

Assignment2

Q1(a) There is the code to complete the request:

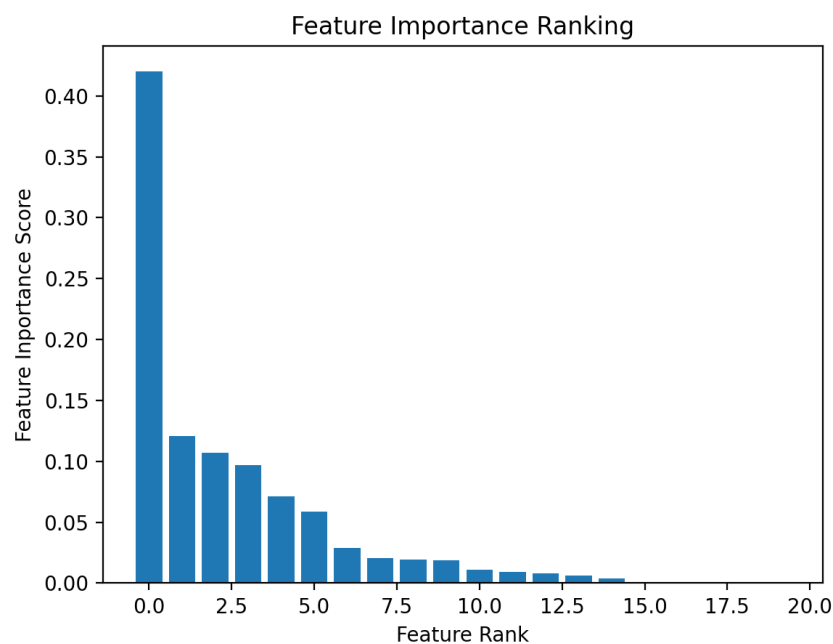
```
1. import pandas as pd
2. import numpy as np
3. import random
4. import matplotlib.pyplot as plt
5. from sklearn.datasets import make_classification
6. from sklearn.preprocessing import StandardScaler
7. from sklearn.tree import DecisionTreeClassifier
8.
9. #Question 1a
10. #generate data
11. X, y = make_classification(n_samples=1000, n_features=20, n_informative=5,
n_redundant=15, shuffle=False, random_state=4)
12.
13. #Normalize the data
14. scaler = StandardScaler()
15. scaler.fit(X)
16. X = scaler.transform(X)
17.
18. #shuffle the data
19. shuffle_idx = np.random.default_rng(seed=0).permutation(X.shape[1])
20. X_shff = X[:,shuffle_idx]
21.
22. #Decision Tree Classifier
23. clf = DecisionTreeClassifier(criterion='entropy', random_state=4)
24. clf.fit(X_shff, y)
25. importances = clf.feature_importances_
26. #descending order
27. indices = np.argsort(importances)[::-1][:5] #top 5 features
28. #map the top 5 features to the original data
29. top5 = shuffle_idx[indices]
30. #count the number of times each feature is in the top 5
31. count=np.sum(top5<5)
32. print(f"Number of true informative features in top 5: {count}")
33.
34. #plot the top 5 features
35. plt.bar(range(len(importances)), importances[np.argsort(importances)[::-1]])
36. plt.xlabel('Feature Rank')
37. plt.ylabel('Feature Inportance Score')
38. plt.title('Feature Importance Ranking')
39. plt.show()
```

```

C:\Users > 11412 > Desktop > COMP9417 > Assignment2 > HW2.py > ...
1  import pandas as pd
2  import numpy as np
3  import matplotlib.pyplot as plt
4  from sklearn.datasets import make_classification
5  from sklearn.preprocessing import StandardScaler
6  from sklearn.tree import DecisionTreeClassifier
7
8  #Question 1a
9  #generate data
10 X, y = make_classification(n_samples=1000, n_features=20, n_informative=5, n_redundant=15, shuffle=False, random_state=4)
11
12 #Normalize the data
13 scaler = StandardScaler()
14 scaler.fit(X)
15 X = scaler.transform(X)
16
17 #shuffle the data
18 shuffle_idx = np.random.default_rng(seed=0).permutation(X.shape[1])
19 X_shff = X[:,shuffle_idx]
20
21 #Decision Tree Classifier
22 clf = DecisionTreeClassifier(criterion='entropy', random_state=4)
23 clf.fit(X_shff, y)
24 importances = clf.feature_importances_
25 #descending order
26 indices = np.argsort(importances)[::-1][5] #top 5 features
27 #map the top 5 features to the original data
28 top5 = shuffle_idx[indices]
29
30 #count the number of times each feature is in the top 5
31 true_importance = np.arange(5)
32 count = len(set(true_importance).intersection(set(indices)))
33 print(f"Number of true informative features in top 5: {count}")
34
35 #plot the top 5 features
36 plt.bar(range(len(importances)), importances[np.argsort(importances)[::-1]])
37 plt.xlabel('Feature Rank')
38 plt.ylabel('Feature Importance Score')
39 plt.title('Feature Importance Ranking')
40 plt.show()

```

The result of above code is like followings:



And the number of true informative feature in top 5 is:

Number of true informative features in top 5: 2

Q1(b) The first question: Gini Importance calculate the importance of feature by summing the impurity decrease it causes across all trees in the forest. If we assume that at the split points, each point impurity decrease ΔI , and if we at the point t then the proportion of the number of samples corresponding to this node to the total number of samples is $\frac{N_t}{N}$. Thus, for all point

the reduction of total impurities is $\sum_{\text{every point } j \text{ in this feature}} \Delta I \frac{N_j}{N}$, where the N is the number of all sample, the N_t is the number of samples corresponding to the point. When we split the point we just want to have less impurity in the next layer, so if the feature is not used to split, which means it has no attribution to decreasing the impurity, and its $\Delta I = 0$, and its importance is also 0.

