25T1 9417 HW2 Solutions

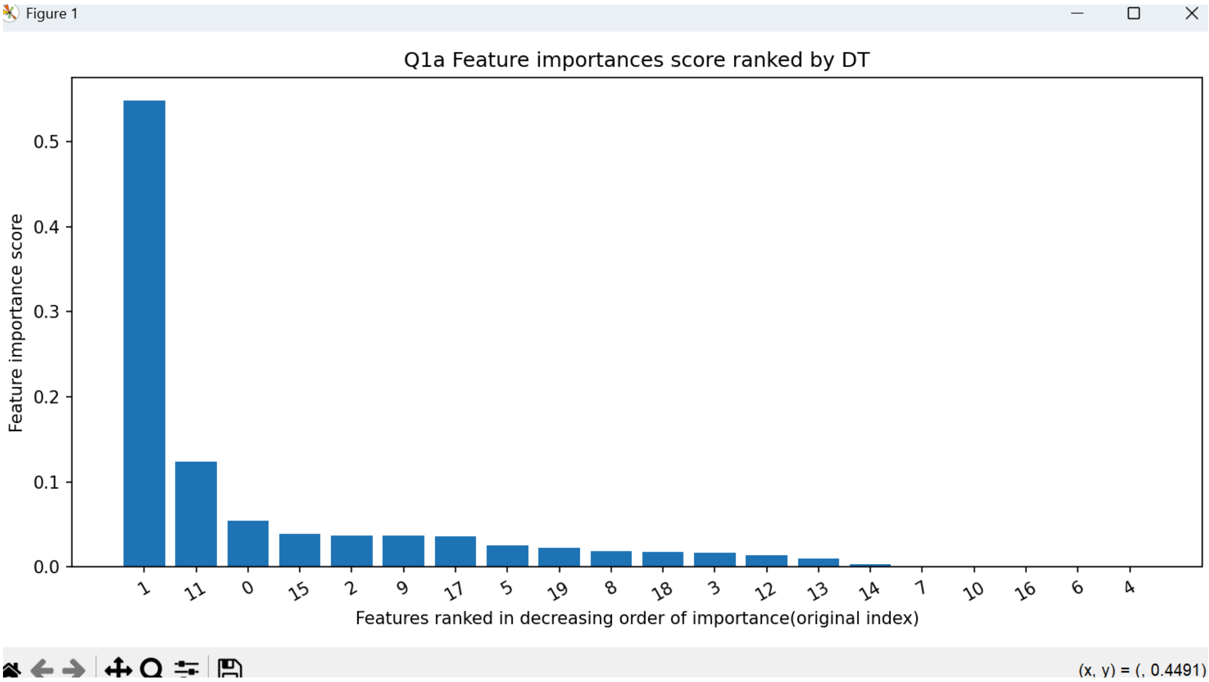
z5270589 YiJie Zhao

Question1 Explore Model-Based Feature Importance

(a)

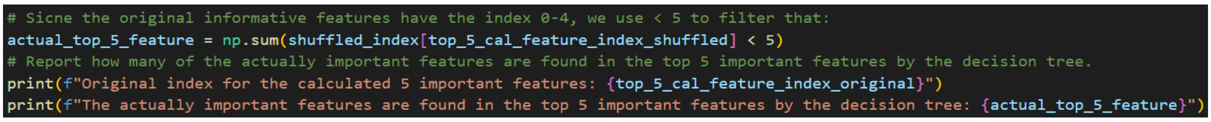
Plot a histogram with x-axis showing the features ranked in decreasing order of importance, and the y-axis showing the feature importance score:

- The histogram is shown below, which its x-axis showing the features ranked in decreasing order of importance, and its y-axis showing the feature importance score:



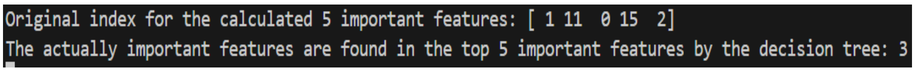
Report how many of the actually important features are found in the top 5 important features by the decision tree.

- After we get the original index of the top 5 features, we proceed these steps to get the actually important features are found in the top 5 important features by the decision tree:



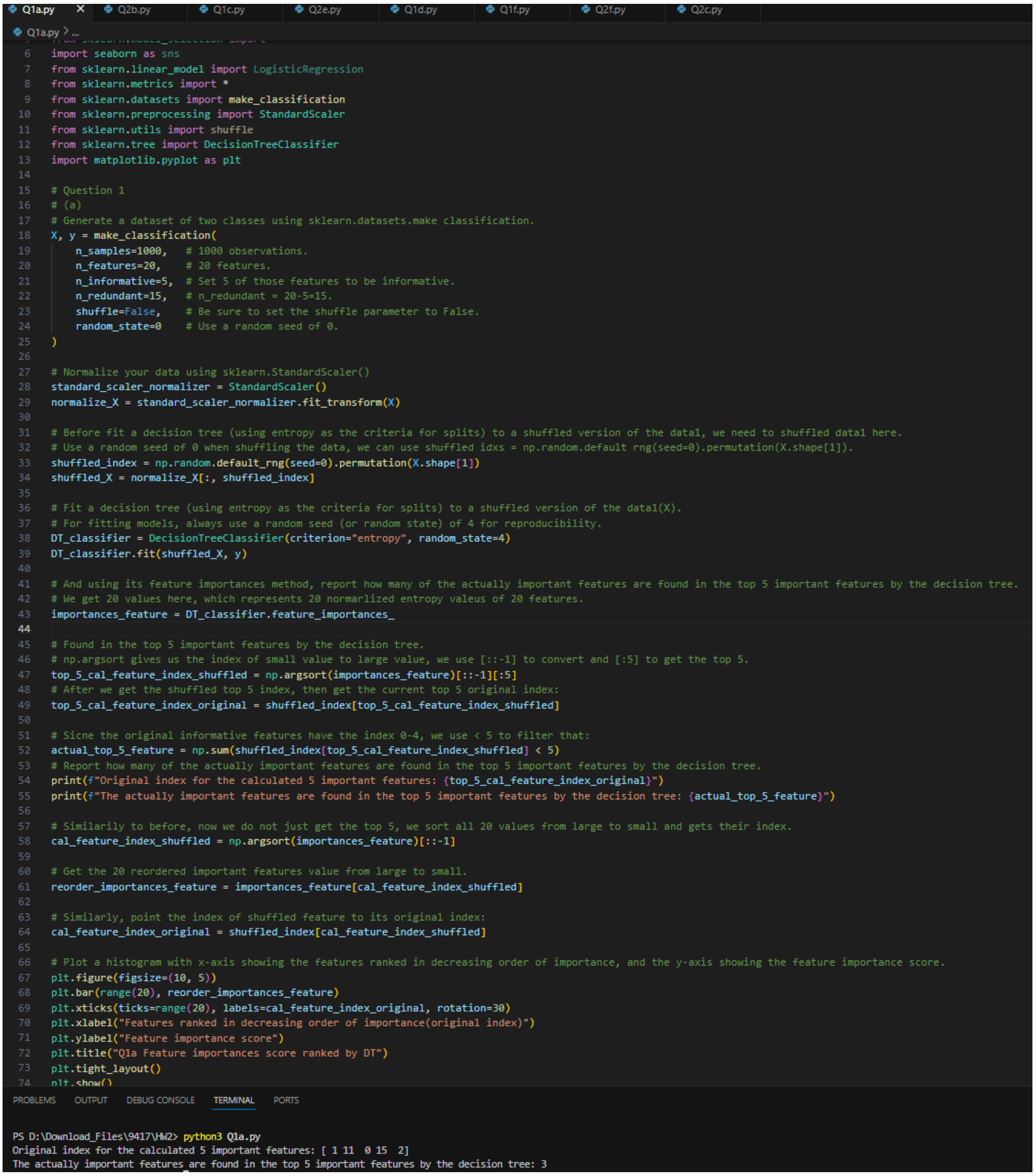
```
# Sicne the original informative features have the index 0-4, we use < 5 to filter that:
actual_top_5_feature = np.sum(shuffled_index[top_5_cal_feature_index_shuffled] < 5)
# Report how many of the actually important features are found in the top 5 important features by the decision tree.
print(f"Original index for the calculated 5 important features: {top_5_cal_feature_index_original}")
print(f"The actually important features are found in the top 5 important features by the decision tree: {actual_top_5_feature}")
```

And we can also find those 5 indexes:



```
Original index for the calculated 5 important features: [ 1 11  0 15  2]
The actually important features are found in the top 5 important features by the decision tree: 3
```

