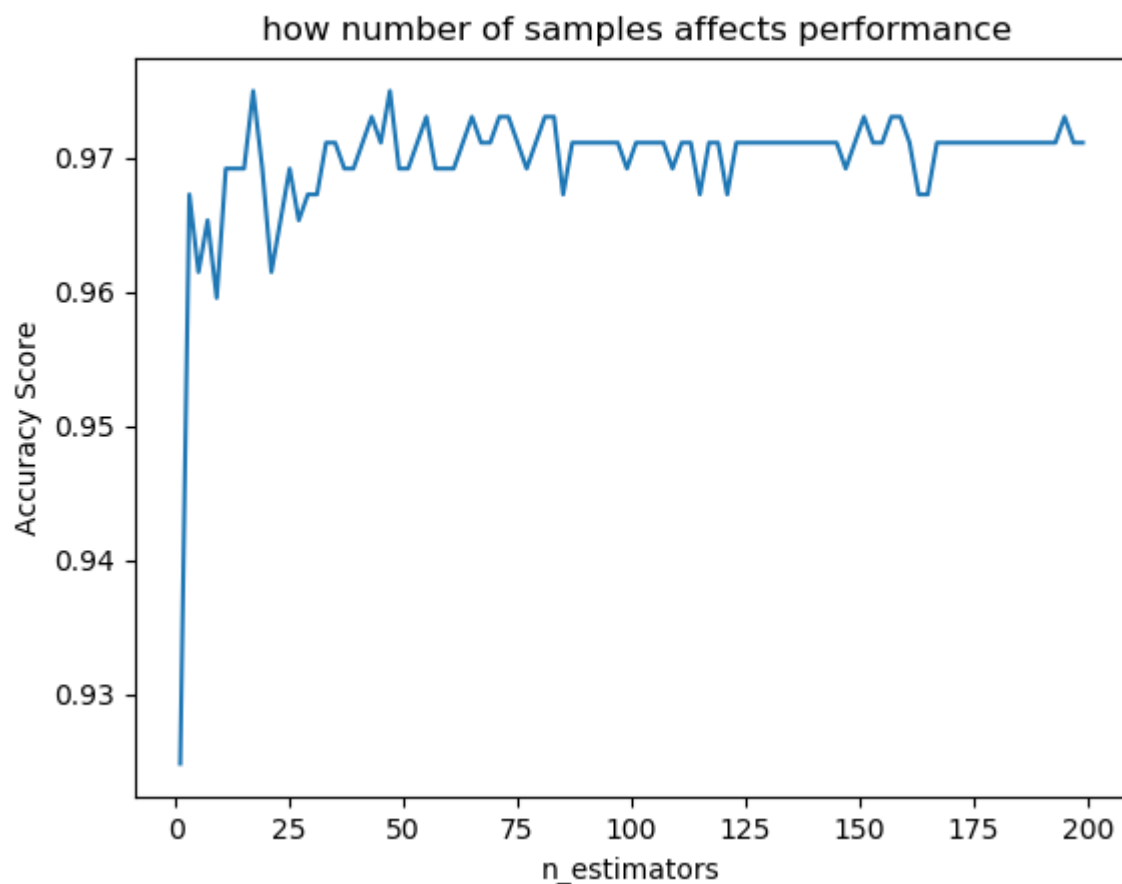
6  for i in range(1, 1001, 50):
7      forest = rf.RandomForest(n_estimators=i, random_state=1)
8      forest.fit(X_train, y_train)
9      y_pred = forest.predict(X_test)
10     score = accuracy_score(y_test, y_pred)
11     n_estimators_score.append(score)
12     n_estimators_number.append(i)
13
14 # # Plotting the result
15 fig, ax = plt.subplots()
16 ax.plot(n_estimators_number, n_estimators_score)
17 ax.set(xlabel="n_estimators", ylabel="Accuracy Score", title=" how number
18 fig.savefig("n_estimators_performance.png")
19 plt.show()

```



Here we can see that scores from 50 to 200 are as high as those >200, therefore it is reasonable to set `n_estimators` in the range of 50 to 200. Let's take a closer look by running Random Forests again, but for all values between 1 and 200.

 The code is omitted, as it can be just amended a bit to check this.



As it can be seen, starting from `n_estimators = 30`, we getting a stable scores of around 97%. After that, the performance does not really change much. Hence, to increase our time performance setting `n_estimators = 30` is appropriate.