The second question: The feature importance of 0.15 means this feature accounts for 15% of the total importance in all splits.

Q1(c) Set the random_state=I, then the completely code is as follows:

```
1. counts=[]
2. for i in range(1000):
3.     X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=i)
4.
5.     #Normalize the data
6.     scaler = StandardScaler()
7.     scaler.fit(X)
8.     X = scaler.transform(X)
9.
10.    #shuffle the data with seed=i
11.    shuffle_idx = np.random.default_rng(seed=i).permutation(X.shape[1])
12.    X_shff = X[:,shuffle_idx]
13.
14.    #Decision Tree Classifier
15.    clf = DecisionTreeClassifier(criterion='entropy', random_state=4)
16.    clf.fit(X_shff, y)
17.    importances = clf.feature_importances_
18.    #descending order
19.    indices = np.argsort(importances)[::-1][:5] #top 5 features
20.    #map the top 5 features to the original data
21.
22.    top5 = shuffle_idx[indices]
```

```

23.     #count the number of times each feature is in the top 5
24.     count=np.sum(top5<5)
25.     counts.append(count)
26.
27.     #plot the top 5 features
28.     plt.hist(counts,bins=np.arange(0,6)-0.5,edgecolor = 'black')
29.     plt.xticks(range(0,6))
30.     plt.xlabel('Number of True Features Recovered')
31.     plt.ylabel('Frequency')
32.     plt.title('Histogram of True Features Recovered (Decision Tree)')
33.     plt.show()
34.     average=np.mean(counts)
35.     print(f"Average number of good feature recovered:{average}")

```

```

40. #Q1(c)
41. counts=[]
42. for i in range(1000):
43.     X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=i)
44.
45.     #Normalize the data
46.     scaler = StandardScaler()
47.     scaler.fit(X)
48.     X = scaler.transform(X)
49.
50.     #shuffle the data with seed=i
51.     shuffle_idx = np.random.default_rng(seed=i).permutation(X.shape[1])
52.     X_shff = X[:,shuffle_idx]
53.
54.     #Decision Tree Classifier
55.     clf = DecisionTreeClassifier(criterion='entropy', random_state=4)
56.     clf.fit(X_shff, y)
57.     importances = clf.feature_importances_
58.     #descending order
59.     indices = np.argsort(importances)[::-1][:5] #top 5 features
60.     #map the top 5 features to the original data
61.
62.     top5 = shuffle_idx[indices]
63.     #count the number of times each feature is in the top 5
64.     count=np.sum(top5<5)
65.     counts.append(count)

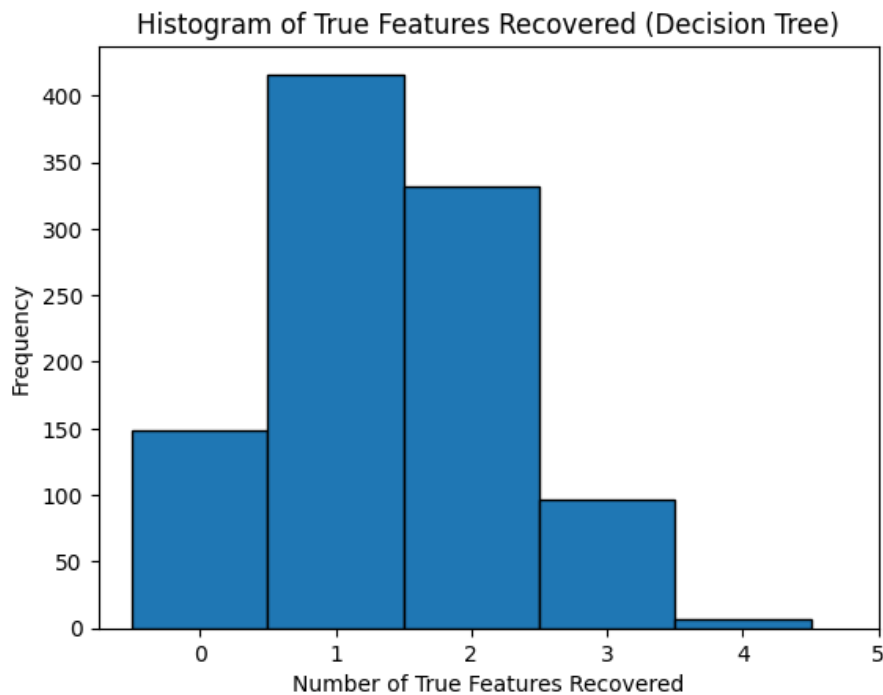
```

```

67.     #plot
68.     plt.hist(counts,bins=np.arange(0,6)-0.5,edgecolor = 'black')
69.     plt.xticks(range(0,6))
70.     plt.xlabel('Number of True Features Recovered')
71.     plt.ylabel('Frequency')
72.     plt.title('Histogram of True Features Recovered (Decision Tree)')
73.     plt.show()
74.     average=np.mean(counts)
75.     print(f"Average number of good feature recovered:{average}")

