Assignment1

Q1 (a) the picture shows how to create a variable X and a variably Y, first need to import some package and then import the 'heart.csv', the code is as follows:

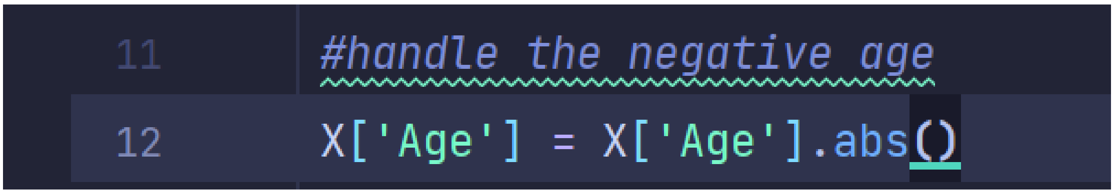
```
import pandas as pd
import numpy as np
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt

#import data and spilt X and y
data = pd.read_csv("heart.csv", keep_default_na= False, na_values=[""])
X = data.drop(columns=['Last_Checkup', 'Heart_Disease'],axis=1)
y = data['Heart_Disease']

```

(b)For Age I transfer the negative value to positive value use following code:

```
#handle the negative age
X['Age'] = X['Age'].abs()
```



(c)Make Gender and Smoker columns encode consistent, use the following code and the result is as following:

```
#make these codings consistent and categorical encoding
mapping_G = {'Male': 0, 'M': 0, 'Female':1, 'F': 1, 'Unknown': 2}
X['Gender'] = X['Gender'].map(mapping_G)
```

```
mapping_S = {'No':0, 'N': 0, 'Yes': 1, 'Y': 1, 'nan': 2}
X['Smoker'] = X['Smoker'].map(mapping_S)
```

```
#translate gender
#make these codings consistent and categorical encoding
mapping_G = {'Male': 0, 'M': 0, 'Female':1, 'F': 1, 'Unknown': 2}
X['Gender'] = X['Gender'].map(mapping_G)
mapping_S = {'No':0, 'N': 0, 'Yes': 1, 'Y': 1, 'nan': 2}
X['Smoker'] = X['Smoker'].map(mapping_S)
print(X.iloc[0:15,:])
```

	Age	Gender	Height_feet	Weight_kg	Blood_Pressure	Cholesterol	Smoker
0	45.0	0	5.0512	61	120/80	207	0
1	101.0	1	4.9856	78	130/85	208	0
2	NaN	0	5.3792	51	125/75	267	1
3	58.0	0	5.2152	126	138/88	221	0
4	64.0	1	4.9200	102	142/90	182	0
5	50.0	0	6.1336	75	115/70	245	2
6	69.0	2	5.6416	67	128/82	274	1
7	31.0	0	5.0840	116	135/78	256	1
8	70.0	1	5.3792	95	140/85	250	1
9	31.0	1	5.9696	118	122/76	191	1
10	69.0	1	5.7728	125	130/80	236	1
11	21.0	1	6.0352	109	125/78	209	0
12	91.0	0	5.9368	115	136/84	222	0
13	80.0	1	6.0352	68	145/92	260	1
14	71.0	0	5.8384	63	118/74	257	0