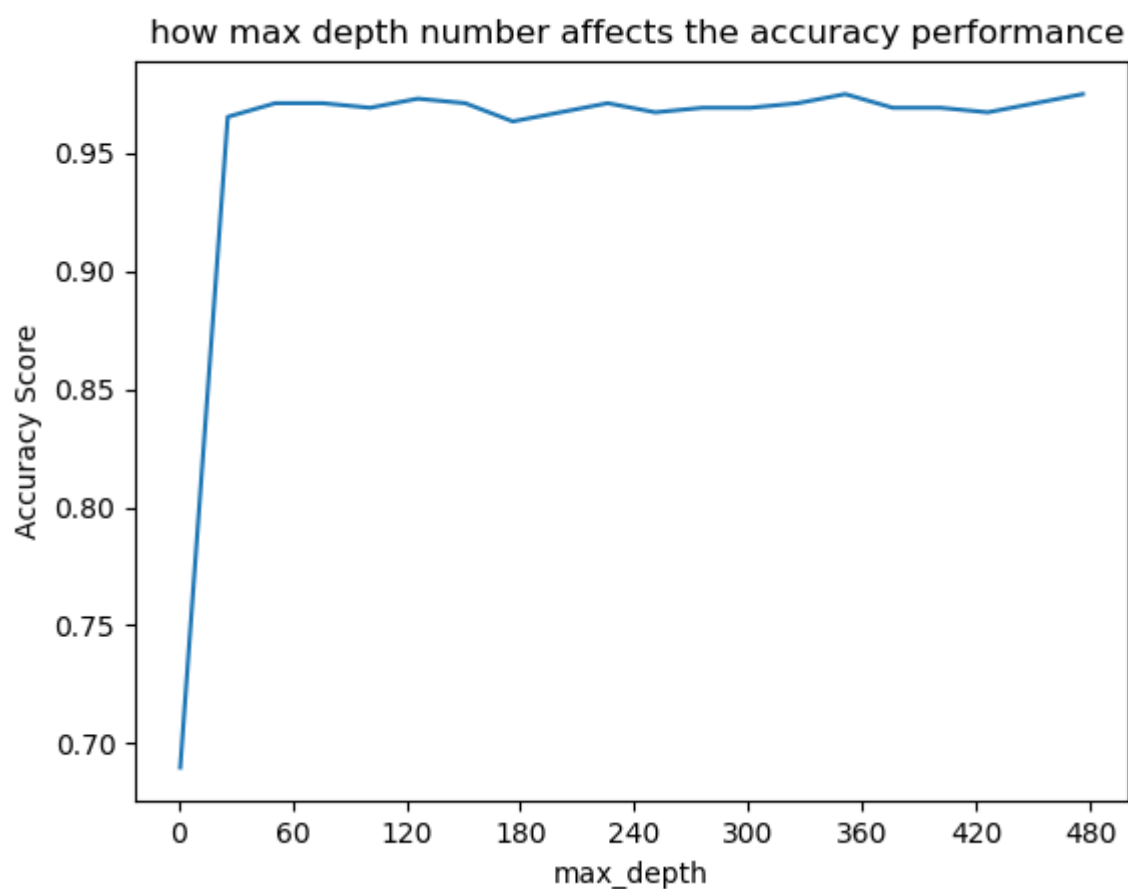
How the maximum depth number affects the performance of the algorithm.

Applying the Random Forests algorithm with the `max_depth` argument being in range of 1 to 500, with. the step of 25.

