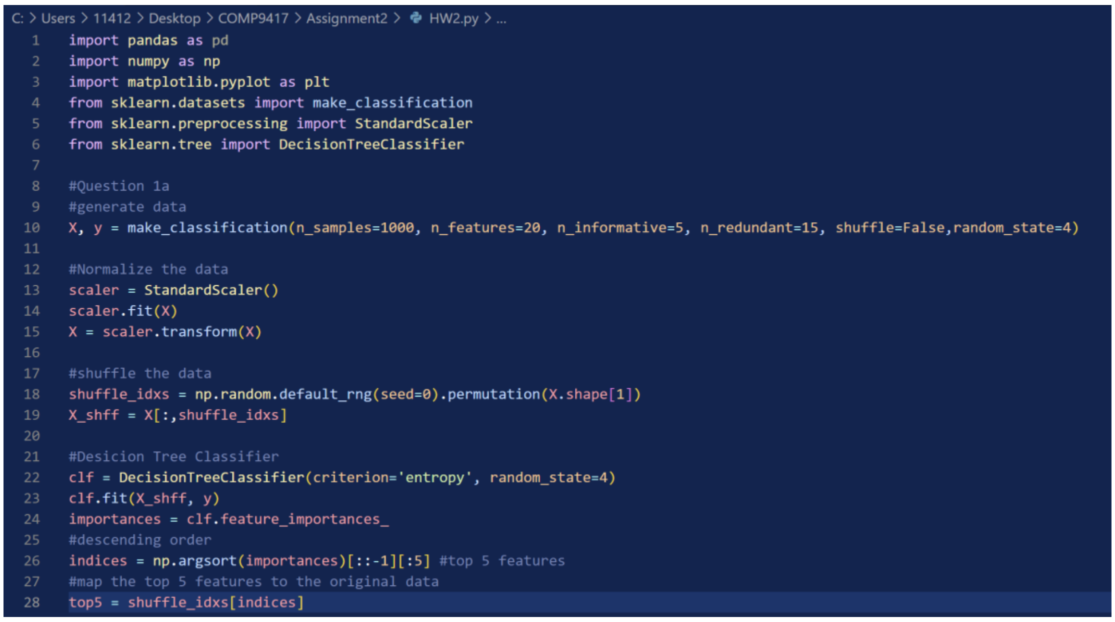
Assignment2

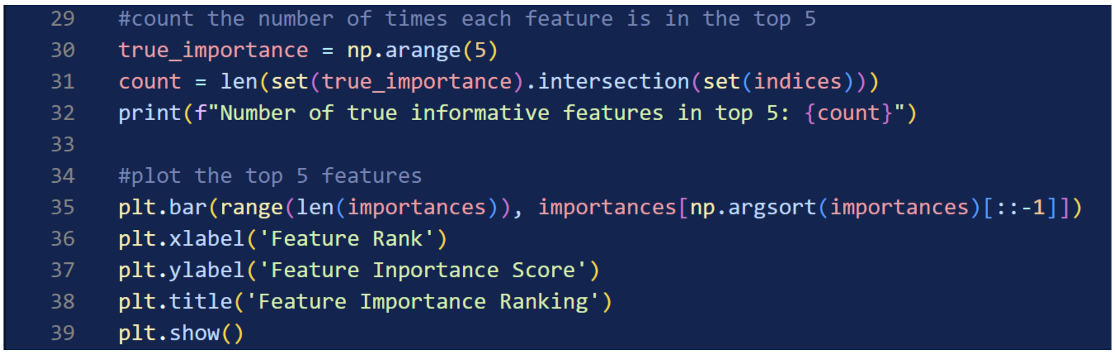
Q1(a) There is the code to complete the request:

```python
import pandas as pd
import numpy as np
import random
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier

#Question 1a
#generate data
X, y = make_classification(n_samples=1000, n_features=20, n_informative=5,
n_redundant=15, shuffle=False, random_state=4)

#Normalize the data
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)

#shuffle the data
shuffle_idxs = np.random.default_rng(seed=0).permutation(X.shape[1])
X_shff = X[:,shuffle_idxs]

#Desicion Tree Classifier
clf = DecisionTreeClassifier(criterion='entropy', random_state=4)
clf.fit(X_shff, y)
importances = clf.feature_importances_
#descending order
indices = np.argsort(importances)[::-1][:5] #top 5 features
#map the top 5 features to the original data
top5 = shuffle_idxs[indices]
#count the number of times each feature is in the top 5
count=np.sum(top5<5)
print(f"Number of true informative features in top 5: {count}")

#plot the top 5 features
plt.bar(range(len(importances)), importances[np.argsort(importances)[::-1]])
plt.xlabel('Feature Rank')
plt.ylabel('Feature Inportance Score')
plt.title('Feature Importance Ranking')
plt.show()
```



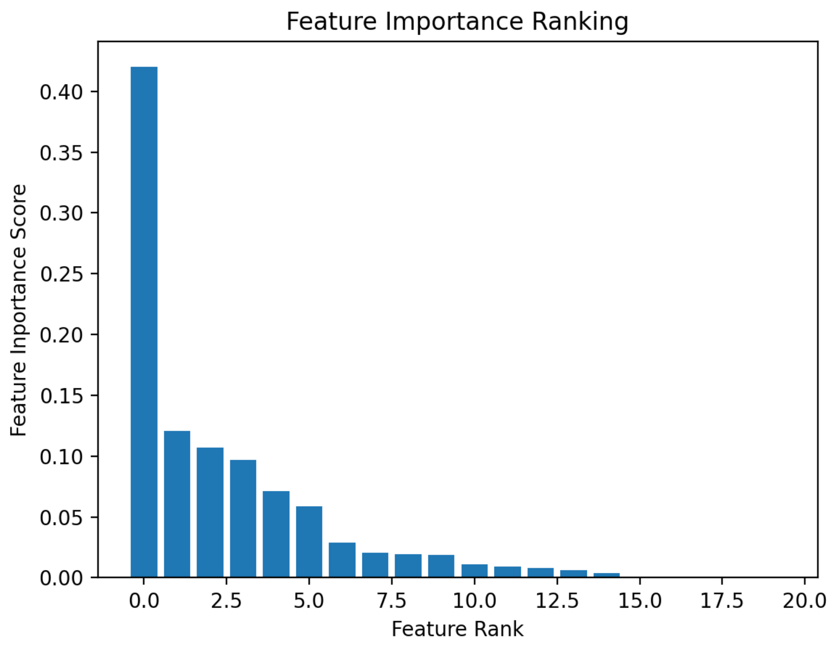
C: > Users > 11412 > Desktop > COMP9417 > Assignment2 > HW2.py > ...