(d)The following code is to split ‘Blood_Pressure’ column into ‘Systolic’ and ‘Diastolic’ columns and the result is shown (first 20) as:

```
#split Blood
X[['Systolic','Diastolic']] = X['Blood_Pressure'].str.split('/',expand =
True).astype(int)
X = X.drop(columns=['Blood_Pressure'])
print("After split:\n",X.iloc[0:15,:])5.
```

```
#split Blood
X[['Systolic','Diastolic']] = X['Blood_Pressure'].str.split('/',expand = True).astype(int)
X = X.drop(columns=['Blood_Pressure'])
print("After split:\n",X.iloc[0:15,:])
```

```
After split:
       Age  Gender  Height_feet  ...  Smoker  Systolic  Diastolic
0     45.0       0       5.0512  ...       0       120         80
1    101.0       1       4.9856  ...       0       130         85
2      NaN       0       5.3792  ...       1       125         75
3     58.0       0       5.2152  ...       0       138         88
4     64.0       1       4.9200  ...       0       142         90
5     50.0       0       6.1336  ...       2       115         70
6     69.0       2       5.6416  ...       1       128         82
7     31.0       0       5.0840  ...       1       135         78
8     70.0       1       5.3792  ...       1       140         85
9     31.0       1       5.9696  ...       1       122         76
10    69.0       1       5.7728  ...       1       130         80
11    21.0       1       6.0352  ...       0       125         78
12    91.0       0       5.9368  ...       0       136         84
13    80.0       1       6.0352  ...       1       145         92
14    71.0       0       5.8384  ...       0       118         74
```

(e) Then use the train_test_split, create the feature training and test sets like followings:

```
#split into train, test, test_size= =0.3,random_state=2
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=2)
```

```
#split train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train,y_test = train_test_split(X,y,test_size=0.3, random_state=2)
```

(f)Follow the rules to calculate the median and impute values, the code is:

```
#imputr value in Age: drop nan, calculate mediean
X_train_without_nan = X_train.dropna(subset=['Age'])
median_age_by_gender = X_train_without_nan.groupby('Gender')['Age'].median()
for index, row in X_test.iterrows():
    if pd.isna(row['Age']):
        gender = row['Gender']
        X_test.at[index, 'Age'] = median_age_by_gender[gender]
```

```python
#fill the Age
X_train_without_nan = X_train.dropna(subset=['Age'])
median_age_by_gender = X_train_without_nan.groupby('Gender')['Age'].median()
for index, row in X_test.iterrows():
    if pd.isna(row['Age']):
        gender = row['Gender']
        X_test.at[index, 'Age'] = median_age_by_gender[gender]
```

The result is like followings, and we could find out the value of the row (index = 2) which in the ‘Age’ column now is equal to 74 not NaN(could find from last result)

```
[15 rows x 8 columns]
      Age  Gender  Height_feet  ...  Smoker  Systolic  Diastolic
83  116.0       0       6.0024  ...       0       136         83
30   74.0       0       5.1496  ...       2       137         89
56   74.0       0       6.1336  ...       1       141         89
24   68.0       1       5.1496  ...       1       135         82
16   68.0       1       5.3136  ...       1       124         79
23   74.0       0       6.1336  ...       0       126         79
2    74.0       0       5.3792  ...       1       125         75
27   74.0       0       5.1824  ...       1       134         86
28  100.0       1       5.9368  ...       1       128         80
13   80.0       1       6.0352  ...       1       145         92
99   45.0       1       5.8056  ...       2       127         80
92   24.0       0       6.1664  ...       0       138         90
76   44.0       0       6.0352  ...       0       127         80
14   71.0       0       5.8384  ...       0       118         74
0    45.0       0       5.0512  ...       0       120         80
```

(g)Now we could import a library to help us scale the coulmns: ’Age’, ’Height_feet’, ’Weight_kg’, ’Cholesterol’, ’Systolic’, ’Diastolic’, the code is as following:

```python
#StandarScale
from sklearn.preprocessing import MinMaxScaler
col_scale =  ['Age', 'Height_feet', 'Weight_kg', 'Cholesterol', 'Systolic', 'Diastolic']
scaler = MinMaxScaler()
X_train[col_scale] = scaler.fit_transform(X_train[col_scale])
X_test[col_scale] = scaler.transform(X_test[col_scale])
```

```python
#StandarScale
from sklearn.preprocessing import MinMaxScaler
col_scale = ['Age', 'Height_feet', 'Weight_kg', 'Cholesterol', 'Systolic', 'Diastolic']
scaler = MinMaxScaler()
X_train[col_scale] = scaler.fit_transform(X_train[col_scale])
X_test[col_scale] = scaler.transform(X_test[col_scale])
print("After standarscaler X_test",X_test.iloc[0:15,:])
print("After standarscaler X_train",X_train.iloc[0:15,:])
```