- which we can find that **3** of the actually important features are found in the top 5 important features by the decision tree, and their indexes are **[1, 0, 2]**.
- Since Q1a is the begin and we use a lot similar code in the future steps, so I provide a screenshots of the code used in this part:

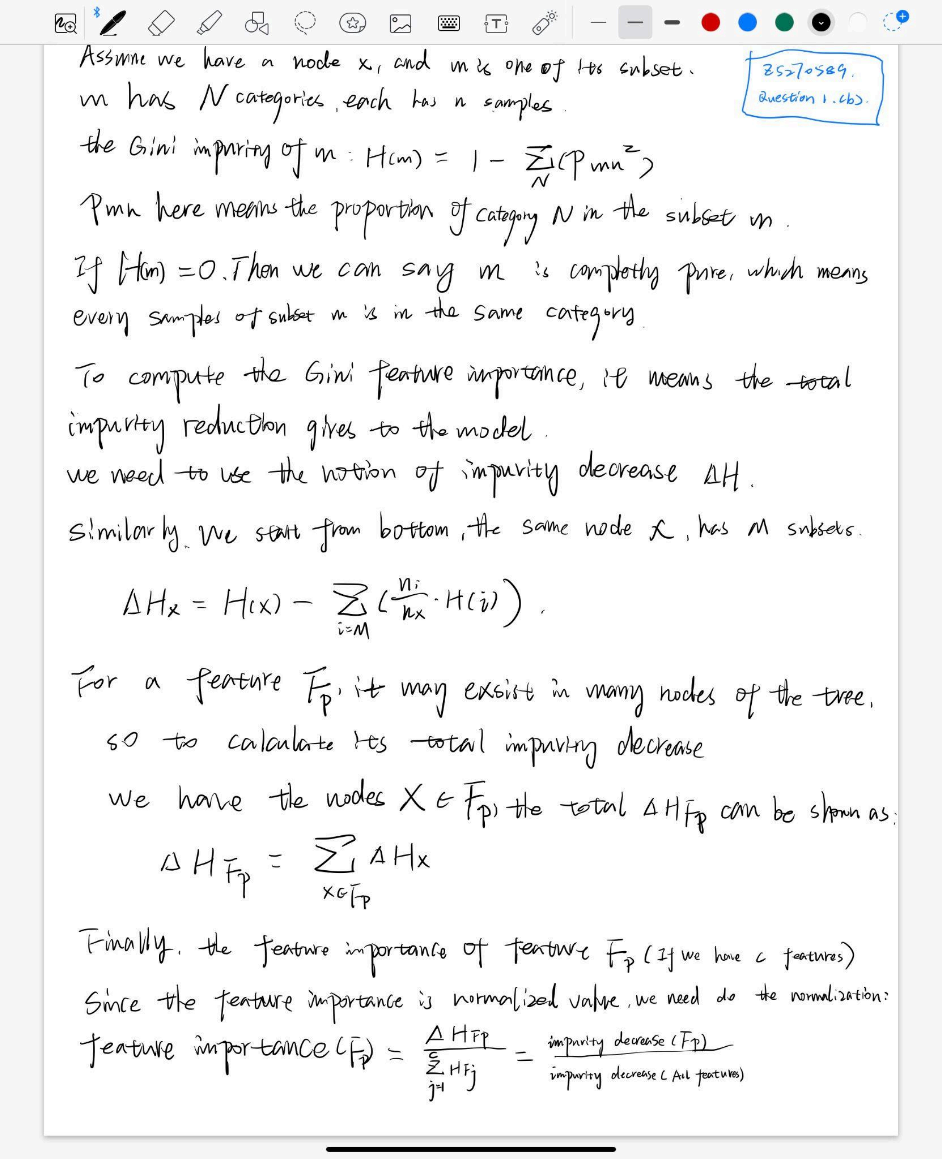


```python
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import *
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.utils import shuffle
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

# Question 1
# (a)
# Generate a dataset of two classes using sklearn.datasets.make classification.
X, y = make_classification(
    n_samples=1000,   # 1000 observations.
    n_features=20,    # 20 features.
    n_informative=5,  # Set 5 of those features to be informative.
    n_redundant=15,   # n_redundant = 20-5=15.
    shuffle=False,    # Be sure to set the shuffle parameter to False.
    random_state=0    # Use a random seed of 0.
)

# Normalize your data using sklearn.StandardScaler()
standard_scaler_normalizer = StandardScaler()
normalize_X = standard_scaler_normalizer.fit_transform(X)

# Before fit a decision tree (using entropy as the criteria for splits) to a shuffled version of the data1, we need to shuffled data1 here.
# Use a random seed of 0 when shuffling the data, we can use shuffled idxs = np.random.default rng(seed=0).permutation(X.shape[1]).
shuffled_index = np.random.default_rng(seed=0).permutation(X.shape[1])
shuffled_X = normalize_X[:, shuffled_index]

# Fit a decision tree (using entropy as the criteria for splits) to a shuffled version of the data1(X).
# For fitting models, always use a random seed (or random state) of 4 for reproducibility.
DT_classifier = DecisionTreeClassifier(criterion="entropy", random_state=4)
DT_classifier.fit(shuffled_X, y)

# And using its feature importances method, report how many of the actually important features are found in the top 5 important features by the decision tree.
# We get 20 values here, which represents 20 normarlized entropy valeus of 20 features.
importances_feature = DT_classifier.feature_importances_

# Found in the top 5 important features by the decision tree.
# np.argsort gives us the index of small value to large value, we use [::-1] to convert and [:5] to get the top 5.
top_5_cal_feature_index_shuffled = np.argsort(importances_feature)[::-1][:5]
# After we get the shuffled top 5 index, then get the current top 5 original index:
top_5_cal_feature_index_original = shuffled_index[top_5_cal_feature_index_shuffled]

# Sicne the original informative features have the index 0-4, we use < 5 to filter that:
actual_top_5_feature = np.sum(shuffled_index[top_5_cal_feature_index_shuffled] < 5)
# Report how many of the actually important features are found in the top 5 important features by the decision tree.
print(f"Original index for the calculated 5 important features: {top_5_cal_feature_index_original}")
print(f"The actually important features are found in the top 5 important features by the decision tree: {actual_top_5_feature}")

# Similarily to before, now we do not just get the top 5, we sort all 20 values from large to small and gets their index.
cal_feature_index_shuffled = np.argsort(importances_feature)[::-1]

# Get the 20 reordered important features value from large to small.
reorder_importances_feature = importances_feature[cal_feature_index_shuffled]

# Similarly, point the index of shuffled feature to its original index:
cal_feature_index_original = shuffled_index[cal_feature_index_shuffled]

# Plot a histogram with x-axis showing the features ranked in decreasing order of importance, and the y-axis showing the feature importance score.
plt.figure(figsize=(10, 5))
plt.bar(range(20), reorder_importances_feature)
plt.xticks(ticks=range(20), labels=cal_feature_index_original, rotation=30)
plt.xlabel("Features ranked in decreasing order of importance(original index)")
plt.ylabel("Feature importance score")
plt.title("Q1a Feature importances score ranked by DT")
plt.tight_layout()
plt.show()
```

```
PS D:\Download_Files\9417\HW2> python3 Q1a.py
Original index for the calculated 5 important features: [ 1 11  0 15  2]
The actually important features are found in the top 5 important features by the decision tree: 3
```

(b)

Provide a detailed explanation of how the feature importance (a.k.a. Gini importance) in the previous question are computed; use formulas to explain the exact calculation. Further, answer the following:



25570589.
Question 1.(b).

Assume we have a node x, and m is one of its subset.
m has N categories, each has n samples.

the Gini impurity of m: $H(m) = 1 - \sum_{N} (P_{mn}^2)$

P_{mn} here means the proportion of category N in the subset m.

If $H(m) = 0$. Then we can say m is completly pure, which means every samples of subset m is in the same category.