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier

#Question 1a
#generate data
X, y = make_classification(n_samples=1000, n_features=20, n_informative=5, n_redundant=15, shuffle=False,random_state=4)

#Normalize the data
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)

#shuffle the data
shuffle_idxs = np.random.default_rng(seed=0).permutation(X.shape[1])
X_shff = X[:,shuffle_idxs]

#Desicion Tree Classifier
clf = DecisionTreeClassifier(criterion='entropy', random_state=4)
clf.fit(X_shff, y)
importances = clf.feature_importances_
#descending order
indices = np.argsort(importances)[::-1][:5] #top 5 features
#map the top 5 features to the original data
top5 = shuffle_idxs[indices]
```

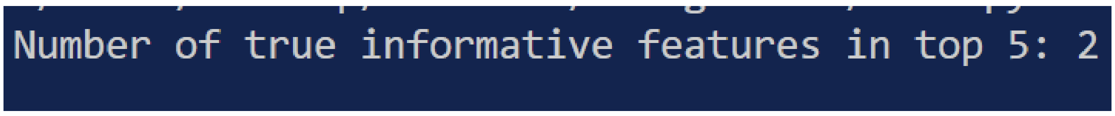


```python
#count the number of times each feature is in the top 5
true_importance = np.arange(5)
count = len(set(true_importance).intersection(set(indices)))
print(f"Number of true informative features in top 5: {count}")

#plot the top 5 features
plt.bar(range(len(importances)), importances[np.argsort(importances)[::-1]])
plt.xlabel('Feature Rank')
plt.ylabel('Feature Inportance Score')
plt.title('Feature Importance Ranking')
plt.show()
```

The result of above code is like followings:



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



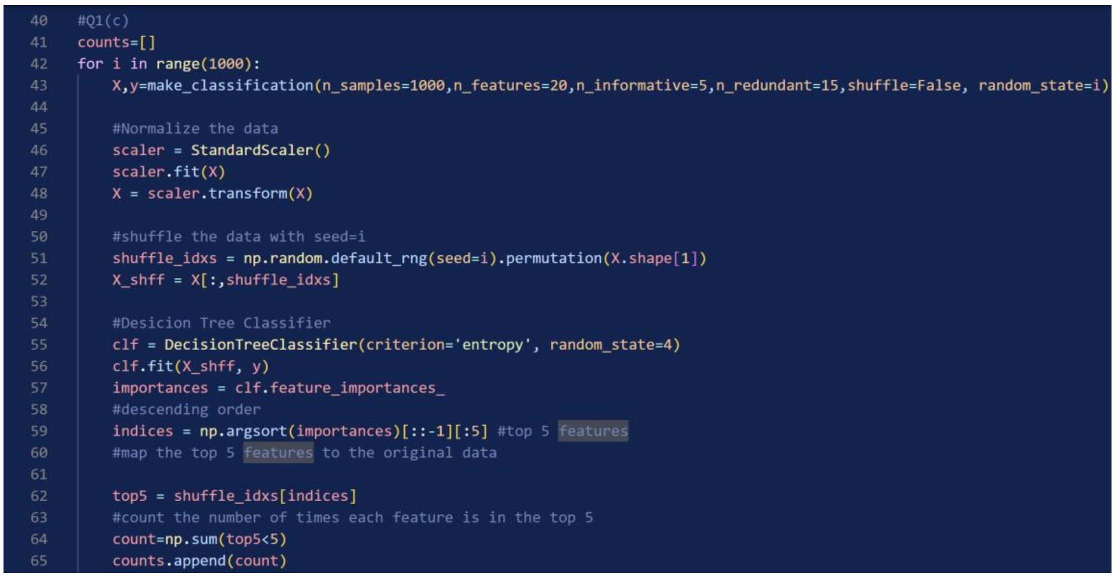
```
Number of true informative features in top 5: 2
```

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 $\Delta \boldsymbol{I}$, and if we at the point $\boldsymbol{t}$ then the proportion of the number of samples corresponding to this node to the total number of samples is $\frac{\boldsymbol{N_t}}{\boldsymbol{N}}$. Thus, for all point the reduction of total impurities is $\sum_{\boldsymbol{every\ point\ j\ in\ this\ feature}} \Delta \boldsymbol{I} \frac{\boldsymbol{N_j}}{\boldsymbol{N}}$, where the $\boldsymbol{N}$ is the number of all sample, the $\boldsymbol{N_t}$ 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 $\Delta \boldsymbol{I} = \boldsymbol{0}$, and its importance is also 0.

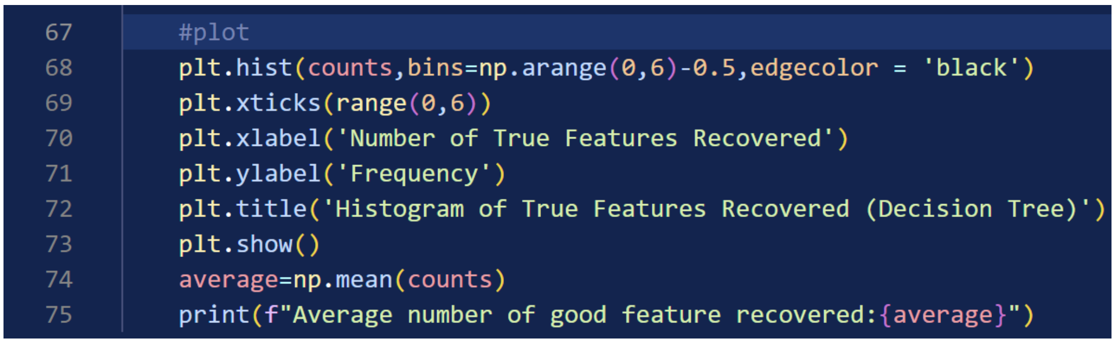
The second question: The feature importance of 0.15 means this feature accounts for 15% of the total importance in all splits.

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

```python
counts=[]
for i in range(1000):

    X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=i)

    #Normalize the data
    scaler = StandardScaler()
    scaler.fit(X)
    X = scaler.transform(X)

    #shuffle the data with seed=i
    shuffle_idxs = np.random.default_rng(seed=i).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]

    #Desicion Tree Classifier
    clf = DecisionTreeClassifier(criterion='entropy', random_state=4)
    clf.fit(X_shff, y)
    importances = clf.feature_importances_
    #descending order
    indices = np.argsort(importances)[::-1][:5] #top 5 features
    #map the top 5 features to the original data

    top5 = shuffle_idxs[indices]
    #count the number of times each feature is in the top 5
    count=np.sum(top5<5)
    counts.append(count)