```

And the result is as follows:



The average is: Average number of good feature recovered:1.396

According to sklearn's documentation, when `shuffle = False`, the order of features is in the following order: `n_informative`, `n_redundant`, `n_repeated`, `random_state`. Therefore, regardless of the `random_state`, as long as `shuffle = False`, the first 5 features are informative, followed by redundancy. Therefore, in this part, the first 5 features in the data generated by each trial i are always informative, followed by redundancy. Then when shuffling the feature order, we use a random seed of i . Therefore, in each trial i , the first 5 features of the generated data are informative, and the order of the scrambled features is random, determined by `seed = i`. Therefore, in statistics, the number of correct identifications depends on whether the model can identify those of the original top 5 among the scrambled features.

Q1(d) This part use logistic regression without penalty, the code is as following:

```

1. #Q1(d)
2. #without scaling
3. counts_nscale = []
4. #repeat Q1c
5. for i in range(1000):
6. X,y =
make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=
False,random_state=i)
7.
8.     #shuffle the data with seed=i
9.     shuffle_idx = np.random.default_rng(seed=i).permutation(X.shape[1])
10.     X_shff = X[:,shuffle_idx]
11.

```

```

12.     #Logistic Regression
13.     lr = LogisticRegression(penalty=None,max_iter=1000,random_state=4)
14.     lr.fit(X_shff,y)
15.     coef = np.abs(lr.coef_[0])
16.     #descending order
17.     indices = np.argsort(coef)[::-1][:5] #top 5 features
18.     #map the top 5 features to the original data
19.     top5 = shuffle_idx[indices]
20.     #count the number of times each feature is in the top 5
21.     count=np.sum(top5<5)
22.     counts_nscale.append(count)
23.
24. #with scale
25. counts_scale = []
26. for i in range(1000):
27.     X,y =
make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=
False,random_state=i)
28.
29.     #Normalisation
30.     scaler = StandardScaler()
31.     X = scaler.fit_transform(X)
32.
33.     #shuffle the data with seed=i
34.     shuffle_idx = np.random.default_rng(seed=i).permutation(X.shape[1])
35. X_shff = X[:,shuffle_idx]
36.
37.     #Logistic Regression
38.     lr = LogisticRegression(penalty=None,max_iter=1000,random_state=4)
39.     lr.fit(X_shff,y)
40.     coef = np.abs(lr.coef_[0])
41.     #descending order
42.     indices = np.argsort(coef)[::-1][:5] #top 5 features
43.     #map the top 5 features to the original data
44.     top5 = shuffle_idx[indices]
45.     #count the number of times each feature is in the top 5
46.     count=np.sum(top5<5)
47.     counts_scale.append(count)
48.
49. #plot with scale
50. plt.hist(counts_nscale,bins=np.arange(0,6)-0.5, alpha=0.5, label='Unscaled')
51. plt.hist(counts_scale,bins=np.arange(0,6)-0.5, alpha=0.5, label='Scaled')
52. plt.legend()
53. plt.xlabel('Number of True Features Recovered')

```

```

54. plt.ylabel('Frequency')
55. plt.title('Histogram of True Features Recovered (Logistic Regression)')
56. plt.show()
57. average_nscale=np.mean(counts_nscale)
58. average_scale=np.mean(counts_scale)
59. print(f"Average number of good feature recovered(LR without
scale):{average_nscale}")
60. print(f"Average number of good feature recovered(LR with scale):{average_scale}")