To compute the Gini feature importance, it means the ~~total~~ impurity reduction gives to the model.
we need to use the notion of impurity decrease ΔH.

similarly. We start from bottom, the same node x, has M subsets.

$$\Delta H_x = H(x) - \sum_{i=M} \left(\frac{n_i}{n_x} \cdot H(i) \right).$$

For a feature F_p, it may exsist in many nodes of the tree,
so to calculate its ~~total~~ impurity decrease
we have the nodes $X \in F_p$, the total ΔH_{F_p} can be shown as:

$$\Delta H_{F_p} = \sum_{X \in F_p} \Delta H_x$$

Finally. the feature importance of feature F_p (If we have c features)
Since the feature importance is normalized value, we need do the normalization:

$$\text{feature importance}(F_p) = \frac{\Delta H_{F_p}}{\sum_{j=1}^{c} H_{F_j}} = \frac{\text{impurity decrease }(F_p)}{\text{impurity decrease (All features)}}$$

1. What Feature importance score is assigned to a feature that is not used for any splits of the tree. Why?

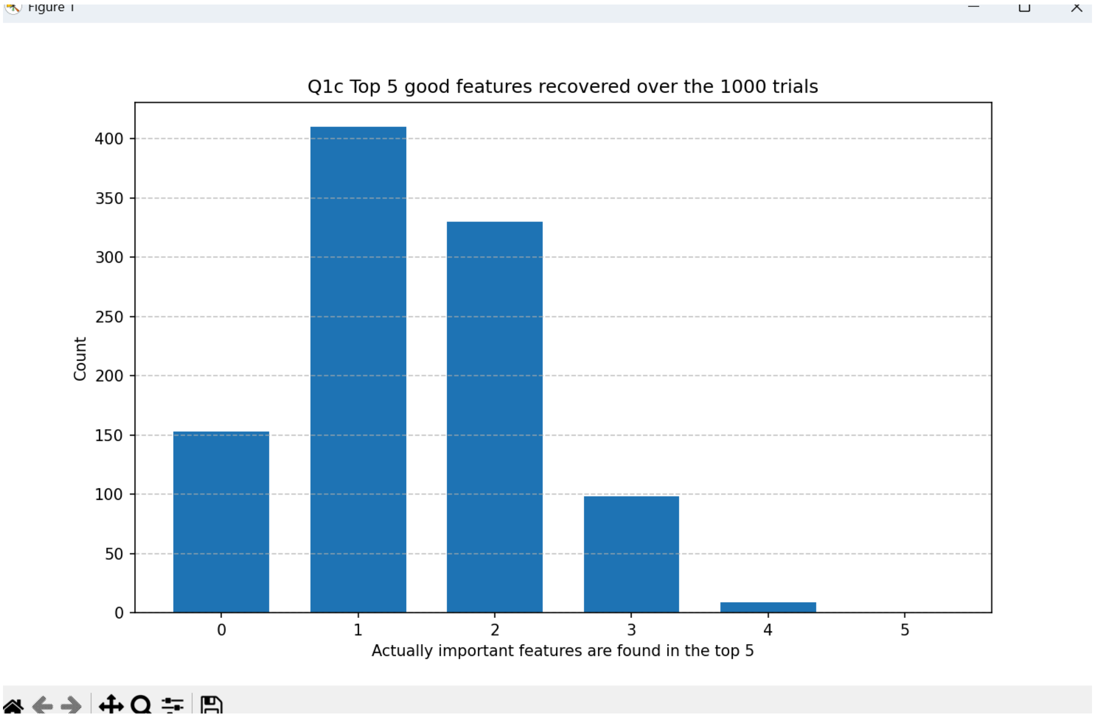
- From the picture above, we talked about the feature importance of a feature means the total impurity reduction this feature brings to the model, thus if a feature is not used for any splits of a tree, clearly its feature importance score will be **0** because it has not reduced any impurity for the DT model.

2. What does a feature importance of 0.15 mean?

- From the picture above, we know that the feature importance is a normalized value, if a feature importance is 0.15, which means that this feature contributes to 15% of the total Gini impurity reduction for our DT model.

(c)

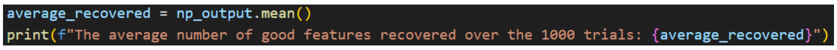
Provide a histogram of this metric over the 1000 trials.



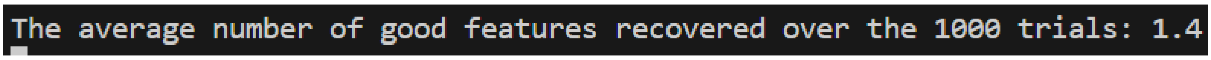
- We can see from the histogram that, based on the 1000 trials of **Decision Tree** method:
- about 415 times 1 truly important features found,
- about 330 times 2 truly important features found
- about 95 times 3 truly important features found.
- about 155 times 0 truly important features found,
- about 10 times 4 truly important features found.
- **0** time 5 truly important features found.

Report the average number of good features recovered over the 1000 trials.

- We use , and finally get the output, which the average number is **1.4**.



```
average_recovered = np_output.mean()
print(f"The average number of good features recovered over the 1000 trials: {average_recovered}")
```



```
The average number of good features recovered over the 1000 trials: 1.4
```

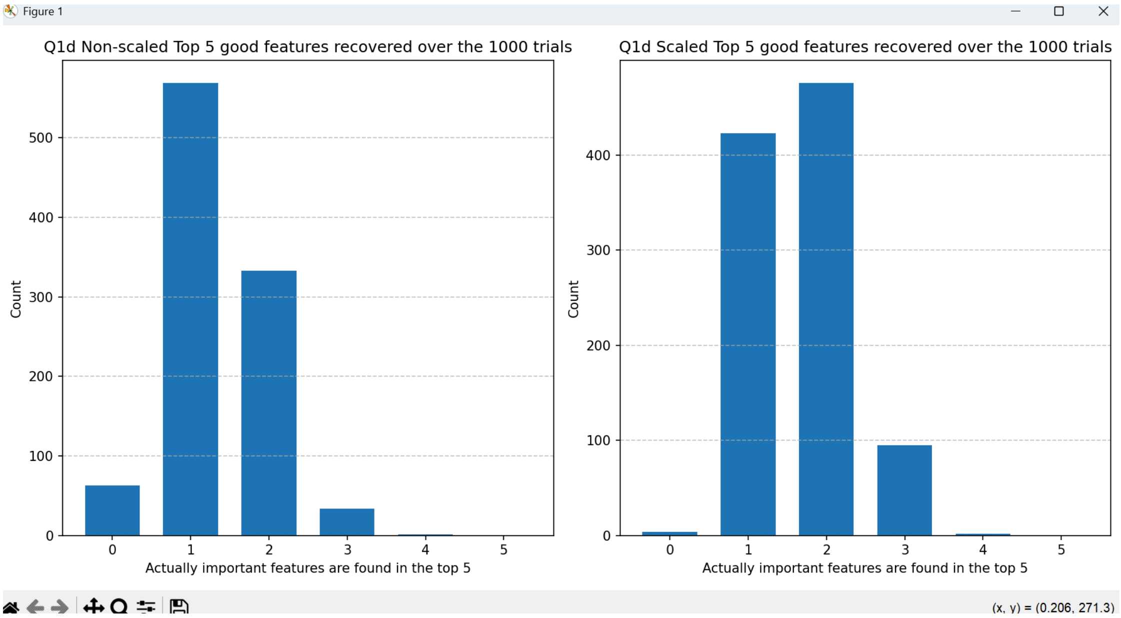
What do you think about the ability of decision trees to pick out the top features?

- According to the result above, we can find that the average number of informative features recovered over the 1000 trials is **1.4**, which means that the decision tree indeed has the ability to identify the informative features, but it also indicates that its ability to identify the informative features is **limited** since **0** time 5 truly important features found, which in 1,000 trials, on average, only about **28.0%** informative features were correctly identified.
- The reason that the ability of decision trees to pick out the top features is limited may be caused by the **overfitting issues** of the decision tree and some **noises**, when there are some noises, the decision tree might give the high importance to some redundant features and lead to this outcome.

(d)

Repeat Part(c), but now use logistic regression with no penalty. Do this once with and once Without scaling the feature matrix. As a feature importance metric, use the absolute value of the coefficient of that feature. Plot a histogram as before and report the average number of features recovered over the 1000 trials.

- The histogram is shown below:

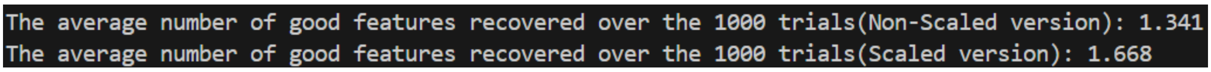


Q1d Non-scaled Top 5 good features recovered over the 1000 trials

Q1d Scaled Top 5 good features recovered over the 1000 trials

Compare the scaled and non-scaled versions.

- From the screenshot below, we can see that the average number of informative features recovered over the 1000 trials of un-scaled version is **1.341**, and the average number of informative features recovered over the 1000 trials of scaled version is **1.668**, which is **larger** than the un-scaled version. Meanwhile, we find that in non-scaled version: about **570** times **1** truly important features found and about **330** times **2** truly important features found while in scaled version, about **430** times **1** truly important features found and about **470** times **2** truly important features found, also scaled version performs better in the 3 truly important features found cases, which means the scaled version **perform better** than the un-scaled version in this informative features recovered task, I believe this is true because we know the logistic regression is depending on the gradient descent, and the scaled feature matrix can make the gradient more stable and finally lead to a better outcome.
-



```
The average number of good features recovered over the 1000 trials(Non-Scaled version): 1.341
The average number of good features recovered over the 1000 trials(Scaled version): 1.668
```

How does logistic regression compare to decision trees?