    #plot the top 5 features
    plt.hist(counts,bins=np.arange(0,6)-0.5,edgecolor = 'black')
    plt.xticks(range(0,6))
    plt.xlabel('Number of True Features Recovered')
    plt.ylabel('Frequency')
    plt.title('Histogram of True Features Recovered (Decision Tree)')
    plt.show()
    average=np.mean(counts)
    print(f"Average number of good feature recovered:{average}")
```

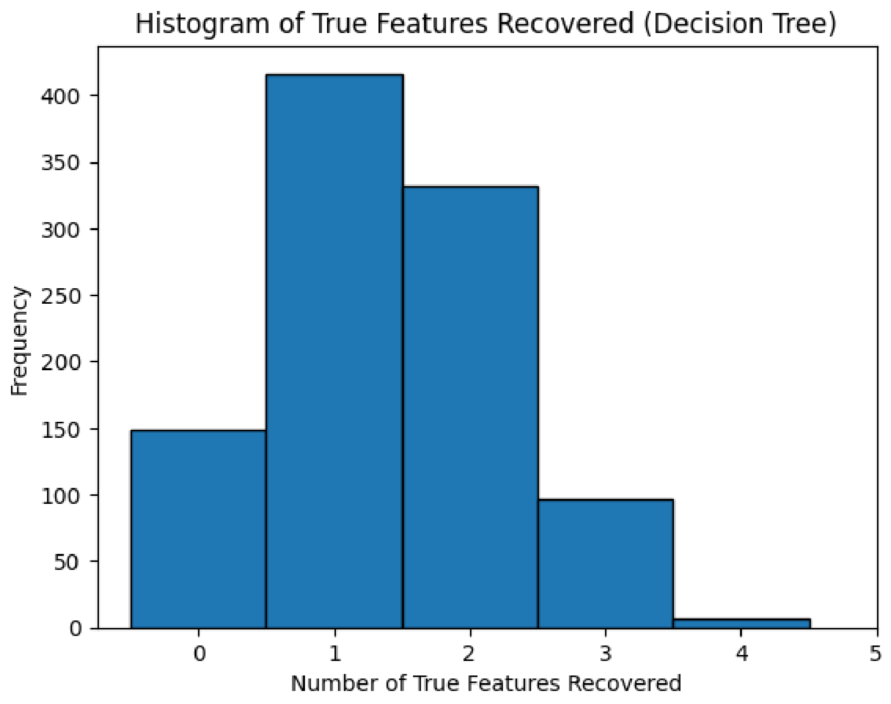


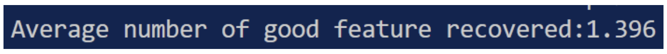
```
#Q1(c)
counts=[]
for i in range(1000):
    X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=i)

    #Normalize the data
    scaler = StandardScaler()
    scaler.fit(X)
    X = scaler.transform(X)

    #shuffle the data with seed=i
    shuffle_idxs = np.random.default_rng(seed=i).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]

    #Desicion Tree Classifier
    clf = DecisionTreeClassifier(criterion='entropy', random_state=4)
    clf.fit(X_shff, y)
    importances = clf.feature_importances_
    #descending order
    indices = np.argsort(importances)[::-1][:5] #top 5 features
    #map the top 5 features to the original data

    top5 = shuffle_idxs[indices]
    #count the number of times each feature is in the top 5
    count=np.sum(top5<5)
    counts.append(count)
```



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

And the result is as follows:



The average is: 

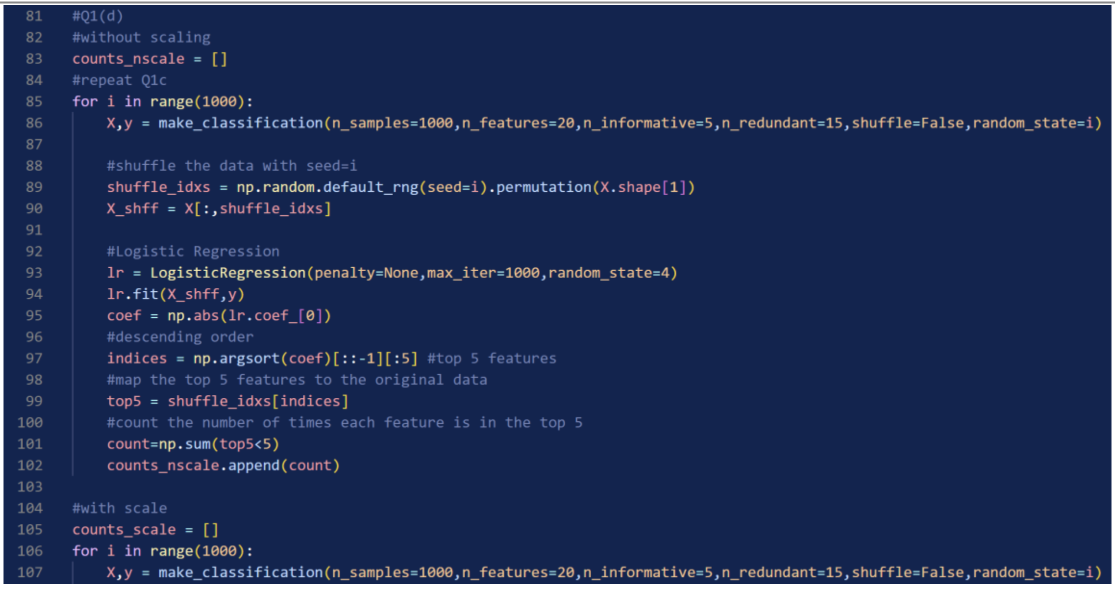
```
Average number of good feature recovered:1.396
```

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.

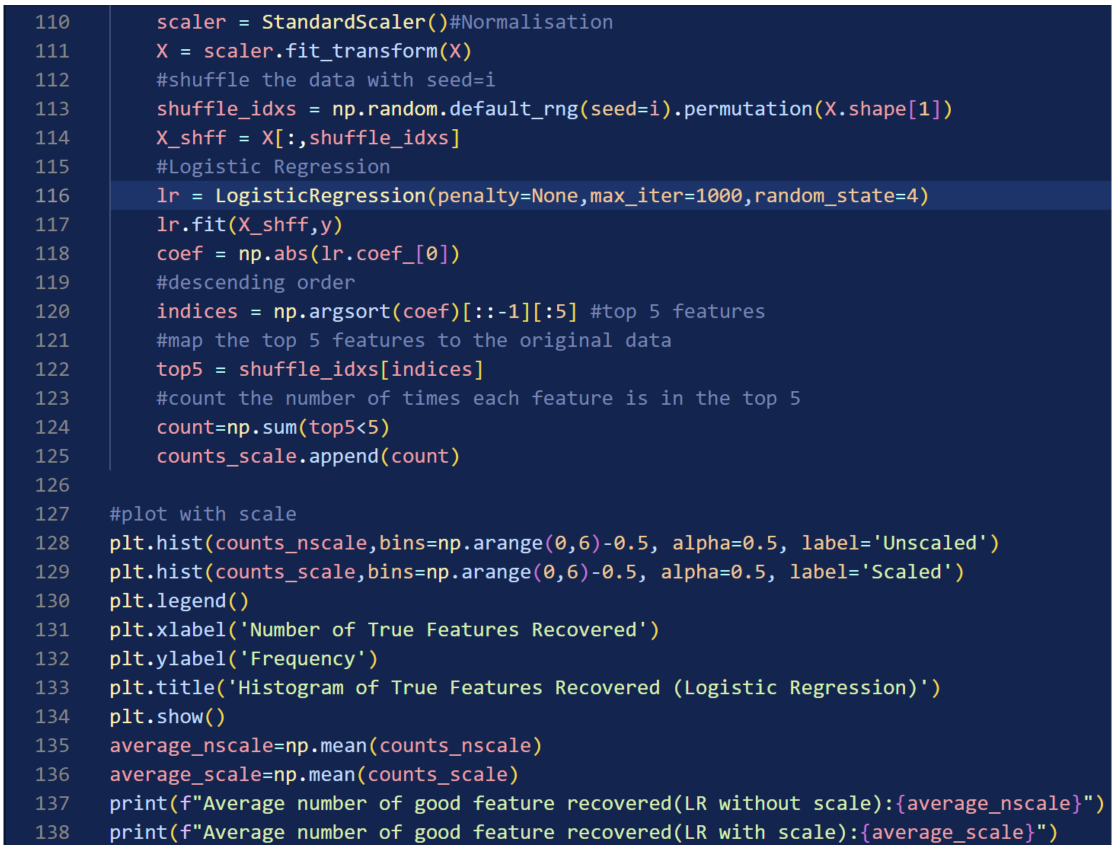
Q1(d) This part use logistic regression without penalty, the code is as following:

```
#Q1(d)
#without scaling
counts_nscale = []
#repeat Q1c
for i in range(1000):
X,y =
make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=
False,random_state=i)

    #shuffle the data with seed=i
    shuffle_idxs = np.random.default_rng(seed=i).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]

    #Logistic Regression
    lr = LogisticRegression(penalty=None,max_iter=1000,random_state=4)
    lr.fit(X_shff,y)
    coef = np.abs(lr.coef_[0])
    #descending order
    indices = np.argsort(coef)[::-1][:5] #top 5 features
    #map the top 5 features to the original data
    top5 = shuffle_idxs[indices]
    #count the number of times each feature is in the top 5
    count=np.sum(top5<5)
    counts_nscale.append(count)

#with scale
counts_scale = []
for i in range(1000):
    X,y = make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False,random_state=i)

    #Normalisation
    scaler = StandardScaler()
    X = scaler.fit_transform(X)

    #shuffle the data with seed=i
    shuffle_idxs = np.random.default_rng(seed=i).permutation(X.shape[1])
X_shff = X[:,shuffle_idxs]

    #Logistic Regression
    lr = LogisticRegression(penalty=None,max_iter=1000,random_state=4)
    lr.fit(X_shff,y)
    coef = np.abs(lr.coef_[0])
    #descending order
    indices = np.argsort(coef)[::-1][:5] #top 5 features
    #map the top 5 features to the original data
    top5 = shuffle_idxs[indices]
    #count the number of times each feature is in the top 5
    count=np.sum(top5<5)
    counts_scale.append(count)

#plot with scale
plt.hist(counts_nscale,bins=np.arange(0,6)-0.5, alpha=0.5, label='Unscaled')
plt.hist(counts_scale,bins=np.arange(0,6)-0.5, alpha=0.5, label='Scaled')
plt.legend()
plt.xlabel('Number of True Features Recovered')
```

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

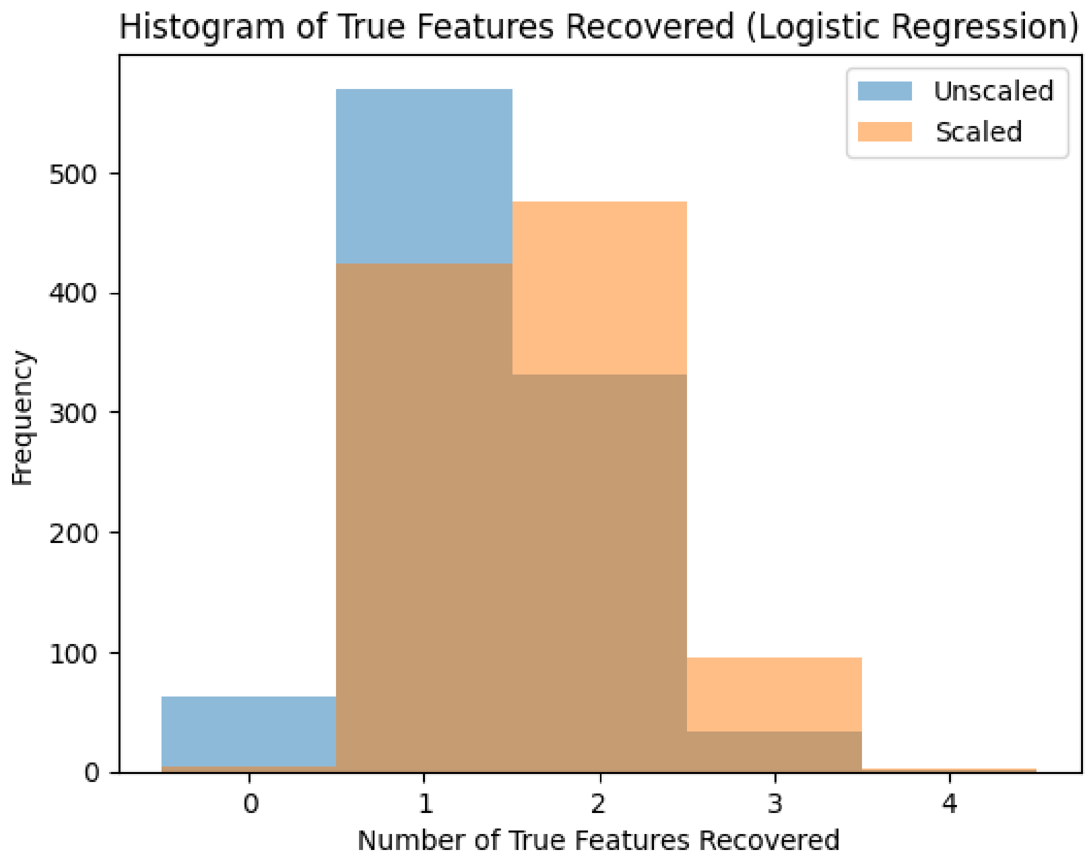


```python
#Q1(d)
#without scaling
counts_nscale = []
#repeat Q1c
for i in range(1000):
    X,y = make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False,random_state=i)

    #shuffle the data with seed=i
    shuffle_idxs = np.random.default_rng(seed=i).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]

    #Logistic Regression
    lr = LogisticRegression(penalty=None,max_iter=1000,random_state=4)
    lr.fit(X_shff,y)
    coef = np.abs(lr.coef_[0])
    #descending order
    indices = np.argsort(coef)[::-1][:5] #top 5 features
    #map the top 5 features to the original data
    top5 = shuffle_idxs[indices]
    #count the number of times each feature is in the top 5
    count=np.sum(top5<5)
    counts_nscale.append(count)

#with scale
counts_scale = []
for i in range(1000):
    X,y = make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False,random_state=i)
```



```python
    scaler = StandardScaler()#Normalisation
    X = scaler.fit_transform(X)
    #shuffle the data with seed=i
    shuffle_idxs = np.random.default_rng(seed=i).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]
    #Logistic Regression
    lr = LogisticRegression(penalty=None,max_iter=1000,random_state=4)
    lr.fit(X_shff,y)
    coef = np.abs(lr.coef_[0])
    #descending order
    indices = np.argsort(coef)[::-1][:5] #top 5 features
    #map the top 5 features to the original data
    top5 = shuffle_idxs[indices]
    #count the number of times each feature is in the top 5
    count=np.sum(top5<5)
    counts_scale.append(count)