```

```

81 #Q1(d)
82 #without scaling
83 counts_nscale = []
84 #repeat Q1c
85 for i in range(1000):
86     X,y = make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False,random_state=i)
87
88     #shuffle the data with seed=i
89     shuffle_idx = np.random.default_rng(seed=i).permutation(X.shape[1])
90     X_shff = X[:,shuffle_idx]
91
92     #Logistic Regression
93     lr = LogisticRegression(penalty=None,max_iter=1000,random_state=4)
94     lr.fit(X_shff,y)
95     coef = np.abs(lr.coef_[0])
96     #descending order
97     indices = np.argsort(coef)[::-1][5:] #top 5 features
98     #map the top 5 features to the original data
99     top5 = shuffle_idx[indices]
100     #count the number of times each feature is in the top 5
101     count=np.sum(top5<5)
102     counts_nscale.append(count)
103
104 #with scale
105 counts_scale = []
106 for i in range(1000):
107     X,y = make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False,random_state=i)

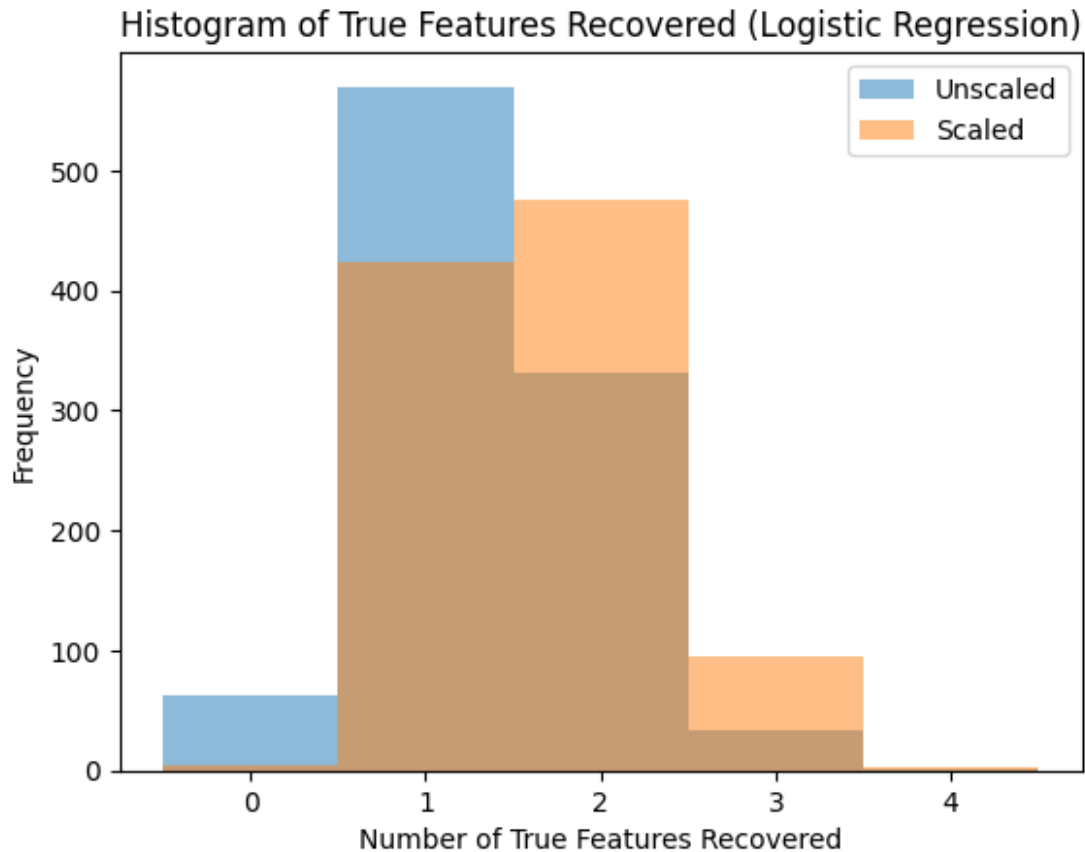
```

```

110     scaler = StandardScaler()#Normalisation
111     X = scaler.fit_transform(X)
112     #shuffle the data with seed=i
113     shuffle_idx = np.random.default_rng(seed=i).permutation(X.shape[1])
114     X_shff = X[:,shuffle_idx]
115     #Logistic Regression
116     lr = LogisticRegression(penalty=None,max_iter=1000,random_state=4)
117     lr.fit(X_shff,y)
118     coef = np.abs(lr.coef_[0])
119     #descending order
120     indices = np.argsort(coef)[::-1][5:] #top 5 features
121     #map the top 5 features to the original data
122     top5 = shuffle_idx[indices]
123     #count the number of times each feature is in the top 5
124     count=np.sum(top5<5)
125     counts_scale.append(count)
126
127 #plot with scale
128 plt.hist(counts_nscale,bins=np.arange(0,6)-0.5, alpha=0.5, label='Unscaled')
129 plt.hist(counts_scale,bins=np.arange(0,6)-0.5, alpha=0.5, label='Scaled')
130 plt.legend()
131 plt.xlabel('Number of True Features Recovered')
132 plt.ylabel('Frequency')
133 plt.title('Histogram of True Features Recovered (Logistic Regression)')
134 plt.show()
135 average_nscale=np.mean(counts_nscale)
136 average_scale=np.mean(counts_scale)
137 print(f"Average number of good feature recovered(LR without scale):{average_nscale}")
138 print(f"Average number of good feature recovered(LR with scale):{average_scale}")

```

The histogram is like followings:



The averages are as following respectively:

Average number of good feature recovered(LR without scale):1.34
Average number of good feature recovered(LR with scale):1.667

In logistic regression without scaling, the size of the coefficients may be affected by the feature scale, so the feature importance may not be accurate. However, after scaling, the features are in the same dimension, and the coefficient size can more accurately reflect the importance. Therefore, logistic regression should perform better after scaling.

Q1(e) No, scaling does not affect decision trees since they split based on feature value order, not magnitude.

Q1(f) Using the followings code to complete the requirement:

```

1. #Q1(f)
2. overlaps=[]
3. for i in range(1000):
4.     X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=1
5,shuffle=False, random_state=i)
6.     #Normalize the data
7.     scaler = StandardScaler()
8.     X = scaler.fit_transform(X)
9.     #shuffle the data with seed=i
10.    shuffle_idx = np.random.default_rng(seed=i).permutation(X.shape[1])
11.    X_shff = X[:,shuffle_idx]
12.    #Desicion Tree Classifier