- From the previous, we know that the average number of informative features recovered from the decision tree over the 1000 trials is **1.4**, which we can see that it **performs better** than the non-scaled logistic regression case, but **performs worse** than the scaled logistic regression. The reason this happened I believe is we know that the logistic regression is a **linear classifier** and the decision tree is a **non-linear** model, so if the dataset has more linear relationship, the logistic regression is expected to perform better than the decision tree. Meanwhile, if the dataset has more complex non-linear relationships, the decision tree might take some advantages over logic regression.

(e)

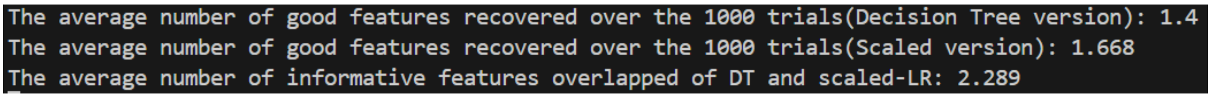
Does scaling features affect the result for decision trees? Explain.

- **No**, the scaling features do not affect the result for decision trees. Not like the logistic regression depend on the gradient descent, the factor affect the result for decision tree will be **entropy or Gini impurity** of each feature, from (b), we know that how the feature importance was computed, it only depend on the **distribution of class labels** rather than the real values of the features themselves. If you change the scaling of the features, the value of the feature importance will stay the same, and the structure of the decision tree will not change, so the scaling features does not affect the result for decision trees.

(f)

We now want to assess how often the two models (Decision trees and logistic regression (with scaling)) identify the same features as being important. Using the set-up of part (c), for each trial, record the number of overlaps for the top-5 ranked features for each of the two models.

- For each trial, we record the number of overlaps for the top-5 ranked features for each of the two models in a list, and finally we get 1000 values and calculate an average value of the number of overlaps for the top-5 ranked features for each of the two models in the 1000 trials, which is shown below:

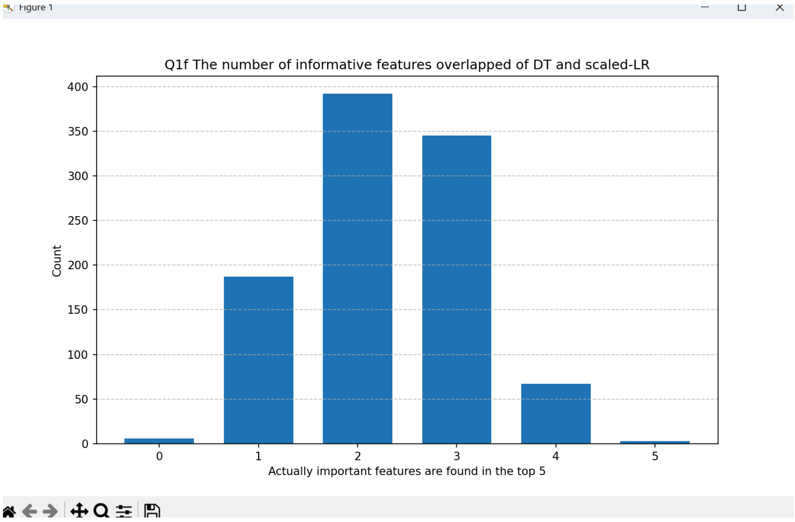


```
The average number of good features recovered over the 1000 trials(Decision Tree version): 1.4
The average number of good features recovered over the 1000 trials(Scaled version): 1.668
The average number of informative features overlapped of DT and scaled-LR: 2.289
```

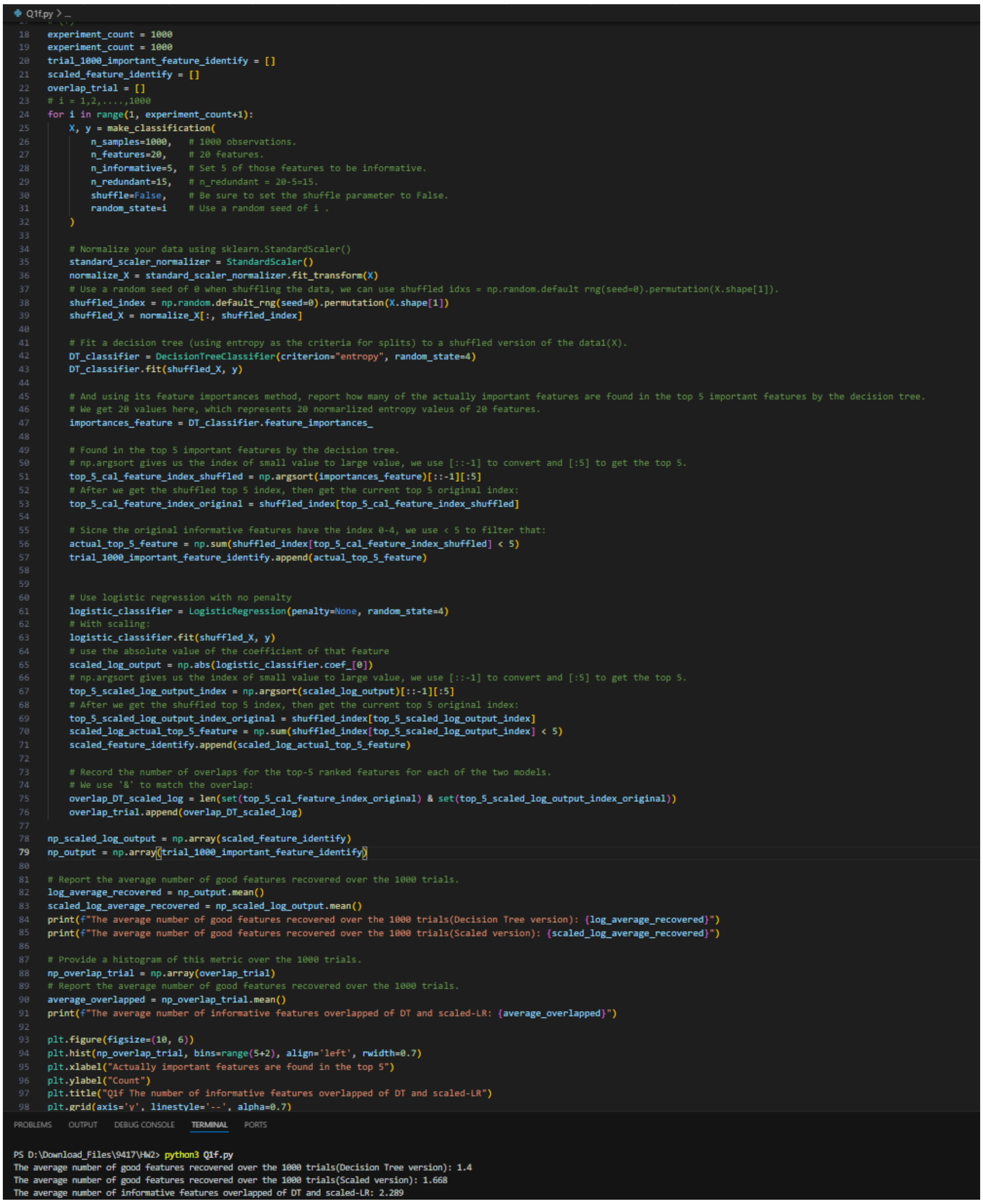
- We can see that the average value of the number of overlaps for the top-5 ranked features for each of the two models in the 1000 trials is **2.289**, which means these two models indeed has some similarities when selecting the features, but overall since they have different strategies, the selected informative features of these two models still have some difference.

Plot a histogram of the number of overlaps over all trials. For example, if on a particular trial, DT has [1, 2, 3,4,5] in its top-5, and Logistic regression has [1,2,6,7,8], the number of overlaps for this trial is 2.