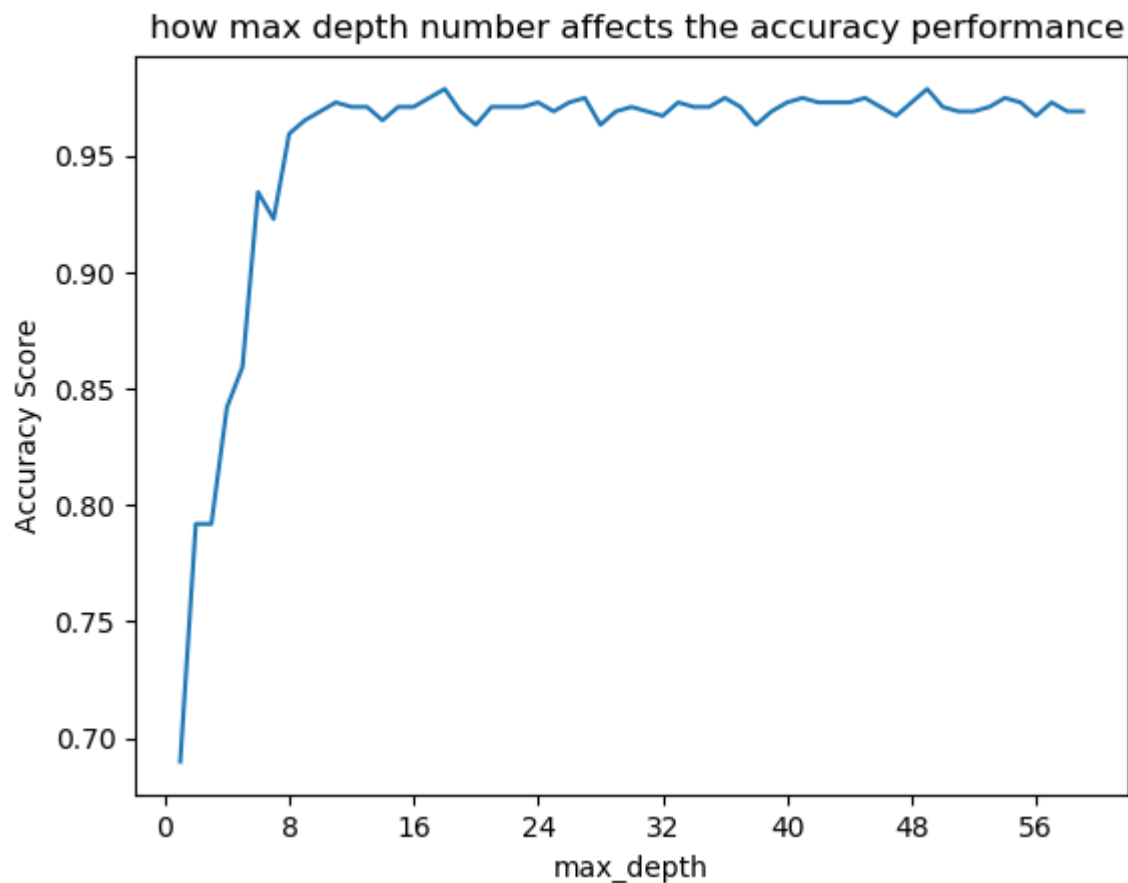
```

1 # storing all the scores and corresponding max_depth arguments in the list
2 max_depth_score = list()
3 max_depth_number = list()
4
5 # Calculating the accuracy scores using Random Forests
6 for i in range(1,500,25):
7     forest = rf.RandomForest(n_estimators=30, max_depth=i, random_state =
8     forest.fit(X_train, y_train)
9     y_pred = forest.predict(X_test)
10    score = accuracy_score(y_test, y_pred)
11    max_depth_score.append(score)
12    max_depth_number.append(i)
13
14 # Plotting the result
15 fig, ax = plt.subplots()
16 ax.plot(max_depth_number,max_depth_score)
17 ax.set(xlabel="max_depth", ylabel="Accuracy Score", title=" how max depth
18 ax.xaxis.set_major_locator(MaxNLocator(integer=True))
19 fig.savefig("max_depth_performance.png")
20 plt.show()

```



It can be seen that the effect of max_depth is not that significant. Let's take a closer look in the range between 1 to 60:



I assume, that depth is usually in the range of 10-15, that's why the results starting from 10 are very similar. Hence, setting max_depth = 10 might be a good choice for this particular dataset. However if the dataset is going be larger, then maybe it is a better choice to increase max_depth in order to have a higher accuracy score.