```

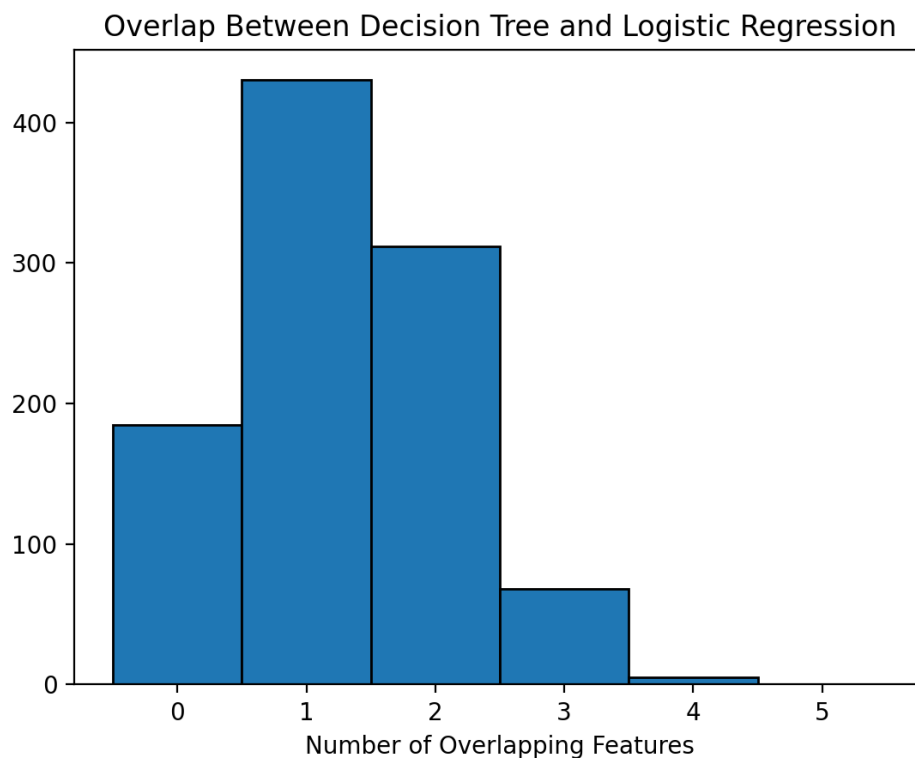


```

12.   dtc = DecisionTreeClassifier(criterion='entropy', random_state=4)
13.   dtc.fit(X_shff, y)
14.   importances = clf.feature_importances_
15.   #descending order
16.   indices = np.argsort(importances)[::-1][:5] #top 5 features
17.   #map the top 5 features to the original data
18.   top5_tree = shuffle_idx[indices]
19.   #Logistics Regression
20.   lr = LogisticRegression(penalty=None, random_state=4)
21.   lr.fit(X_shff,y)
22.   coef = np.abs(lr.coef_[0])
23.   #descending order
24.   indices = np.argsort(coef)[::-1][:5] #top 5 features
25.   top5_lr = shuffle_idx[indices]
26.   overlaps.append(len(set(top5_tree)&set(top5_lr)))
27. #plot
28. plt.hist(overlaps,bins=np.arange(7)-0.5,edgecolor='black')
29. plt.xlabel('Number of Overlapping Features')
30. plt.title('Overlap Between Decision Tree and Logistic Regression')
31. plt.show()

```

The result shows as following:



Q1(g) Disadvantages of Model-Based Feature Importance:

1. Model bias: Different models emphasize different features, like decision tree and logistics regression.

2. Redundant feature: For the different redundancies will have different results. If the redundancy is too high, which will dilute importance scores and make true features harder to identify.
3. Just like the example, if the model prefer some feature types, the model-based will miss true associations.

Q2(a) Because 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, which is only the local optimal choice and not the global optimal choice. The pitfalls of greedy algorithms are that we will miss the global optimal choice, remove one feature might degrade the utility of others and computational cost a lot.

Q2(b) The code implementing the backward elimination algorithm as followings:

```

1. #Q2(b)
2. #backward elimination algorithm
3. from sklearn.metrics import accuracy_score
4. def backward_elimination(n_feature_keep=5, seed=0):
5.     X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=1
6.     5,shuffle=False, random_state=seed)
7.     #Normalize the data
8.     X = scaler.fit_transform(X)
9.     #shuffle the data with seed
10.    shuffle_idx = np.random.default_rng(seed=seed).permutation(X.shape[1])
11.    X_shff = X[:,shuffle_idx]
12.    features = list(range(X_shff.shape[1]))
13.    while len(features) > n_feature_keep:
14.        scores=[]
15.        for f in features:
16.            X_subset = X[:, [x for x in features if x!=f]]
17.            lr = LogisticRegression(penalty=None, random_state=4)
18.            lr.fit(X_subset,y)
19.            scores.append(accuracy_score(y,lr.predict(X_subset)))
20.        smallest_feature = features[np.argmin(scores)]#find the feature corresponding
21.        to th smallest drop in the metric
22.        features.remove(smallest_feature)
23.        origin_left = shuffle_idx[features]
24.        correct = np.sum(origin_left<5)
25.        print(f"Left features:{origin_left}")
26.    return correct
27. correct = backward_elimination(n_feature_keep=5, seed=0)
28. print(f" Number of correct features: {correct}")