- The histogram of the number of overlaps over all trials is shown below:



- From the histogram above, we can observe that during the 1000 trials,
- about 395 times 2 overlaps happened,
- about 345 times 3 overlaps happened,
- about 180 times 1 overlaps happened,
- about 65 times 4 overlaps happened.
- about 10 times 0 overlaps happened.
- 5 overlap cases are rare in the 1000 trials(about 5 less than 10).
- This happened because although Decision Tree and Logistic Regression have different strategies(reduce impurity and absolute value of coefficient), some informative features are **inherently more informative** and tend to be selected by both models, so the number of overlap is hard to be 0. But as the strategy is different, it has very little chance to have the 5 overlaps, which mostly these two models have 2, 3, 1, 4 overlaps.
- The work for this part is relatively heavy, so I will provide a screenshots of the code used in this part:



```python
experiment_count = 1000
experiment_count = 1000
trial_1000_important_feature_identify = []
scaled_feature_identify = []
overlap_trial = []
# i = 1,2,....,1000
for i in range(1, experiment_count+1):
    X, y = make_classification(
        n_samples=1000,   # 1000 observations.
        n_features=20,    # 20 features.
        n_informative=5,  # Set 5 of those features to be informative.
        n_redundant=15,   # n_redundant = 20-5=15.
        shuffle=False,    # Be sure to set the shuffle parameter to False.
        random_state=i    # Use a random seed of i .
    )

    # Normalize your data using sklearn.StandardScaler()
    standard_scaler_normalizer = StandardScaler()
    normalize_X = standard_scaler_normalizer.fit_transform(X)
    # Use a random seed of 0 when shuffling the data, we can use shuffled idxs = np.random.default rng(seed=0).permutation(X.shape[1]).
    shuffled_index = np.random.default_rng(seed=0).permutation(X.shape[1])
    shuffled_X = normalize_X[:, shuffled_index]

    # Fit a decision tree (using entropy as the criteria for splits) to a shuffled version of the data1(X).
    DT_classifier = DecisionTreeClassifier(criterion="entropy", random_state=4)
    DT_classifier.fit(shuffled_X, y)

    # And using its feature importances method, report how many of the actually important features are found in the top 5 important features by the decision tree.
    # We get 20 values here, which represents 20 normarlized entropy valeus of 20 features.
    importances_feature = DT_classifier.feature_importances_

    # Found in the top 5 important features by the decision tree.
    # np.argsort gives us the index of small value to large value, we use [::-1] to convert and [:5] to get the top 5.
    top_5_cal_feature_index_shuffled = np.argsort(importances_feature)[::-1][:5]
    # After we get the shuffled top 5 index, then get the current top 5 original index:
    top_5_cal_feature_index_original = shuffled_index[top_5_cal_feature_index_shuffled]

    # Sicne the original informative features have the index 0-4, we use < 5 to filter that:
    actual_top_5_feature = np.sum(shuffled_index[top_5_cal_feature_index_shuffled] < 5)
    trial_1000_important_feature_identify.append(actual_top_5_feature)


    # Use logistic regression with no penalty
    logistic_classifier = LogisticRegression(penalty=None, random_state=4)
    # With scaling:
    logistic_classifier.fit(shuffled_X, y)
    # use the absolute value of the coefficient of that feature
    scaled_log_output = np.abs(logistic_classifier.coef_[0])
    # np.argsort gives us the index of small value to large value, we use [::-1] to convert and [:5] to get the top 5.
    top_5_scaled_log_output_index = np.argsort(scaled_log_output)[::-1][:5]
    # After we get the shuffled top 5 index, then get the current top 5 original index:
    top_5_scaled_log_output_index_original = shuffled_index[top_5_scaled_log_output_index]
    scaled_log_actual_top_5_feature = np.sum(shuffled_index[top_5_scaled_log_output_index] < 5)
    scaled_feature_identify.append(scaled_log_actual_top_5_feature)

    # Record the number of overlaps for the top-5 ranked features for each of the two models.
    # We use '&' to match the overlap:
    overlap_DT_scaled_log = len(set(top_5_cal_feature_index_original) & set(top_5_scaled_log_output_index_original))
    overlap_trial.append(overlap_DT_scaled_log)

np_scaled_log_output = np.array(scaled_feature_identify)
np_output = np.array(trial_1000_important_feature_identify)

# Report the average number of good features recovered over the 1000 trials.
log_average_recovered = np_output.mean()
scaled_log_average_recovered = np_scaled_log_output.mean()
print(f"The average number of good features recovered over the 1000 trials(Decision Tree version): {log_average_recovered}")
print(f"The average number of good features recovered over the 1000 trials(Scaled version): {scaled_log_average_recovered}")

# Provide a histogram of this metric over the 1000 trials.
np_overlap_trial = np.array(overlap_trial)
# Report the average number of good features recovered over the 1000 trials.
average_overlapped = np_overlap_trial.mean()
print(f"The average number of informative features overlapped of DT and scaled-LR: {average_overlapped}")

plt.figure(figsize=(10, 6))
plt.hist(np_overlap_trial, bins=range(5+2), align='left', rwidth=0.7)
plt.xlabel("Actually important features are found in the top 5")
plt.ylabel("Count")
plt.title("Q1f The number of informative features overlapped of DT and scaled-LR")
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

```
PS D:\Download_Files\9417\HW2> python3 Q1f.py
The average number of good features recovered over the 1000 trials(Decision Tree version): 1.4
The average number of good features recovered over the 1000 trials(Scaled version): 1.668
The average number of informative features overlapped of DT and scaled-LR: 2.289
```

-

(g)

The approaches considered so far are called ”model-based” feature importance methods, since they define importance with respect to a particular algorithm/model being used.
Discuss some potential disadvantages of using a model-based approach if your goal is to uncover truly important features, referring to the previous exercises for evidence.

- The model-based approach depends on the model we selected. For the Decision Tree, it selects the important features by calculating the impurity reduction of that feature.
- For Logistic Regression, it selects the important features by calculating the absolute value of the coefficient of that feature.
- From previous questions, we know that mostly only 2/3/1 truly important features were selected by Decision tree and logistic regression models, and the situation of all 5 truly important features being selected is basically hard to occur.
- Since each different model calculates the features differently, **some features may rank highly in one model but lowly in another**, leading to inconsistent selected important features, thus the outcomes are not stable, this is one of the potential disadvantages of using a model-based approach if my goal is to uncover truly important features.
- Meanwhile, we know that the model-based methods always select the features that contribute most to the model, but those features may not be those truly important features, **if our goal is to uncover all truly important features**, that will be the potential disadvantage.

For example, suppose that you are studying a rare genetic disease and that the 20 features represent specific genetic features, only 5 of which are truly associated with the disease.

- For this task, our final goal is to successfully select the 5 genetic features which are truly associated with the disease. From the previous experience, we know that if we use the model-based feature importance method, it has a large possibility that only 2 or 3 truly features associated with the disease will be selected, other unrelated features may contribute more to the model than the truly features so they are chosen as the important features.
- According to the results from previous questions, we also can observe that in each trial, there are different unrelated features that are finally chosen as the important features by the model. In this question, this means **we will always miss some truly features associated with the disease**. This can also reveal the potential disadvantages of using a model-based approach if our goal is to uncover truly important features.

Further, discuss the effect of the number of redundant features used when creating the data set.

- As we talked above, due to the mechanism of the”model-based” feature importance methods selecting important features, they always select the features that contribute most to the model, but those features may not be those truly important features. So we know that **the more redundant features we have, the more possibility that we**

will miss the truly important features, because those redundant features may contribute more to the model than the truly important features and the truly important features will be eliminated. Similarly, the fewer redundant features, the truly important features will have fewer competitors, thus it has more possibility to be chosen by the model and gain a better accuracy.

Question 2 Greedy Feature Selection

We now consider a different approach to feature selection known as backward selection. In backward selection, we:

- **1. start with all features in the model**
- **2. At each round, we remove the j-th feature from the model based on the drop in the value of a certain metric. We eliminate the feature corresponding to the smallest drop in the metric.**
- **3. We repeat step 2 until there are no features left.**

(a)

Why do you think this is referred to as a greedy feature importance algorithm?

- The notion of a greedy algorithm is when solving a problem, it always **makes the best choice at the moment**, which greedy algorithm does **not consider the overall optimal solution**, and the solution it achieves is a **local optimal solution in each stage**.
- And when referring to the feature selection, we repeatedly remove the j-th feature from the model based on the drop in the value of a certain metric. We eliminate the feature corresponding to the smallest drop in the metric, which has the **same strategy** as the greedy algorithm. Meanwhile, as we know the removed features will not be reconsidered in our future steps, this is also the same with the greedy method. Finally, we might miss the overall best overall feature subset by using this method, which is also considered as a typical characteristic of greedy algorithms, so we can conclude that this is referred to as a greedy feature importance algorithm.

What do you think are some of the pitfalls of greedy algorithms in this context?

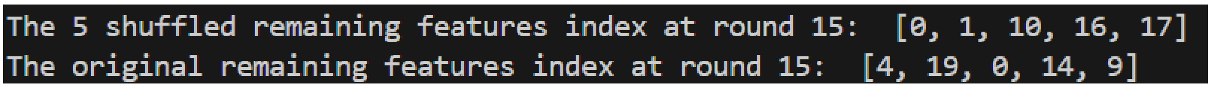
- As we know, greedy algorithms will **not consider the interactions between features well**, which means there may exist some features that contribute less individually but contribute significantly when combined with other features. Therefore, after deleting these features in each round of greedy algorithm, we are likely to miss these features that contribute highly to the model in subsequent combinations.
- Another pitfall is that greedy algorithms in this context are likely to get the local optimal solution in each turn, and ultimately we might **miss the overall optimal solution**. When we are selecting the best features, this might be a concern.
- Finally I believe the **cost** is another pitfall, compared with Decision Tree. The greedy algorithms in this context need to retrain the model at every turn, which has a **huge computational cost** concern.

(b)

Using the same set-up as Question 1 part 1(a) write code implementing the backward elimination algorithm. Use a logistic regression model with no penalty, and the same metric as in Question 1 part (d). Be sure to generate the data without shuffling but then to shuffle the data before fitting the model.

Report the remaining features at round 15 (that is, when only 5 features are left).

- The original index of the the remaining features at round 15 can be shown as below:

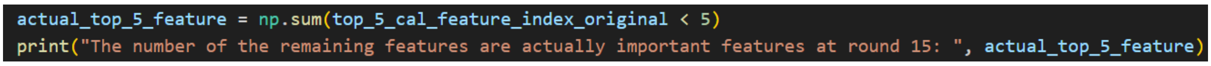