#plot with scale
plt.hist(counts_nscale,bins=np.arange(0,6)-0.5, alpha=0.5, label='Unscaled')
plt.hist(counts_scale,bins=np.arange(0,6)-0.5, alpha=0.5, label='Scaled')
plt.legend()
plt.xlabel('Number of True Features Recovered')
plt.ylabel('Frequency')
plt.title('Histogram of True Features Recovered (Logistic Regression)')
plt.show()
average_nscale=np.mean(counts_nscale)
average_scale=np.mean(counts_scale)
print(f"Average number of good feature recovered(LR without scale):{average_nscale}")
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
```

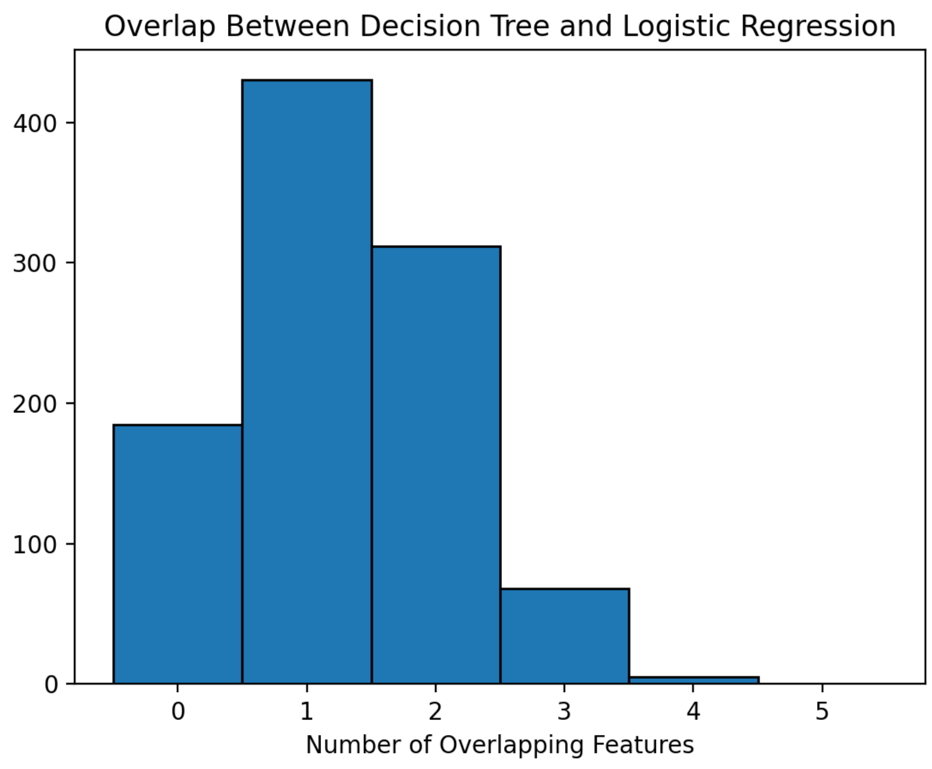
In logistic regression without scaling, the size of the coefficients may be affected by the feature scale, so the feature importance may not be accurate. However, after scaling, the features are in the same dimension, and the coefficient size can more accurately reflect the importance. Therefore, logistic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

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

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

```
#Q1(f)
overlaps=[]
for i in range(1000):
    X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=i)
    #Normalize the data
    scaler = StandardScaler()
    X = scaler.fit_transform(X)
    #shuffle the data with seed=i
    shuffle_idxs = np.random.default_rng(seed=i).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]
    #Desicion Tree Classifier
    dtc = DecisionTreeClassifier(criterion='entropy', random_state=4)
    dtc.fit(X_shff, y)
    importances = clf.feature_importances_
    #descending order
    indices = np.argsort(importances)[::-1][:5] #top 5 features
    #map the top 5 features to the original data
    top5_tree = shuffle_idxs[indices]
    #Logistics Regression
    lr = LogisticRegression(penalty=None, random_state=4)
    lr.fit(X_shff,y)
    coef = np.abs(lr.coef_[0])
    #descending order
    indices = np.argsort(coef)[::-1][:5] #top 5 features
    top5_lr = shuffle_idxs[indices]
    overlaps.append(len(set(top5_tree)&set(top5_lr)))
#plot
plt.hist(overlaps,bins=np.arange(7)-0.5,edgecolor='black')
plt.xlabel('Number of Overlapping Features')
plt.title('Overlap Between Decision Tree and Logistic Regression')
plt.show()
```

The result shows as following:



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

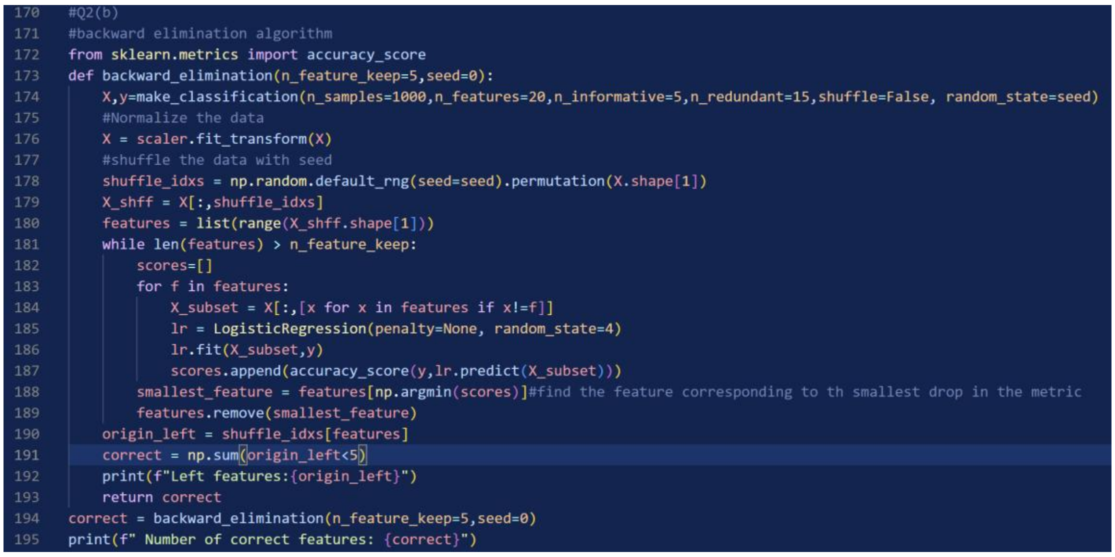
1. Model bias: Different models emphasize different features, like decision tree and logistics regression.

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.

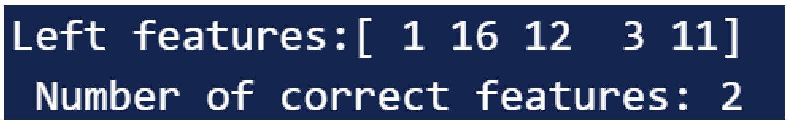
Q2(a) Because at each round, we remove the j-th feature from the model based on the drop in the value of a certain metric. We eliminate the feature corresponding to the smallest drop in the metric, which is only the local optimal choice and not the global optimal choice. The pitfalls of greedy algorithms 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 e implementing the backward elimination algorithm as followings:

```python
#Q2(b)
#backward elimination algorithm
from sklearn.metrics import accuracy_score
def backward_elimination(n_feature_keep=5,seed=0):