```

```

170 #Q2(b)
171 #backward elimination algorithm
172 from sklearn.metrics import accuracy_score
173 def backward_elimination(n_feature_keep=5,seed=0):
174     X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=seed)
175     #Normalize the data
176     X = scaler.fit_transform(X)
177     #shuffle the data with seed
178     shuffle_idx = np.random.default_rng(seed=seed).permutation(X.shape[1])
179     X_shff = X[:,shuffle_idx]
180     features = list(range(X_shff.shape[1]))
181     while len(features) > n_feature_keep:
182         scores=[]
183         for f in features:
184             X_subset = X[:,[x for x in features if x!=f]]
185             lr = LogisticRegression(penalty=None, random_state=4)
186             lr.fit(X_subset,y)
187             scores.append(accuracy_score(y,lr.predict(X_subset)))
188         smallest_feature = features[np.argmin(scores)]#find the feature corresponding to th smallest drop in the metric
189         features.remove(smallest_feature)
190     origin_left = shuffle_idx[features]
191     correct = np.sum(origin_left<5)
192     print(f"Left features:{origin_left}")
193     return correct
194 correct = backward_elimination(n_feature_keep=5,seed=0)
195 print(f" Number of correct features: {correct}")

```

Left features:[1 16 12 3 11]
Number of correct features: 2

The result is:

Q2(c) Repeat part a for 1000:

```

1. #Q2(c)
2. recovered_f=[]
3. for i in range(1000):
4.     correct = backward_elimination(n_feature_keep=5,seed=i)
5.     recovered_f.append(correct)
6. plt.hist(recovered_f, bins=np.arange(0,6)-0.5, edgecolor='black')
7. plt.xlabel('Number of True Features Recovered(Backward Elimination)')
8. plt.ylabel('Frequency')
9. plt.title('Backward Elimination: True Features Recovered (Logistic Regression)')
10. plt.show()
11. average_re=np.mean(recovered_f)
12. print(f"Average number of recovered feature (Backward Elimination):{average_re}")

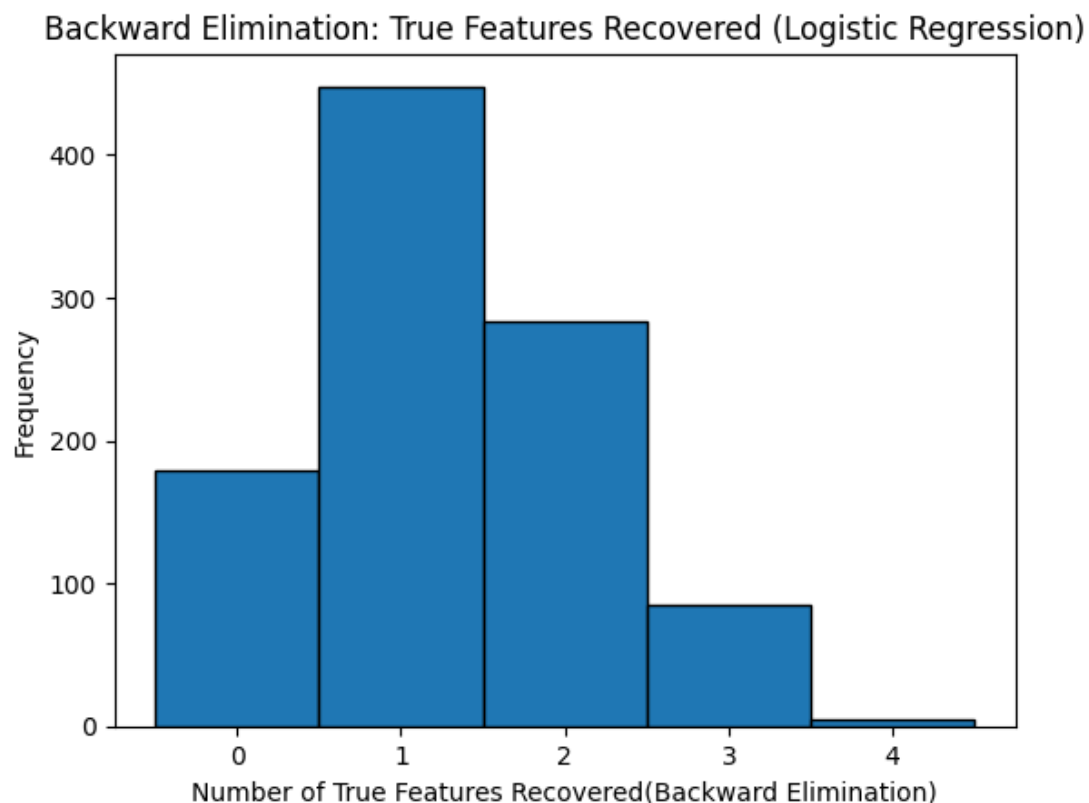
```

```

197 #Q2(c)
198 recovered_f=[]
199 for i in range(1000):
200     correct = backward_elimination(n_feature_keep=5,seed=i)
201     recovered_f.append(correct)
202 plt.hist(recovered_f, bins=np.arange(0,6)-0.5, edgecolor='black')
203 plt.xlabel('Number of True Features Recovered(Backward Elimination)')
204 plt.ylabel('Frequency')
205 plt.title('Backward Elimination: True Features Recovered (Logistic Regression)')
206 plt.show()
207 average_re=np.mean(recovered_f)
208 print(f"Average number of recovered feature (Backward Elimination):{average_re}")

```

The histogram chart is like followings:



The average number of recovered feature is:

Average number of recovered feature (Backward Elimination):1.289

Q2(d) Best subset selection could avoid to select the local optimal choice, so it will perform well on selecting global optimal choice. But this will cost a lot time more than Backward elimination, since we need find all possible subsets and train a model on each subset. Therefore, best subset selection will not always outperform backward elimination. The disadvantages of best subset selection are that for large data it can't work and prone to overfitting with limited data.