The result is like: (first picture is the X_train, Second one is X_test)

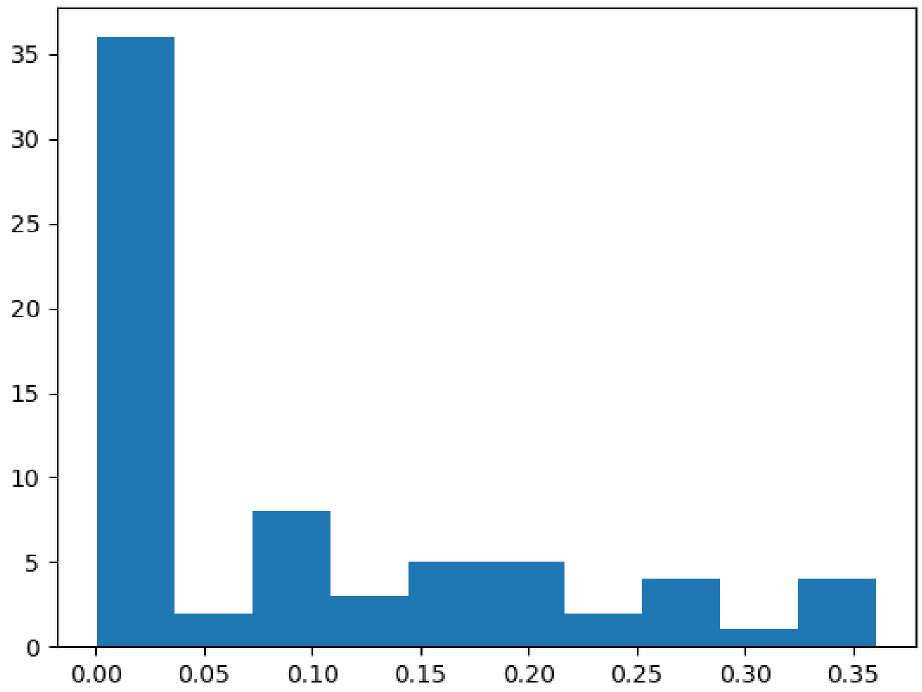
```
[15 rows x 8 columns]
After standarscaler X_test:
     Age  Gender  Height_feet  ...  Smoker  Systolic  Diastolic
83  0.98       0     0.326910  ...       0  0.700000   0.583333
30  0.56       0     0.069345  ...       2  0.733333   0.833333
56  0.56       0     0.366536  ...       1  0.866667   0.833333
24  0.50       1     0.069345  ...       1  0.666667   0.541667
16  0.50       1     0.118876  ...       1  0.300000   0.416667
23  0.56       0     0.366536  ...       0  0.366667   0.416667
2   0.56       0     0.138689  ...       1  0.333333   0.250000
27  0.56       0     0.079251  ...       1  0.633333   0.708333
28  0.82       1     0.307098  ...       1  0.433333   0.458333
13  0.62       1     0.336817  ...       1  1.000000   0.958333
99  0.27       1     0.267472  ...       2  0.400000   0.458333
92  0.06       0     0.376442  ...       0  0.766667   0.875000
76  0.26       0     0.336817  ...       0  0.400000   0.458333
14  0.53       0     0.277378  ...       0  0.100000   0.208333
0   0.27       0     0.039625  ...       0  0.166667   0.458333
```

```
[15 rows x 8 columns]
After standarscaler X_train:
        Age  Gender  Height_feet  ...  Smoker  Systolic  Diastolic
65  0.63       0      0.326910  ...       1  0.366667   0.375000
1   0.83       1      0.019813  ...       0  0.500000   0.666667
18  0.17       1      0.128783  ...       0  0.766667   0.708333
48  0.23       0      0.217940  ...       0  0.766667   0.625000
36  0.08       0      0.178315  ...       0  0.866667   0.875000
78  0.83       1      0.277378  ...       1  0.233333   0.250000
6   0.51       2      0.217940  ...       1  0.433333   0.541667
89  0.62       0      0.336817  ...       1  0.633333   0.708333
91  0.27       1      0.198127  ...       1  0.666667   0.625000
10  0.51       1      0.257566  ...       1  0.500000   0.458333
12  0.73       0      0.307098  ...       0  0.700000   0.625000
53  0.16       1      0.168408  ...       1  0.800000   0.708333
87  0.63       1      0.029719  ...       1  0.366667   0.416667
54  0.12       0      0.049532  ...       1  0.033333   0.125000
95  0.48       0      1.000000  ...       0  0.133333   0.166667
```