    X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=seed)
    #Normalize the data
    X = scaler.fit_transform(X)
    #shuffle the data with seed
    shuffle_idxs = np.random.default_rng(seed=seed).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]
    features = list(range(X_shff.shape[1]))
    while len(features) > n_feature_keep:
        scores=[]
        for f in features:
            X_subset = X[:,[x for x in features if x!=f]]
            lr = LogisticRegression(penalty=None, random_state=4)
            lr.fit(X_subset,y)
            scores.append(accuracy_score(y,lr.predict(X_subset)))
        smallest_feature = features[np.argmin(scores)]#find the feature corresponding to th smallest drop in the metric
        features.remove(smallest_feature)
    origin_left = shuffle_idxs[features]
    correct = np.sum(origin_left<5)
    print(f"Left features:{origin_left}")
    return correct
correct = backward_elimination(n_feature_keep=5,seed=0)
print(f" Number of correct features: {correct}")
```



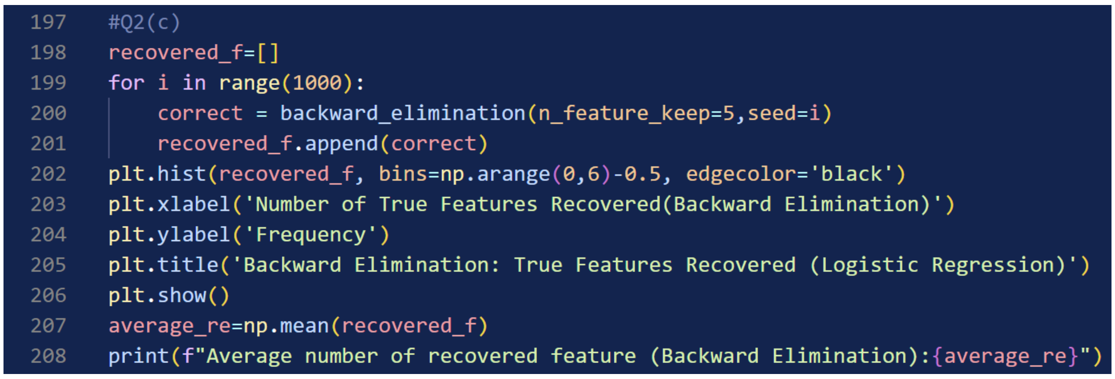
```python
#Q2(b)
#backward elimination algorithm
from sklearn.metrics import accuracy_score
def backward_elimination(n_feature_keep=5,seed=0):
    X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=seed)
    #Normalize the data
    X = scaler.fit_transform(X)
    #shuffle the data with seed
    shuffle_idxs = np.random.default_rng(seed=seed).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]
    features = list(range(X_shff.shape[1]))
    while len(features) > n_feature_keep:
        scores=[]
        for f in features:
            X_subset = X[:,[x for x in features if x!=f]]
            lr = LogisticRegression(penalty=None, random_state=4)
            lr.fit(X_subset,y)
            scores.append(accuracy_score(y,lr.predict(X_subset)))
        smallest_feature = features[np.argmin(scores)]#find the feature corresponding to th smallest drop in the metric
        features.remove(smallest_feature)
    origin_left = shuffle_idxs[features]
    correct = np.sum(origin_left<5)
    print(f"Left features:{origin_left}")
    return correct
correct = backward_elimination(n_feature_keep=5,seed=0)
print(f" Number of correct features: {correct}")
```

The result is: 

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

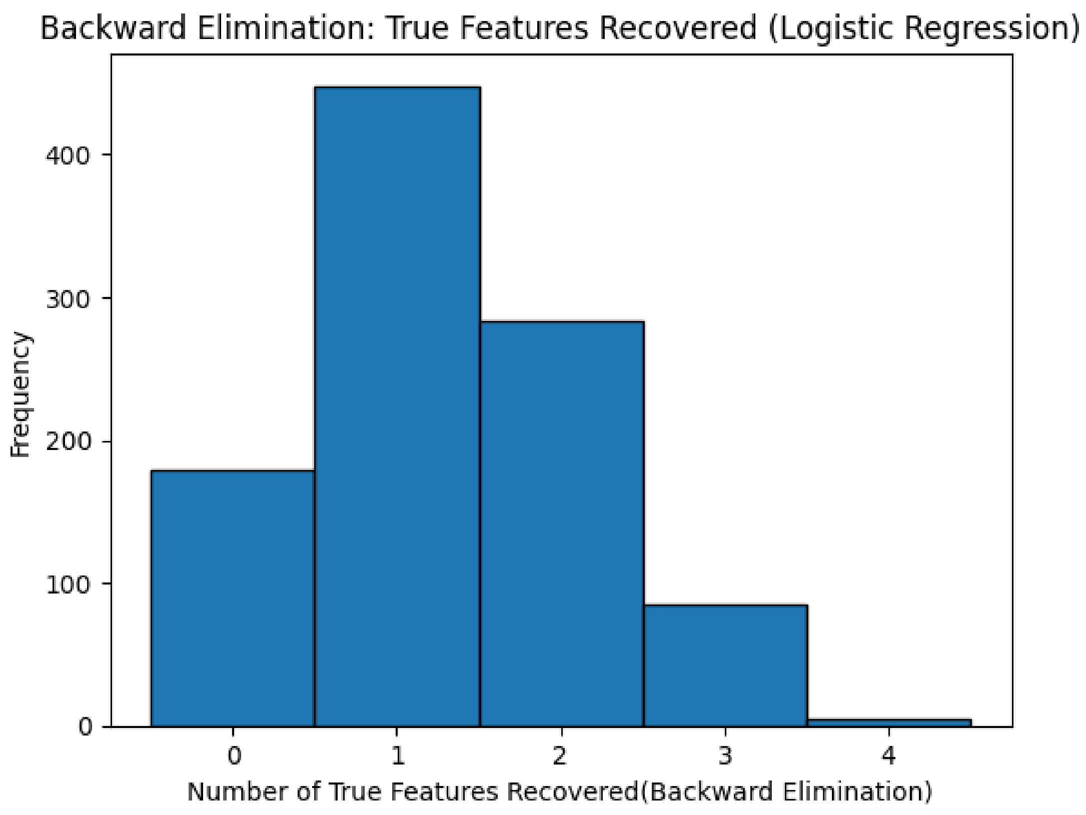
Q2(c) Repeat part a for 1000:

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



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

The histogram chart is like followings:



The average number of recovered feature is:

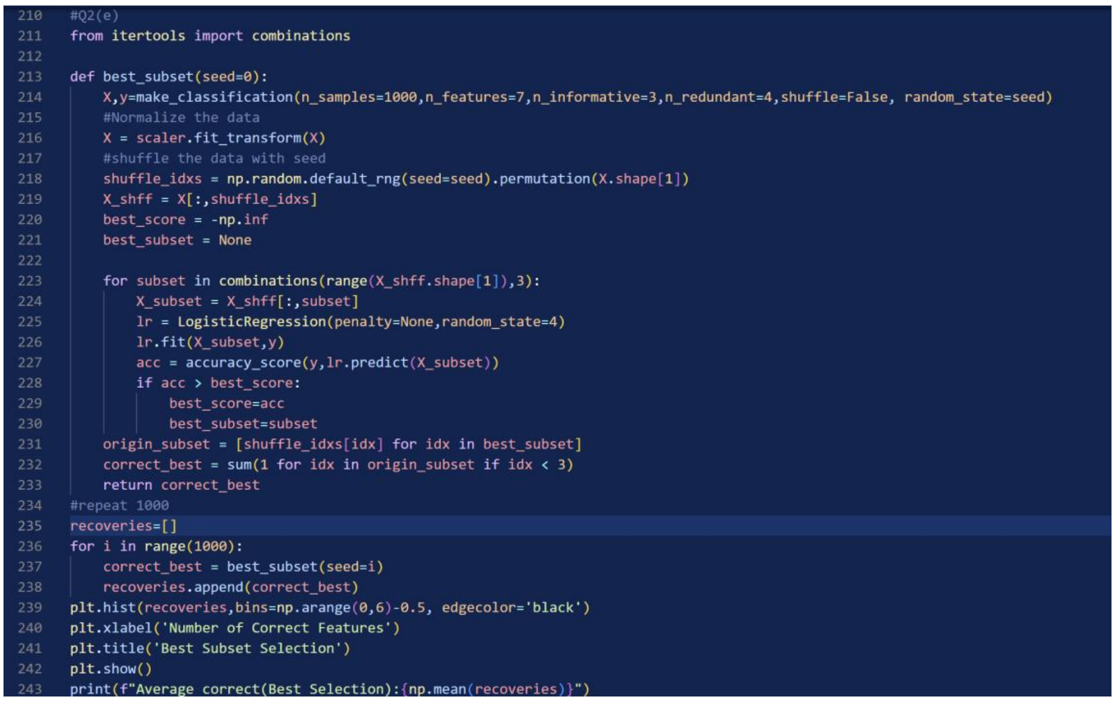
```
Average number of recovered feature (Backward Elimination):1.289
```

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:

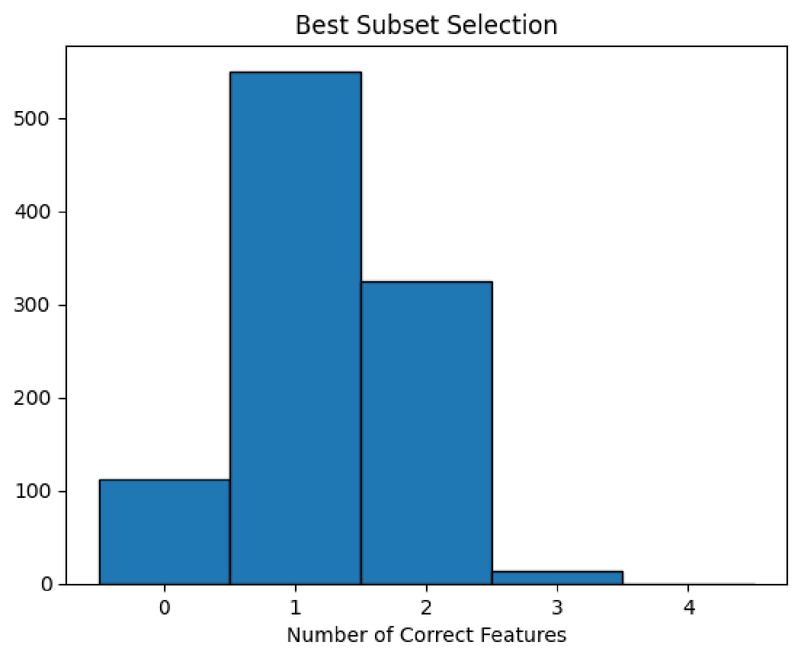
```
#Q2(e)
from itertools import combinations