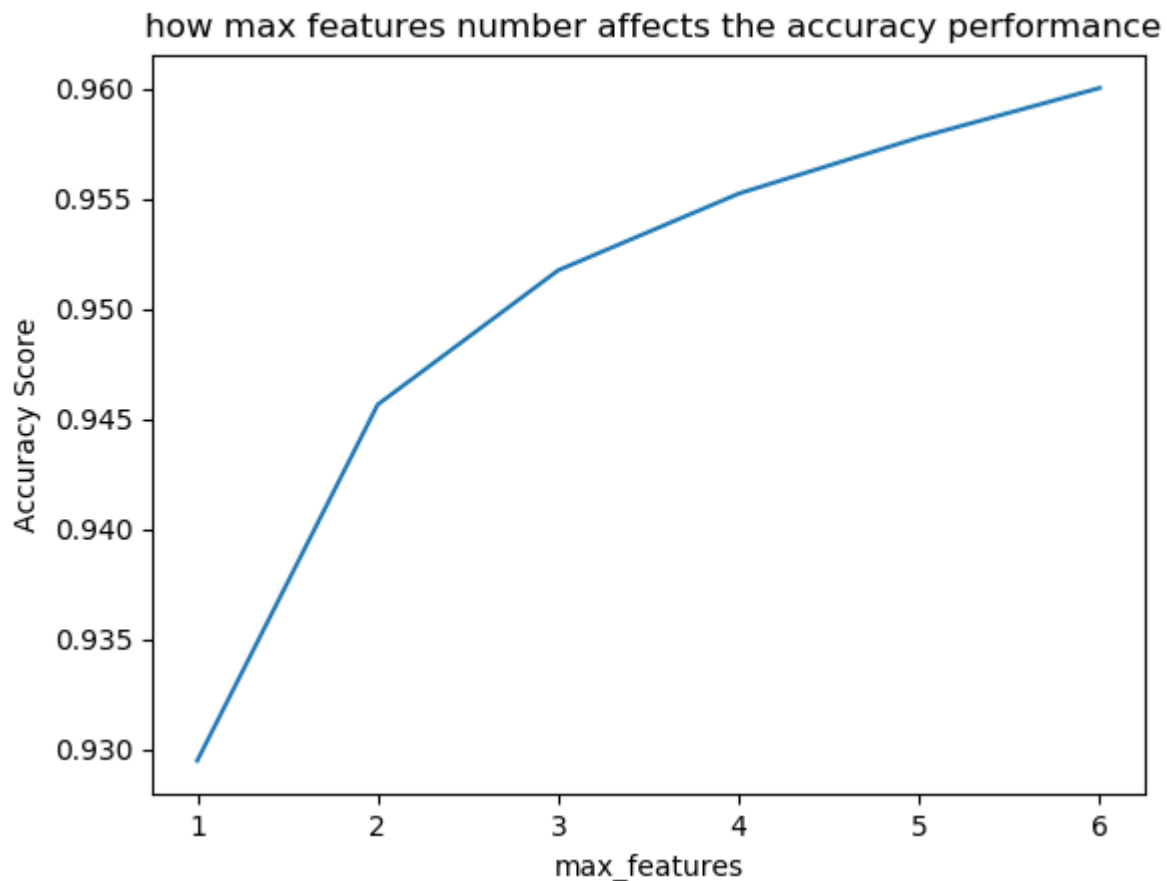
How the maximum features number affects the performance of the algorithm.

Recall, the total number of features is 6.

Therefore, applying the Random Forests algorithm several times with

the `max_features` argument being in range of 1 to 6, and then comparing the mean values for each case.

```
1  # storing all the scores and corresponding max_features arguments in the l
2  max_features_score = list()
3  max_features_number = list()
4  max_features_score_mean = list()
5  # Calculating the accuracy scores using Random Forests
6
7  for i in range(1,7):
8      for iteration in range(10):
9          forest = rf.RandomForest(n_estimators=30, max_features=i, random_s
10          forest.fit(X_train, y_train)
11          y_pred = forest.predict(X_test)
12          score = accuracy_score(y_test, y_pred)
13          max_features_score.append(score)
14          max_features_score_mean.append(np.mean(max_features_score))
15          max_features_number.append(i)
16
17  # Plotting the result
18  fig, ax = plt.subplots()
19  ax.plot(max_features_number, max_features_score_mean)
20  ax.set(xlabel="max_features", ylabel="Accuracy Score", title=" how max fea
21  ax.xaxis.set_major_locator(MaxNLocator(integer=True))
22  fig.savefig("max_features_performance.png")
23  plt.show()
```



Not surprisingly, on average the algorithm performs better when has all 6 features. Since we already have a small set of features, there is no need to specify max_depth value, because if we choose the smaller value, then the results might be underfitted. For that reason, there is no need to perform Features Selection.

Untitled