(h) Plot the histogram of the target value then find out that indeed t a large portion of the target value is clustered around zero, so I think the linear regression is not a reasonable model for this data, the histogram is like following and when I set threshold as 0.1, it can be verified that the quantization operation is correct, the result is in the second picture.

```
#plot
plt.hist(y_train)
plt.title('Distribution of Heart Disease')
plt.show()
plt.savefig("Histogram")
plt.clf()
#quantified
threshold = 0.1
y_train_q = (y_train>=threshold).astype(int)
y_test_q = (y_test>=threshold).astype(int)
print("original target value(first 15):\n", y_train[:15])
print("original target value(first 15):\n", y_train_q[:15])
```

```python
#plot
plt.hist(y_train)
plt.title('Distribution of Heart Disease')
plt.show()
plt.savefig("Histogram")
plt.clf()
#quantified
threshold = 0.1
y_train_q = (y_train>=threshold).astype(int)
y_test_q = (y_test>=threshold).astype(int)
print("original target value(first 15):\n", y_train[:15])
print("original target value(first 15):\n", y_train_q[:15])
```



```
original target value(first 15):
 65    0.001595
1     0.192284
18    0.138022
48    0.207859
36    0.075575
78    0.005046
6     0.005940
89    0.002250
91    0.001893
10    0.003016
12    0.111802
53    0.002540
87    0.002577
54    0.003147
95    0.352998
Name: Heart_Disease, dtype: float64
```

```
original target value(first 15):
 65    0
1     1
18    1
48    1
36    0
78    0
6     0
89    0
91    0
10    0
12    1
53    0
87    0
54    0
95    1
Name: Heart_Disease, dtype: int64
```

Q2(a) The whole reasoning process is as follows:

(a). minimize function in sklearn: $\hat{w}, \hat{c} = \arg\min_{w,c} \{ \text{penalty}(w) + C \sum_{i=1}^{n} \log(1+\exp(-\tilde{y}_i (w^T x_i + c))) \}$ $\tilde{y}_i \in \{-1, 1\}$

$y_i \in \{0, 1\}$

(l_2-regularized) log-loss function: $L(\beta_0, \beta) = \text{penalty}(\beta) + \frac{\lambda}{n} \sum_{i=1}^{n} [y_i \ln(\frac{1}{\sigma(\beta_0 + \beta^T x_i)}) + (1-y_i) \ln(\frac{1}{1-\sigma(\beta_0+\beta^T x_i)})]$ $y_i \in \{0, 1\}$

$\sigma(z) = (1+e^{-z})^{-1}$

$L(\beta_0, \beta) = \text{penalty}(\beta) + \frac{\lambda}{n} \sum_{i=1}^{n} [y_i \ln(\frac{1}{\sigma(\beta_0 + \beta^T x_i)}) + (1-y_i) \ln(\frac{1}{1-\sigma(\beta_0+\beta^T x_i)})]$ This function we can simplifed to

$L(\beta_0, \beta) = \text{penalty}(\beta) + \frac{\lambda}{n} \sum_{i=1}^{n} [y_i \ln(1+e^{-z}) + (1-y_i) \ln(e^{z}+1)]$ since we already know $\sigma(z) = \frac{1}{1+e^{-z}}$