def best_subset(seed=0):
    X,y=make_classification(n_samples=1000,n_features=7,n_informative=3,n_redundant=4,
shuffle=False, random_state=seed)
    #Normalize the data
    X = scaler.fit_transform(X)
    #shuffle the data with seed
```

```python
    shuffle_idxs = np.random.default_rng(seed=seed).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]
    best_score = -np.inf
    best_subset = None

    for subset in combinations(range(X_shff.shape[1]),3):
        X_subset = X_shff[:,subset]
        lr = LogisticRegression(penalty=None,random_state=4)
        lr.fit(X_subset,y)
        acc = accuracy_score(y,lr.predict(X_subset))
        if acc > best_score:
            best_score=acc
            best_subset=subset
    origin_subset = [shuffle_idxs[idx] for idx in best_subset]
    correct_best = sum(1 for idx in origin_subset if idx < 3)
    return correct_best
#repeat 1000
recoveries=[]
for i in range(1000):
    correct_best = best_subset(seed=i)
    recoveries.append(correct_best)
plt.hist(recoveries,bins=np.arange(0,6)-0.5, edgecolor='black')
plt.xlabel('Number of Correct Features')
plt.title('Best Subset Selection')
plt.show()
print(f"Average correct(Best Selection):{np.mean(recoveries)}")
```

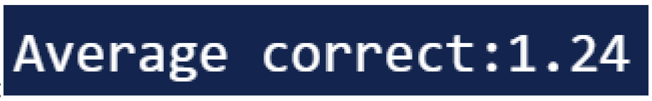


```python
#Q2(e)
from itertools import combinations

def best_subset(seed=0):
    X,y=make_classification(n_samples=1000,n_features=7,n_informative=3,n_redundant=4,shuffle=False, random_state=seed)
    #Normalize the data
    X = scaler.fit_transform(X)
    #shuffle the data with seed
    shuffle_idxs = np.random.default_rng(seed=seed).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]
    best_score = -np.inf
    best_subset = None