```
The 5 shuffled remaining features index at round 15:  [0, 1, 10, 16, 17]
The original remaining features index at round 15:  [4, 19, 0, 14, 9]
```

- According to the picture above, we can find that the original index of remaining features at round 15 is **[4, 19, 0, 14, 9]**.

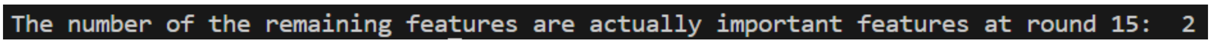
How many of these are actually important features?

- Same as question 1, we use this to get the number of actually important features:



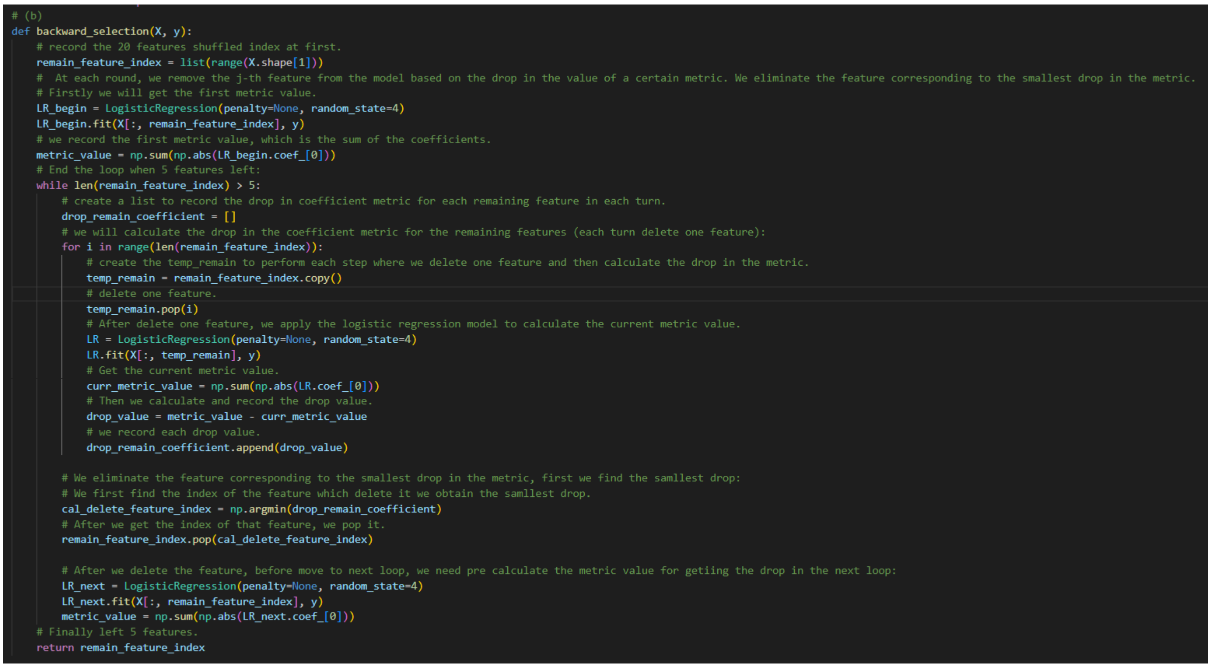
```python
actual_top_5_feature = np.sum(top_5_cal_feature_index_original < 5)
print("The number of the remaining features are actually important features at round 15: ", actual_top_5_feature)
```

And the number of actually important features is:



```
The number of the remaining features are actually important features at round 15:  2
```

- We can see that the number of actually important features is 2, which also matches with the original index of remaining features at round 15 is **[4, 19, 0, 14, 9]**, of which the index 0 and 4 features are the actually important features.
- I believe this outcome is reasonable, since I applied backward elimination algorithm, it will repeatedly remove the j-th feature from the model based on the drop in the value of a certain metric and eliminate the feature corresponding to the smallest drop in the metric as the code shown below:



```python
# (b)
def backward_selection(X, y):
    # record the 20 features shuffled index at first.
    remain_feature_index = list(range(X.shape[1]))
    #  At each round, we remove the j-th feature from the model based on the drop in the value of a certain metric. We eliminate the feature corresponding to the smallest drop in the metric.
    # Firstly we will get the first metric value.
    LR_begin = LogisticRegression(penalty=None, random_state=4)
    LR_begin.fit(X[:, remain_feature_index], y)
    # we record the first metric value, which is the sum of the coefficients.
    metric_value = np.sum(np.abs(LR_begin.coef_[0]))
    # End the loop when 5 features left:
    while len(remain_feature_index) > 5:
        # create a list to record the drop in coefficient metric for each remaining feature in each turn.
        drop_remain_coefficient = []
        # we will calculate the drop in the coefficient metric for the remaining features (each turn delete one feature):
        for i in range(len(remain_feature_index)):
            # create the temp_remain to perform each step where we delete one feature and then calculate the drop in the metric.
            temp_remain = remain_feature_index.copy()
            # delete one feature.
            temp_remain.pop(i)
            # After delete one feature, we apply the logistic regression model to calculate the current metric value.
            LR = LogisticRegression(penalty=None, random_state=4)
            LR.fit(X[:, temp_remain], y)
            # Get the current metric value.
            curr_metric_value = np.sum(np.abs(LR.coef_[0]))
            # Then we calculate and record the drop value.
            drop_value = metric_value - curr_metric_value
            # we record each drop value.
            drop_remain_coefficient.append(drop_value)

        # We eliminate the feature corresponding to the smallest drop in the metric, first we find the samllest drop:
        # We first find the index of the feature which delete it we obtain the samllest drop.
        cal_delete_feature_index = np.argmin(drop_remain_coefficient)
        # After we get the index of that feature, we pop it.
        remain_feature_index.pop(cal_delete_feature_index)