When $y_i = 1$, $\tilde{y}_i = 1$ this term becomes $\ln(1+e^{-z})$.

when $y_i = 0$, $\tilde{y}_i = -1$ this term becomes $\ln(1+e^{z})$

$\therefore$ we could know $y_i \ln(1+e^{-z}) + (1-y_i) \ln(e^{z}+1)$ can be simplifed as $\ln(1+e^{-\tilde{y}_i z})$

$\therefore L(\beta_0, \beta) = \text{penalty}(\beta) + \frac{\lambda}{n} \sum_{i=1}^{n} \ln(1+e^{-\tilde{y}_i(\beta_0 + \beta^T x_i)})$

For sklearn: $\hat{w}, \hat{c} = \arg\min_{w,c} \{ \text{penalty}(w) + C \sum_{i=1}^{n} \log(1+\exp(-\tilde{y}_i (w^T x_i + c))) \}$

$= \arg\min_{w,c} \{ \text{penalty}(w) + C \sum_{i=1}^{n} \log(1+e^{-\tilde{y}_i (w^T x_i + c)}) \}$

let $c = \beta_0$, $w = \beta$, assume $\text{penalty}(w) = \text{penalty}(\beta)$ and $C = \frac{1}{n}$ after re-encoding the labels from $y_i \in \{0, 1\}$ to $\tilde{y}_i \in \{-1, 1\}$, the two objective function are equivalent, that will give the same solution $\hat{\beta}_0 = \hat{c}$, $\hat{\beta} = \hat{w}$.

The role of ***C***: In the objective function of sklearn, *C* is the reciprocal of the regularization strength. A larger *C* means a smaller weight for the regularization term penalty(*w*), and the model prefers to minimize the empirical risk, it may lead to overfitting. A smaller *C* gives a larger weight to the regularization term, and the model prefers simplicity to avoid overfitting.

The relationship with λ: in the (ℓ2-regularized) log-loss function, we could find out that the value of λ will have the positive effect the function. Just like the larger λ will have a stronger regularization, the smaller λ will have weaker regularization. So we could know $C = \frac{1}{\lambda}$.

(b)In this part, first we create C grid as required, and then make evey C fit a logistic regression model on the training data then plot it, the code and chart is like followings:

```
#Q2(b)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
Cs = np.logspace(-4, 4, 100)
train_losses=[]
test_losses = []
for C in Cs:
    model = LogisticRegression(C=C,penalty='l2', solver='lbfgs')
    model.fit(X_train,y_train_q)
    #train loss
    train_probs = model.predict_proba(X_train)[:,1]
    train_loss = log_loss(y_train_q,train_probs)
    train_losses.append(train_loss)
    #test loss
    test_probs = model.predict_proba(X_test)[:,1]
    test_loss = log_loss(y_test_q, test_probs)
```