Q2(e) Implement Best subset selection:

```
1. #Q2(e)
2. from itertools import combinations
3.
4. def best_subset(seed=0):
5.     X,y=make_classification(n_samples=1000,n_features=7,n_informative=3,n_redundant=4,
6.                             shuffle=False, random_state=seed)
7.     #Normalize the data
8.     X = scaler.fit_transform(X)
9.     #shuffle the data with seed
```

```

9.     shuffle_idx = np.random.default_rng(seed=seed).permutation(X.shape[1])
10.    X_shff = X[:,shuffle_idx]
11.    best_score = -np.inf
12.    best_subset = None
13.
14.    for subset in combinations(range(X_shff.shape[1]),3):
15.        X_subset = X_shff[:,subset]
16.        lr = LogisticRegression(penalty=None,random_state=4)
17.        lr.fit(X_subset,y)
18.        acc = accuracy_score(y,lr.predict(X_subset))
19.        if acc > best_score:
20.            best_score=acc
21.            best_subset=subset
22.    origin_subset = [shuffle_idx[idx] for idx in best_subset]
23.    correct_best = sum(1 for idx in origin_subset if idx < 3)
24.    return correct_best
25. #repeat 1000
26. recoveries=[]
27. for i in range(1000):
28.     correct_best = best_subset(seed=i)
29.     recoveries.append(correct_best)
30. plt.hist(recoveries,bins=np.arange(0,6)-0.5, edgecolor='black')
31. plt.xlabel('Number of Correct Features')
32. plt.title('Best Subset Selection')
33. plt.show()
34. print(f"Average correct(Best Selection):{np.mean(recoveries)}")

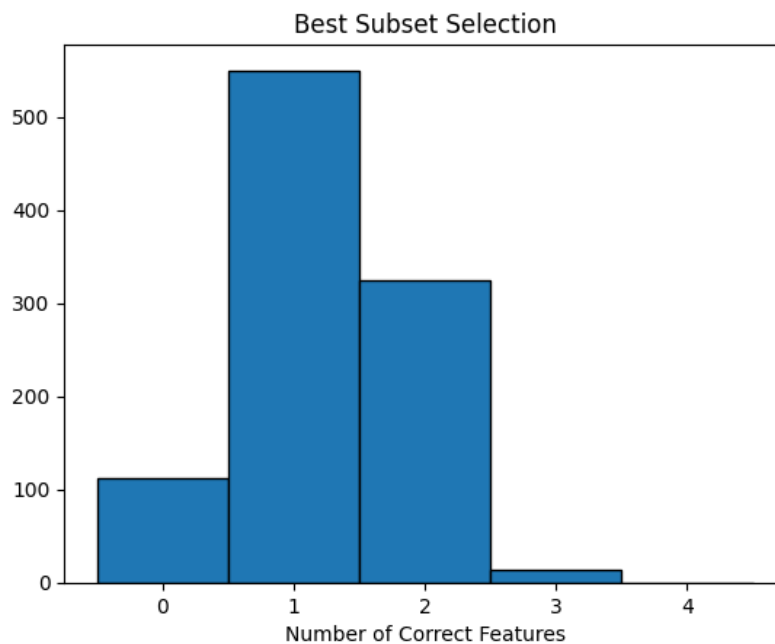
```

```

210 #Q2(e)
211 from itertools import combinations
212
213 def best_subset(seed=0):
214     X,y=make_classification(n_samples=1000,n_features=7,n_informative=3,n_redundant=4,shuffle=False, random_state=seed)
215     #Normalize the data
216     X = scaler.fit_transform(X)
217     #shuffle the data with seed
218     shuffle_idx = np.random.default_rng(seed=seed).permutation(X.shape[1])
219     X_shff = X[:,shuffle_idx]
220     best_score = -np.inf
221     best_subset = None
222
223     for subset in combinations(range(X_shff.shape[1]),3):
224         X_subset = X_shff[:,subset]
225         lr = LogisticRegression(penalty=None,random_state=4)
226         lr.fit(X_subset,y)
227         acc = accuracy_score(y,lr.predict(X_subset))
228         if acc > best_score:
229             best_score=acc
230             best_subset=subset
231     origin_subset = [shuffle_idx[idx] for idx in best_subset]
232     correct_best = sum(1 for idx in origin_subset if idx < 3)
233     return correct_best
234 #repeat 1000
235 recoveries=[]
236 for i in range(1000):
237     correct_best = best_subset(seed=i)
238     recoveries.append(correct_best)
239 plt.hist(recoveries,bins=np.arange(0,6)-0.5, edgecolor='black')
240 plt.xlabel('Number of Correct Features')
241 plt.title('Best Subset Selection')
242 plt.show()
243 print(f"Average correct(Best Selection):{np.mean(recoveries)}")

```

The histogram chart is below:



Average correct:1.24

The average is:

Q2(f) Permutation Feature Importance: it will randomly shuffle the feature's value and measure the drop in model performance. A bigger drop means higher importance. And it does not rely on any model internals(such as coefficients). It's more fair to compare logistic regression and decision trees.