        # After we delete the feature, before move to next loop, we need pre calculate the metric value for getiing the drop in the next loop:
        LR_next = LogisticRegression(penalty=None, random_state=4)
        LR_next.fit(X[:, remain_feature_index], y)
        metric_value = np.sum(np.abs(LR_next.coef_[0]))
    # Finally left 5 features.
    return remain_feature_index
```

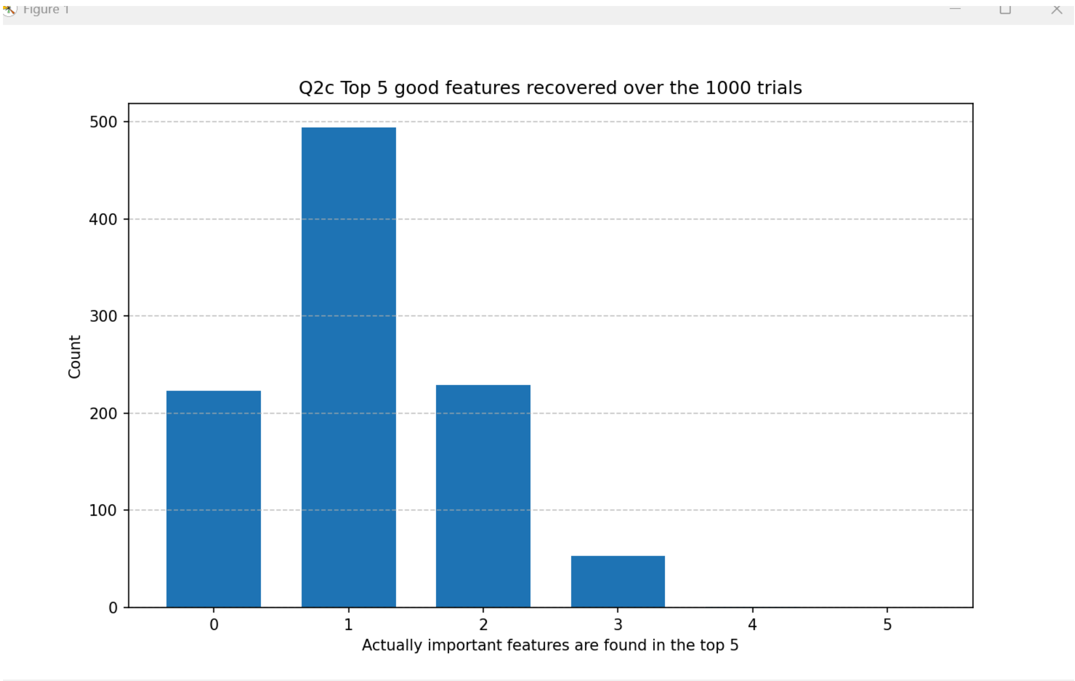
-

- In this situation, we are expected to get the **local optimum solution** and highly possibility to miss the overall solution, so I believe that our outcome **2** of truly important features were found is a reasonable answer in this question.

(c)

Repeat part (a) for 1000 trials (similar to what is done in Q1 (c)). Plot a histogram of the number of important features recovered.

- We took the similar steps as Q1c. This is the histogram of the number of important features recovered:



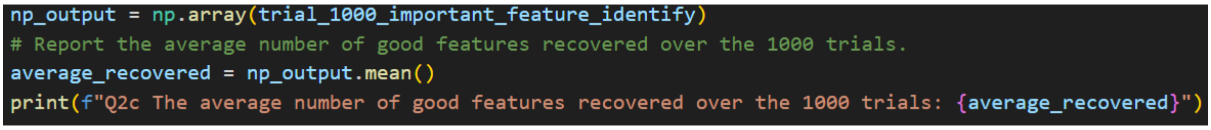
-

We can see from the histogram that, based on the 1000 trials of backward elimination algorithm:

- about 490 times 1 truly important features found,
- about 230 times 2 truly important features found,
- about 210 times 0 truly important features found,
- about 50 times 3 truly important features found.
- about **0** times 4 truly important features found.
- **0** time 5 truly important features found.
- I believe the method backward elimination algorithm is **limited** to select the truly important features, in most cases it can only select 1 or 2 truly important features and not once was 5 truly important features are all identified.

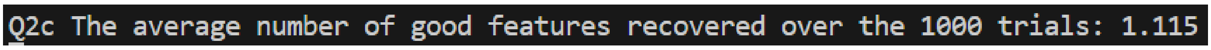
And report the average number of recovered features.

- We use this to get the average number of recovered features,



```
np_output = np.array(trial_1000_important_feature_identify)
# Report the average number of good features recovered over the 1000 trials.
average_recovered = np_output.mean()
print(f"Q2c The average number of good features recovered over the 1000 trials: {average_recovered}")
```

- which the average number of recovered features is **1.313**.



```
Q2c The average number of good features recovered over the 1000 trials: 1.115
```

The average number **1.115** also be indicated with the histogram above, in the 1000 trials, averagely only about **1.115** truly important features can be identified by using the backward elimination algorithm, and about 95% of the trials recover at most two truly important features. We can conclude that this method in feature selection is **limited**.

(d)

Another approach is called best subset selection. This model generates all possible subsets, trains a model on each subset, evaluates the performance and returns the subset with the highest performance. For example, at the t-th round, we consider all subsets with t features.

How does this algorithm compare to backward selections?

- The approach best subset selection is the method that generates all possible feature subsets and trains a model for each subset, after that we will evaluate the performance of each model and select the best performing feature subset.
- Compared with backward selection, we know that the backward selection algorithm is a greedy algorithm and it will eventually reach a **local optimum** solution. But the best subset selection algorithm will enumerate all the cases, so it is expected to reach the true **global optimal** solution.

Will it always outperform backward elimination?

- Since we know the best subset selection algorithm is expected to reach the true global optimal solution while the backward selection is expected to reach a local optimum solution, usually the best subset selection algorithm outperform than backward elimination, but when we has a very **large-scale dataset**, the best subset selection algorithm will get struggled since it will enumerate all possible subsets exponentially and trains a model for each subset. So usually the best subset selection algorithm outperforms the backward elimination in **accuracy**, but when we have a **large dataset** to proceed, the situation might be different.

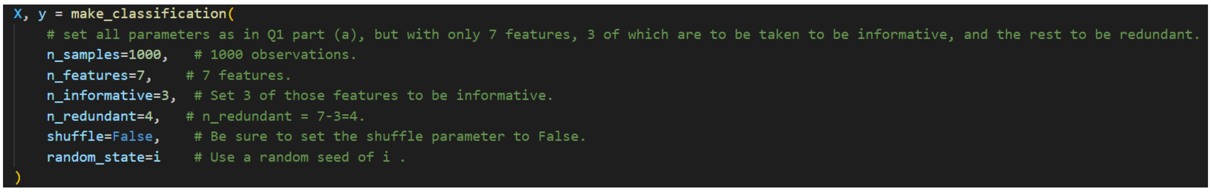
What are some disadvantages of this approach?

- As we talked above, there is always a **cost problem** when we use the best subset selection algorithm, which means as we have more features, the cost will increase **exponentially**. If we will handle 20 features as we did in previous questions, we will have 2^20 subsets, which is a huge number to handle and it will take **much longer training time** compared with decision tree or logistic regression methods. If the number of features continues to get bigger to 80, it is basically difficult for the best subset selection algorithm to implement.

(e)

Implement best subset selection in code. Repeat part (c) using your best subset implementation.

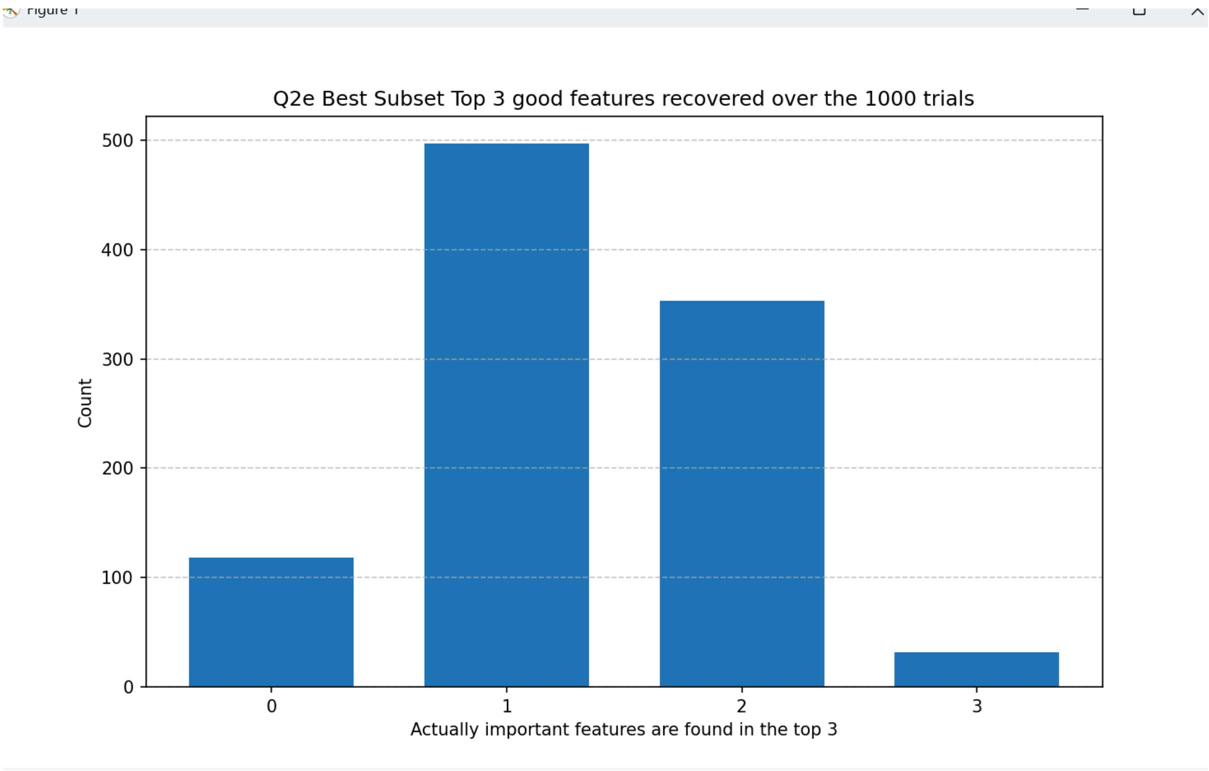
For computational reasons, set all parameters as in Q1 part (a), but with only 7 features, 3 of which are to be taken to be informative, and the rest to be redundant.

- We fix the n_features to **7**, n_reduntant to **4** and n_informative to **3** in Q2e.
- 