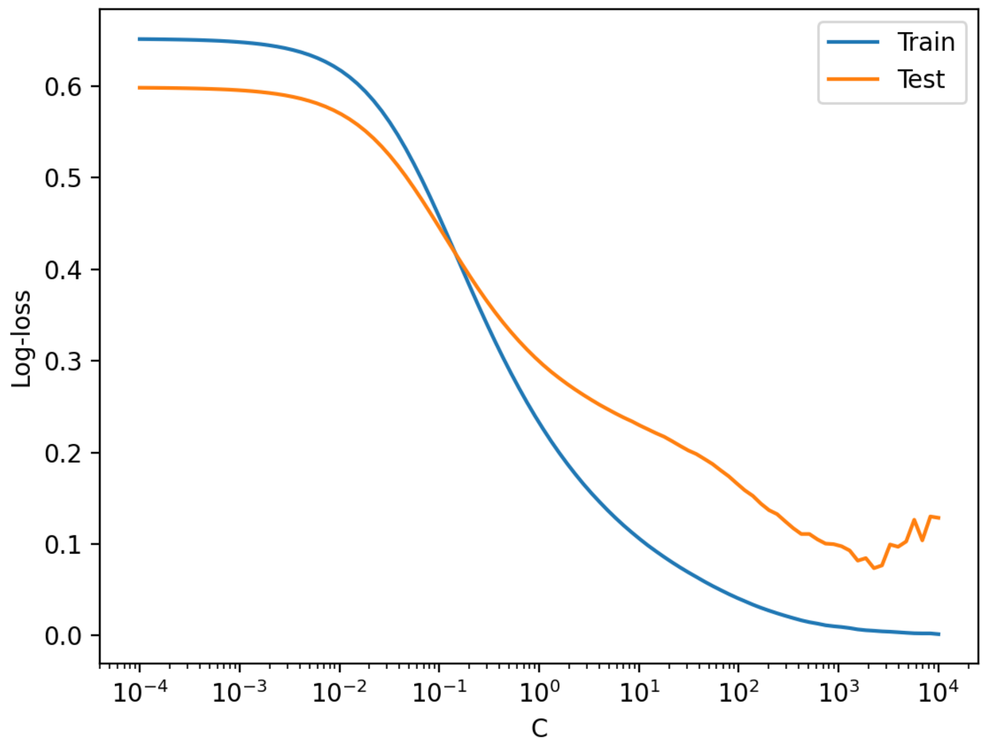
```
    test_losses.append(test_loss)

#plot
plt.plot(Cs,train_losses,label='Train')
plt.plot(Cs,test_losses,label='Test')
plt.xscale('log')
plt.xlabel('C')
plt.ylabel('Log-loss')
plt.legend()
plt.show()
plt.savefig("Log-loss")
```

```
#Q2(b)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
Cs = np.logspace(-4, 4, 100)
train_losses=[]
test_losses = []
for C in Cs:
    model = LogisticRegression(C=C,penalty='l2', solver='lbfgs')
    model.fit(X_train,y_train_q)
    #train loss
    train_probs = model.predict_proba(X_train)[:,1]
    train_loss = log_loss(y_train_q,train_probs)
    train_losses.append(train_loss)
    #test loss
    test_probs = model.predict_proba(X_test)[:,1]
    test_loss = log_loss(y_test_q, test_probs)
    test_losses.append(test_loss)
```

```python

#plot
plt.plot(Cs,train_losses,label='Train')
plt.plot(Cs,test_losses,label='Test')
plt.xscale('log')
plt.xlabel('C')
plt.ylabel('Log-loss')
plt.legend()
plt.show()
plt.savefig("Log-loss")
```



From the chart we could find out, as the regularization strength of C increases, the log-loss curve of the training set begins to decline at around $C = 10^{-2}$, and gradually approaches 0 at $C = 10^{4}$. The log-loss curve of the test set also begins to decline at around $C = 10^{-2}$, and begins to rise at around $C = 10^{2.5}$ and begins to fluctuate to a certain extent. From the chart I think the best C should choose the value when the test loss at the lowest point, and at that time train loss have not reached 0, and can get the suitable model.

(c) In this part, we split the train data into 5 folds as required, and not use the existing cross validation. After fit logistic regression, we produce the box-plot and calculate the train and test accuracy, the code and plot are as followings:

```
1. #Q2(c) 5fold
import seaborn as sns
fold_size = len(X_train)//5
CV_scores = []
for C in Cs:
    fold_losses = []
    for fold in range(5):
        val_start = fold * fold_size
        val_end = (fold+1) *fold_size
        X_val = X_train.iloc[val_start:val_end]
        y_val = y_train_q.iloc[val_start:val_end]
        train_X = pd.concat([X_train.iloc[:val_start], X_train.iloc[val_end:]])
        train_y = pd.concat([y_train_q.iloc[:val_start], y_train_q.iloc[val_end:]])

        model = LogisticRegression(C=C,penalty='l2',solver='lbfgs')
        model.fit(train_X,train_y)
        val_probs = model.predict_proba(X_val)[:,1]
        loss= (log_loss(y_val,val_probs))
        fold_losses.append(loss)
    CV_scores.append(fold_losses)


sns.boxplot(data=CV_scores)
plt.xscale('log')
plt.xlabel('X(log scale)')
plt.ylabel('Log Loss')
plt.title('5-Fold Cross Validation Log Loss for Different C Value')
plt.show()
plt.savefig("Boxplot")


best_index = np.argmin([np.mean(scores) for scores in CV_scores])
best_C = Cs[best_index]
final_model = LogisticRegression(C=best_C, penalty='l2', solver='lbfgs')
final_model.fit(X_train,y_train_q)
train_acc = final_model.score(X_train,y_train_q)
test_acc = final_model.score(X_test, y_test_q)
print("Best C:", best_C)
print("Train Accuracy:", train_acc)
```

```
print("Test Accuracy:", test_acc)