The implement code:

```
1. #Q2(f)
2. #Permutation Feature Importance score
3. from sklearn.inspection import permutation_importance
4. def permutation_importance(seed=0):
5.     X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=1
6.     #Normalize the data
7.     X = scaler.fit_transform(X)
8.     #shuffle the data with seed
9.     shuffle_idx = np.random.default_rng(seed=seed).permutation(X.shape[1])
10.    X_shff = X[:,shuffle_idx]
11.    lr = LogisticRegression(penalty=None, random_state=4)
12.    lr.fit(X_shff,y)
13.    #calculate permutaion importance
```

```

14.     importance = permutation_importance(lr, X_shff,y,n_repeats=10,random_state=4)
15.     mean_importance = importance.importances_mean
16.     #top5
17.     indices = np.argsort(mean_importance)[::-1][:5]
18.     top5 = shuffle_idxes[indices]
19.     correct_pe = np.sum(top5<5)
20.     return correct_pe
21. count_pe=[]
22. for i in range(1000):
23.     correct_pe = permutation_importance(seed=i)
24.     count_pe.append(correct_pe)
25. #plot
26. plt.hist(count_pe,bins=np.arange(0,6)-0.5,edgecolor='black')
27. plt.xlabel('Number of True Feature')
28. plt.title('Permutation Importance')
29. plt.show()
30. print(f"Average correct:{np.mean(count_pe)}")

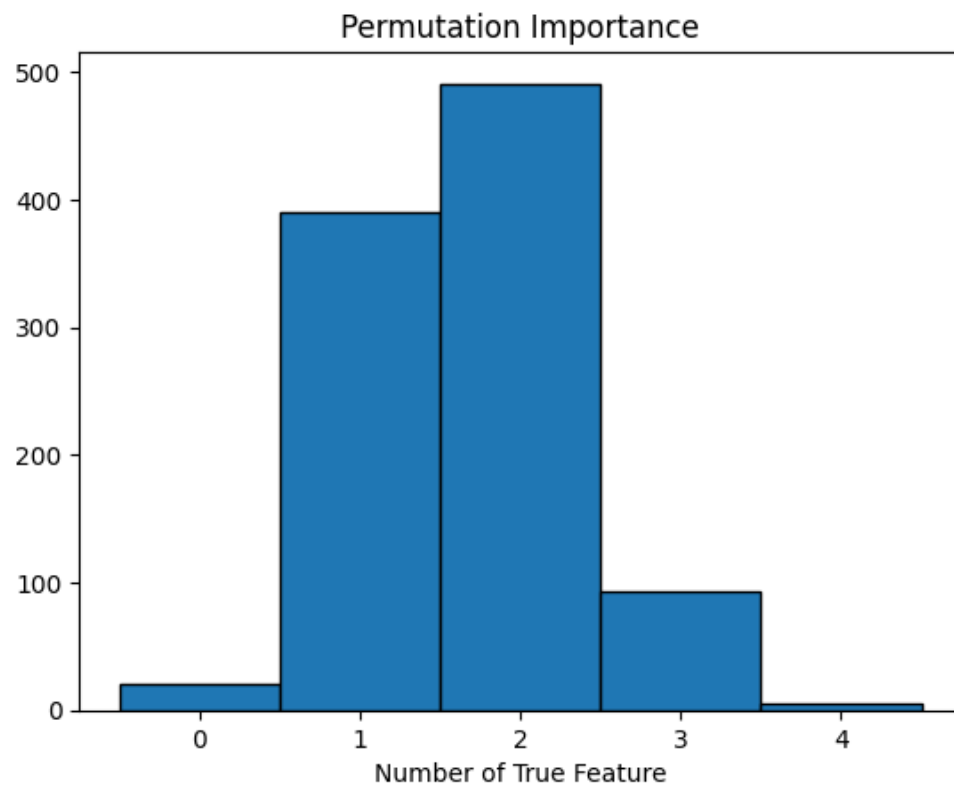
```

```

245 #Q2(f)
246 #Permutation Feature Importance score
247 from sklearn.inspection import permutation_importance
248 def permutation_importance(seed=0):
249     X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=seed)
250     #Normalize the data
251     X = scaler.fit_transform(X)
252     #shuffle the data with seed
253     shuffle_idxes = np.random.default_rng(seed=seed).permutation(X.shape[1])
254     X_shff = X[:,shuffle_idxes]
255     lr = LogisticRegression(penalty=None, random_state=4)
256     lr.fit(X_shff,y)
257     #calculate permutaion importance
258     importance = permutation_importance(lr, X_shff,y,n_repeats=10,random_state=4)
259     mean_importance = importance.importances_mean
260     #top5
261     indices = np.argsort(mean_importance)[::-1][:5]
262     top5 = shuffle_idxes[indices]
263     correct_pe = np.sum(top5<5)
264     return correct_pe
265 count_pe=[]
266 for i in range(1000):
267     correct_pe = permutation_importance(seed=i)
268     count_pe.append(correct_pe)
269 #plot
270 plt.hist(count_pe,bins=np.arange(0,6)-0.5,edgecolor='black')
271 plt.xlabel('Number of True Feature')
272 plt.title('Permutation Importance')
273 plt.show()
274 print(f"Average correct:{np.mean(count_pe)}")

```

The histogram chart of permutation importance as below:



The average is:

Average correct:1.676