```
X, y = make_classification(
    # set all parameters as in Q1 part (a), but with only 7 features, 3 of which are to be taken to be informative, and the rest to be redundant.
    n_samples=1000,   # 1000 observations.
    n_features=7,    # 7 features.
    n_informative=3,  # Set 3 of those features to be informative.
    n_redundant=4,   # n_redundant = 7-3=4.
    shuffle=False,    # Be sure to set the shuffle parameter to False.
    random_state=i    # Use a random seed of i .
)
```

Plot a histogram as before.

- The histogram is shown below:

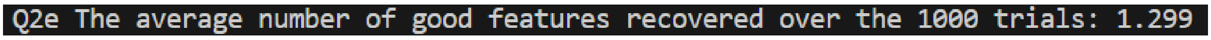


We can see from the histogram that, based on the 1000 trials of best subset selection algorithm:

- about 495 times 1 truly important features found,
- about 365 times 2 truly important features found
- about 110 times 0 truly important features found,
- about **30** times **all 3 truly important features** found.
- Compared with the previous questions, I believe this method **performs better in the small datasets**, in which about 89% trials can find at least 1 truly important feature and we have 30 times all three truly features found in the 1000 trials, this is a significant improvement compared with previous question results. This happened reasonably as we talked in Q2d, as it generates all possible feature subsets and trains a model for each subset, after that we will evaluate the performance of each model and select the best performing feature subset, we have a heavier cost and ultimately we have better results.

And report the average number of recoveries. Comment on your results.

- The average number of recoveries is shown below:



```
Q2e The average number of good features recovered over the 1000 trials: 1.299
```

- The average number **1.299** can also be indicated with the histogram above, in the 1000 trials, averagely only about **1.299** truly important features can be identified by using the best subset algorithm, and about **1.299/3 = 43.33%** of the trials recover **all three** truly important features. We can conclude that the best subset algorithm of feature selection on small datasets **has some advantages**.

(f)

An alternative approach to feature importance is known as the Permutation Feature Importance score, implemented in sklearn.inspection.permutation importance. Read the documentation and provide a detailed explanation of how permutation importance works.

- According to the documentation, we know that the method permutation feature importance works as firstly we choose to train a model (DT or LR or others), and then calculate the baseline prediction performance of the model. After that, we will operate on each feature. For each feature,we will independently **permute its values** to make it lose its original relationship with the target variable while other features remain unchanged. Then we will recalculate the predictions of the model with that permuted feature. After that, we will get the **Permutation Feature Importance score** based on **the degree of performance decline.** Then based on the Permutation Feature Importance score, we can evaluate the importance of the original feature. If the score drops significantly after permuting a feature, it means that this feature is very important. If the performance of a feature is almost unchanged after permuting, it means that this feature is not important.

Compare it to the techniques studied so far in this homework, and explain why we refer to this as a model independent metric.

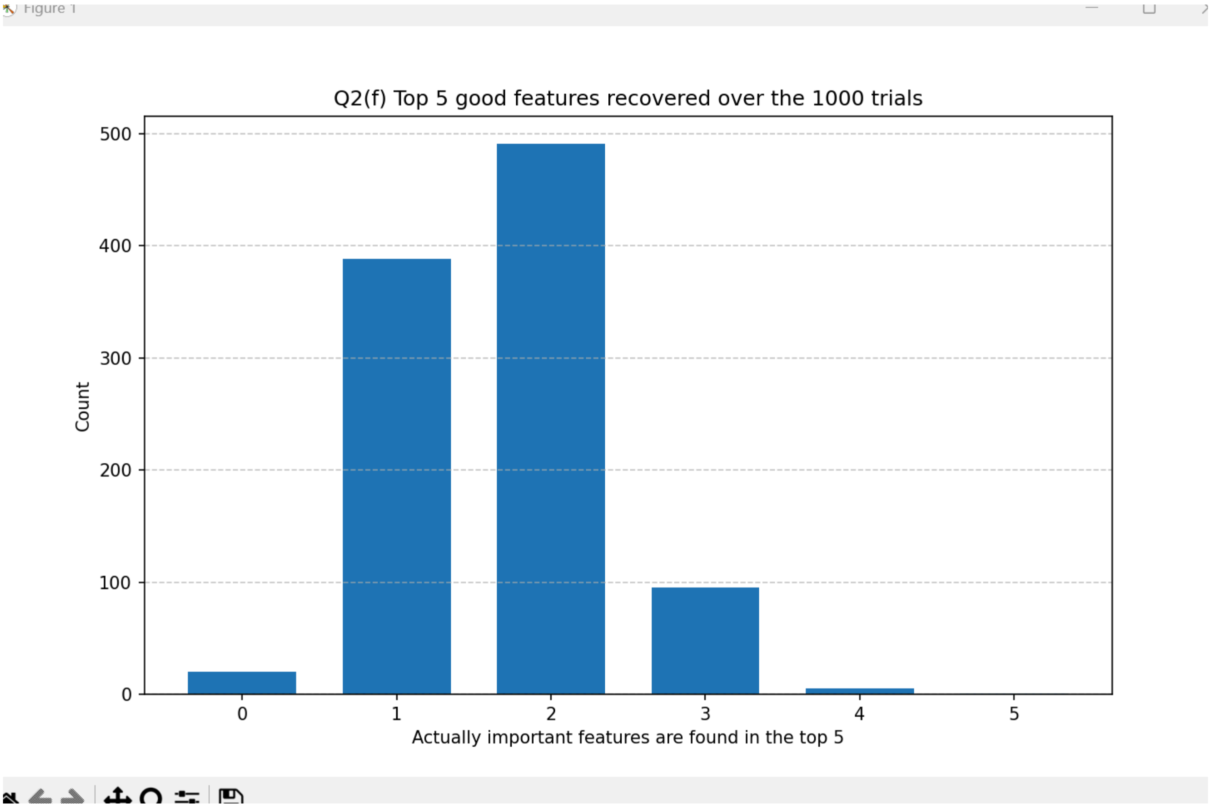
- We refer to this as a model independent metric because it can be applied to any model with predictions for the feature importance assessment, the model can be either logistic regression or decision tree. It does not depend on the training process of a certain model and does not need to know how the model was trained or what kind of model was used. Once we get the precision performances, we can achieve the Permutation Feature Importance score and be able to evaluate the feature importance.

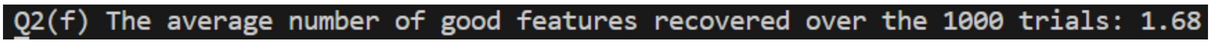
Do you think it's more or less fair to compare logistic regression and decision trees using this metric?

- I believe it's **more** fair to compare logistic regression and decision trees using this metric. Previously to evaluate the feature importance, we used the absolute sum coefficient values for logistic regression and we use the feature_importances_ for decision tree, which these two are all indicators within their corresponding model, if we change another model, these indicators will not be able to used any more.
- And we know that this method is a model independent metric, which can be applied to any model with predictions, thus to compare logistic regression and decision trees, I believe using this method is more fair.

Finally, using the sklearn implementation, re-do part Q2(c) using this new feature importance metric. Similar to before, use 20 features, with 5 to be set as informative and the rest as redundant.

- The histogram is shown below:



- We can see from the histogram that, based on the 1000 trials of Permutation Feature Importance score:
- about 385 times 1 truly important features found,
- about 490 times 2 truly important features found
- about 95 times 3 truly important features found.
- about 20 times 0 truly important features found,
- about 5 times 4 truly important features found.
- 0 time 5 truly important features found.
- The average number of recoveries is shown below:
- 

```
Q2(f) The average number of good features recovered over the 1000 trials: 1.68
```