```

```
#Q2(c) 5fold
import seaborn as sns
fold_size = len(X_train)//5
CV_scores = []
for C in Cs:
    fold_losses = []
    for fold in range(5):
        val_start = fold * fold_size
        val_end = (fold+1) *fold_size
        X_val = X_train.iloc[val_start:val_end]
        y_val = y_train_q.iloc[val_start:val_end]
        train_X = pd.concat([X_train.iloc[:val_start], X_train.iloc[val_end:]])
        train_y = pd.concat([y_train_q.iloc[:val_start], y_train_q.iloc[val_end:]])

        model = LogisticRegression(C=C,penalty='l2',solver='lbfgs')
        model.fit(train_X,train_y)
        val_probs = model.predict_proba(X_val)[:,1]
        loss= (log_loss(y_val,val_probs))
        fold_losses.append(loss)
    CV_scores.append(fold_losses)
```

```
sns.boxplot(data=CV_scores)
plt.xscale('log')
plt.xlabel('X(log scale)')
plt.ylabel('Log Loss')
plt.title('5-Fold Cross Validation Log Loss for Different C Value')
plt.show()
plt.savefig("Boxplot")


best_index = np.argmin([np.mean(scores) for scores in CV_scores])
best_C = Cs[best_index]
final_model = LogisticRegression(C=best_C, penalty='l2', solver='lbfgs')
final_model.fit(X_train,y_train_q)
train_acc = final_model.score(X_train,y_train_q)
test_acc = final_model.score(X_test, y_test_q)
print("Best C:", best_C)
print("Train Accuracy:", train_acc)
print("Test Accuracy:", test_acc)
```

The BestC and train, test accuracy is as followings:

```
Best C: 37.649358067924716

Train Accuracy: 1.0

Test Accuracy: 0.8666666666666667
```

(d)After run the code, I think the reasons of why there are different between the code and previous is : 1. The different data split in thses two part. 2.The different regularisation. 3.The different evaluation metrics. So after I modify the code(also use the 5-fold cross validation), these two parts will have the same results.

```
1.  from sklearn.model_selection import GridSearchCV,KFold
2. CV = KFold(n_splits=5, shuffle=False)
3. param_grid = {'C': Cs}
4. grid_lr = GridSearchCV(estimator=LogisticRegression(penalty='l2',solver='lbfgs'),
param_grid=param_grid, cv=CV,scoring='neg_log_loss')
5. grid_lr.fit(X_train,y_train_q)
6. print("Best C(GridSearchCV):",grid_lr.best_params_['C'])
7.
```


```
from sklearn.model_selection import GridSearchCV,KFold
CV = KFold(n_splits=5, shuffle=False)
param_grid = {'C': Cs}
grid_lr = GridSearchCV(estimator=LogisticRegression(penalty='l2',solver='lbfgs'), param_grid=param_grid, cv=CV,scoring='neg_log_loss')
grid_lr.fit(X_train,y_train_q)
print("Best C(GridSearchCV):",grid_lr.best_params_['C'])
```


(e)The answer is in following picture:

(e) We already know $L(\beta_0, \beta) = \frac{1}{2}\|\beta\|_2^2 + \frac{\lambda}{n}\sum_{i=1}^{n}[y_i \ln(\frac{1}{\sigma(z_i)}) + (1-y_i)\ln(\frac{1}{1-\sigma(z_i)})]$ $\quad z_i = \beta_0 + \beta^T x_i \quad \sigma(z) = \frac{1}{1+e^{-z}}$

For intercept $\frac{\partial L}{\partial \beta_0} = \frac{\lambda}{n}\sum_{i=1}^{n}\frac{\partial}{\partial \beta_0}[-y_i \ln\sigma(z_i) - (1-y_i)\ln(1-\sigma(z_i))]$

and for each i: $\frac{\partial}{\partial \beta_0}[-y_i \ln\sigma(z_i) - (1-y_i)\ln(1-\sigma(z_i))] = -y_i\frac{\sigma(z_i)(1-\sigma(z_i))}{\sigma(z_i)} + (1-y_i)\frac{\sigma(z_i)(1-\sigma(z_i))}{1-\sigma(z_i)} = \sigma(z_i) - y_i$

$\therefore$ the total Gradient is: $\frac{\partial L}{\partial \beta_0} = \frac{\lambda}{n}\sum_{i=1}^{n}(\sigma(z_i) - y_i)$

$\therefore$ the update rule is $\beta_0^{(k)} = \beta_0^{(k-1)} - \eta\frac{\lambda}{n}\sum_{i=1}^{n}(\sigma(z_i) - y_i)$

For parameter $\frac{\partial L}{\partial \beta_j} = \beta_j + \frac{\lambda}{n}\sum_{i=1}^{n}\frac{\partial}{\partial \beta_j}[-y_i \ln\sigma(z_i) - (1-y_i)\ln(1-\sigma(z_i))]$

for each i: $\frac{\partial}{\partial \beta_j}[-y_i \ln\sigma(z_i) - (1-y_i)\ln(1-\sigma(z_i))] = (\sigma(z_i) - y_i)x_{ij}$

$\therefore$ the total Gradient is $\frac{\partial L}{\partial \beta_j} = \beta_j + \frac{\lambda}{n}\sum_{i=1}^{n}(\sigma(z_i) - y_i)x_{ij}$

the update rule is: $\beta_j^{(k)} = \beta_j^{(k-1)} - \eta(\beta_j + \frac{\lambda}{n}\sum_{i=1}^{n}(\sigma(z_i) - y_i)x_{ij})$

(f)The answer is in following picture:

(f). The vectorized form of non-intercept term:

weight parameter β and characteristic matrix X represented as vector. $\beta = [\beta_1, \beta_2, \dots \beta_p]^T$ $X = [x_1, x_2, \dots x_n]^T \subset R^{n \times p}$.

the gradient is $\nabla_\beta L = \beta + \frac{\lambda}{n} X^T (\sigma(X\beta + \beta_0) - y)$

$\therefore$ update rule is $\beta^{(k)} = \beta^{(k-1)} - \eta(\beta^{(k-1)} + \frac{\lambda}{n} X^T(\sigma(X\beta + \beta_0) - y))$

Vectorized Update of Merged Intercept Terms:

let $\gamma = [\beta_0, \beta^T]^T \in R^{p+1}$, then the extended feature matrix is $\tilde{X} = [1, X] \in R^{n \times (p+1)}$

$\therefore$ the gradient of loss function is $\nabla_\gamma L = \begin{bmatrix} \frac{\lambda}{n} 1^T (\sigma(\tilde{X}\gamma) - y) \\ \beta + \frac{\lambda}{n} X^T (\sigma(\tilde{X}\gamma) - y) \end{bmatrix}$

the update rule is: $\gamma^{(k)} = \gamma^{(k-1)} - \eta \begin{bmatrix} \frac{\lambda}{n} 1^T (\sigma(\tilde{X}\gamma^{(k-1)}) - y) \\ \beta^{(k-1)} + \frac{\lambda}{n} X^T (\sigma(\tilde{X}\gamma^{(k-1)}) - y) \end{bmatrix}$