    for subset in combinations(range(X_shff.shape[1]),3):
        X_subset = X_shff[:,subset]
        lr = LogisticRegression(penalty=None,random_state=4)
        lr.fit(X_subset,y)
        acc = accuracy_score(y,lr.predict(X_subset))
        if acc > best_score:
            best_score=acc
            best_subset=subset
    origin_subset = [shuffle_idxs[idx] for idx in best_subset]
    correct_best = sum(1 for idx in origin_subset if idx < 3)
    return correct_best
#repeat 1000
recoveries=[]
for i in range(1000):
    correct_best = best_subset(seed=i)
    recoveries.append(correct_best)
plt.hist(recoveries,bins=np.arange(0,6)-0.5, edgecolor='black')
plt.xlabel('Number of Correct Features')
plt.title('Best Subset Selection')
plt.show()
print(f"Average correct(Best Selection):{np.mean(recoveries)}")
```

The histogram chart is below:



The average is:

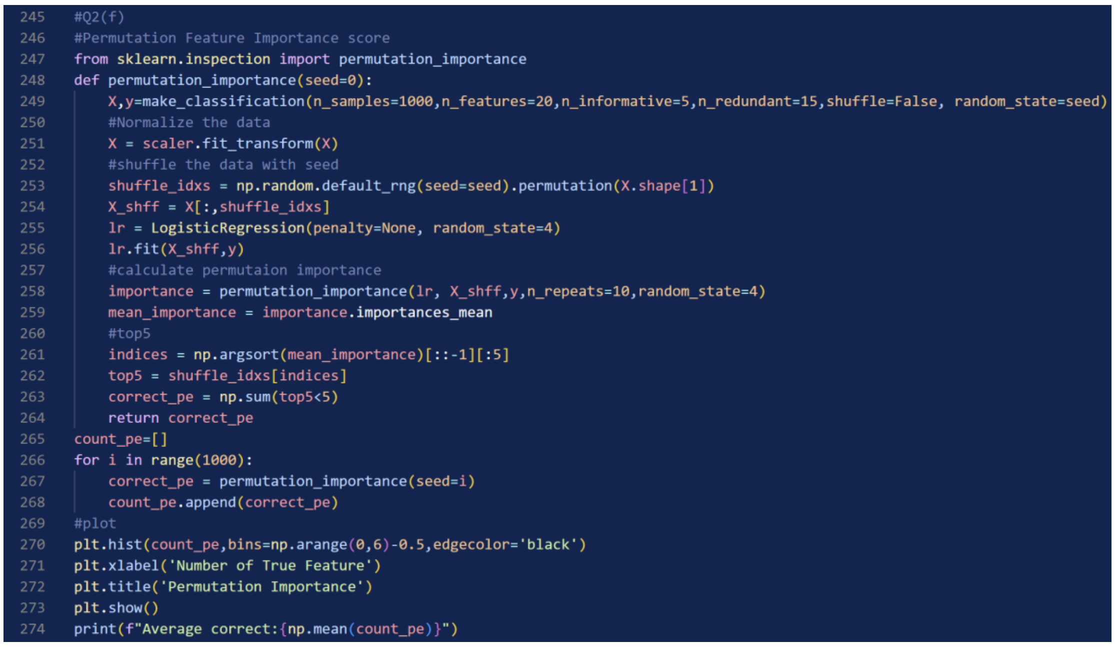


```
Average correct:1.24
```

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.

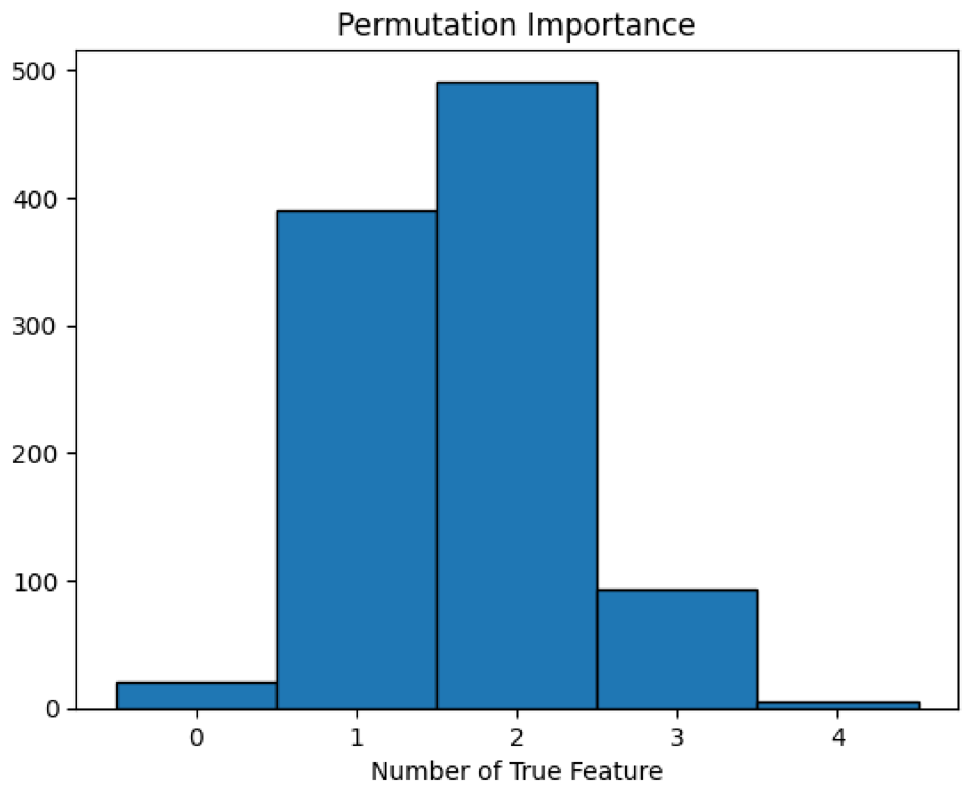
The implement code:

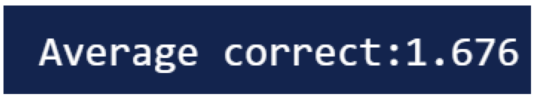
```python
#Q2(f)
#Permutation Feature Importance score
from sklearn.inspection import permutation_importance
def permutation_importance(seed=0):

    X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=seed)
    #Normalize the data
    X = scaler.fit_transform(X)
    #shuffle the data with seed
    shuffle_idxs = np.random.default_rng(seed=seed).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]
    lr = LogisticRegression(penalty=None, random_state=4)
    lr.fit(X_shff,y)
    #calculate permutaion importance
    importance = permutation_importance(lr, X_shff,y,n_repeats=10,random_state=4)
    mean_importance = importance.importances_mean
    #top5
    indices = np.argsort(mean_importance)[::-1][:5]
    top5 = shuffle_idxs[indices]
    correct_pe = np.sum(top5<5)
    return correct_pe
count_pe=[]
for i in range(1000):
    correct_pe = permutation_importance(seed=i)
    count_pe.append(correct_pe)
#plot
plt.hist(count_pe,bins=np.arange(0,6)-0.5,edgecolor='black')
plt.xlabel('Number of True Feature')
plt.title('Permutation Importance')
plt.show()
print(f"Average correct:{np.mean(count_pe)}")
```



```
#Q2(f)
#Permutation Feature Importance score
from sklearn.inspection import permutation_importance
def permutation_importance(seed=0):
    X,y=make_classification(n_samples=1000,n_features=20,n_informative=5,n_redundant=15,shuffle=False, random_state=seed)
    #Normalize the data
    X = scaler.fit_transform(X)
    #shuffle the data with seed
    shuffle_idxs = np.random.default_rng(seed=seed).permutation(X.shape[1])
    X_shff = X[:,shuffle_idxs]
    lr = LogisticRegression(penalty=None, random_state=4)
    lr.fit(X_shff,y)
    #calculate permutaion importance
    importance = permutation_importance(lr, X_shff,y,n_repeats=10,random_state=4)
    mean_importance = importance.importances_mean
    #top5
    indices = np.argsort(mean_importance)[::-1][:5]
    top5 = shuffle_idxs[indices]
    correct_pe = np.sum(top5<5)
    return correct_pe
count_pe=[]
for i in range(1000):
    correct_pe = permutation_importance(seed=i)
    count_pe.append(correct_pe)
#plot
plt.hist(count_pe,bins=np.arange(0,6)-0.5,edgecolor='black')
plt.xlabel('Number of True Feature')
plt.title('Permutation Importance')
plt.show()
print(f"Average correct:{np.mean(count_pe)}")
```

The histogram chart of permutation importance as below:



The average is: 

```
Average correct:1.676
```