Logistic Regression

Mathematical Foundations of Data Science

13 MAY 2018

WORKAROUNDS - https://github.com/abinthomasonline/CH5019

JUPYTER NOTEBOOK - https://bit.ly/2IxLxd9

SUBMITTED BY

ABIN THOMAS - NA15B001

MICHAEL THOMAS - CH15B038

Question 1

Write your own code to fit a logistic regression model to the data set described below in a pro- gramming language of your choice. (**IMPORTANT: DO NOT USE ANY IN-BUILT LIBRARIES**)

Description of Data Set 1:

This data set describes the operating conditions of a reactor and contains class labels about whether the reactor will operate or fail under those operating conditions. Your job is to construct a logistic regression model to predict the same.

- **q1_data_matrix.csv**: This file contains a 1000 × 5 data matrix. The 5 features are the operating conditions of the reactor; their corresponding ranges are described below:

  1. **Temperature:** 400-700 K
  2. **Pressure:** 1-50 bar
  3. **Feed Flow Rate:** 50-200 kmol/hr
  4. **Coolant Flow Rate:** 1000-3600 L/hr
  5. **Inlet Reactant Concentration:** 0.1-0.5 mol fraction

- **q1_labels.csv**: This file contains a 1000 × 1 vector of 0/1 labels for whether the reactor will operate or fail under the corresponding operating conditions.

  - 0: The reactor will operate well under the operating conditions
  - 1: The reactor fails under the operating conditions

Some General Guidelines:

1. Partition your data into a training set and a test set. Keep **70%** of your data for **training** and set aside the remaining **30%** for **testing.**
2. Fit a logistic regression model on the training set. Choose an appropriate objective function to quantify classification error. **Manually code for the gradient descent procedure** used to find optimum model parameters. (**Note:** You may need to perform multiple initializations to avoid local minima)
3. Evaluate the performance of above model on your test data. Report the **confusion matrix** and the F1 **Score**.

Solution

Import Libraries

In [1]:
```
import numpy as np
import pandas as pd
```

Load data

Use `pandas.read_csv` method to load data and label the columns accordingly.

In [2]:
```
features = pd.read_csv('../data/q1_data_matrix.csv', header=None, names=['temp', 'press', 'ffr', 'cfr', 'irc'])
labels = pd.read_csv('../data/q1_labels.csv', header=None, names=['oper'])
```

Data Exploration

Have a look on the first few rows using `pandas.DataFrame.head` method.

In [3]:
```
features.head()
```

Out[3]:

	temp	press	ffr	cfr	irc
0	406.86	17.66	121.83	2109.20	0.1033
1	693.39	24.66	133.18	3138.96	0.3785
2	523.10	23.23	146.55	1058.24	0.4799
3	612.86	40.97	94.44	1325.12	0.3147
4	500.28	37.44	185.48	2474.51	0.2284

In [4]:
```
labels.head()
```
Out[4]:

	oper
0	0.0
1	0.0
2	1.0
3	1.0
4	0.0

All features are real numbers so no encoding required.
Get the number of samples using `pandas.DataFrame.shape` method.

In [5]:
```
features.shape[0]
```
Out[5]:
```
1000
```

As the number is just 1000 no need to use batch or stochastic methods for gradient descent.
Use `pandas.DataFrame.isnull` & `pandas.DataFrame.sum` together to get the count of rows with null values.

In [6]:
```
features.isnull().sum()
```
Out[6]:
```
temp     0
press    0
ffr      0
cfr      0
irc      0
dtype: int64
```

There is no null values in the data. No need to drop any rows.
Find the distribution of samples across two classes using `pandas.Series.value_counts`.

Split Data

Split data into train and test, 70% and 30% respectively using `numpy.split`.

In [7]:
```
[train_X, test_X] = np.split(features, [int(0.7*features.shape[0])], axis=0)
[train_Y, test_Y] = np.split(labels, [int(0.7*labels.shape[0])], axis=0)
```

Check distribution of samples across the classes in the training set.

In [8]:
```
train_Y["oper"].value_counts()
```
Out[8]:
```
0.0    398
1.0    302
Name: oper, dtype: int64
```

Both the classes are sufficiently represented.

Feature Scaling

Subtract every feature by corresponding mean and divide by corresponding standard deviation. Use `pandas.DataFrame.mean` and `pandas.DataFrame.std` for mean and standard deviation respectively.

In [9]:
```
X = (train_X-train_X.mean())/train_X.std()
X.head()
```

Out[9]:

	temp	press	ffr	cfr	irc
0	-1.598642	-0.534905	-0.063694	-0.207417	-1.705472
1	1.685504	-0.044677	0.200960	1.142378	0.689765
2	-0.266323	-0.144823	0.512715	-1.585001	1.572312
3	0.762487	1.097555	-0.702361	-1.235179	0.134473
4	-0.527881	0.850340	1.420466	0.271426	-0.616649

Bias Term

Add a column of all ones to train_X.

In [10]:
```
X.insert(0, 'bias', 1)
X.head()
```

Out[10]:

	bias	temp	press	ffr	cfr	irc
0	1	-1.598642	-0.534905	-0.063694	-0.207417	-1.705472
1	1	1.685504	-0.044677	0.200960	1.142378	0.689765
2	1	-0.266323	-0.144823	0.512715	-1.585001	1.572312
3	1	0.762487	1.097555	-0.702361	-1.235179	0.134473
4	1	-0.527881	0.850340	1.420466	0.271426	-0.616649

Sigmoid

Define a function `sigmoid` that computes sigmoid of input values.

In [11]:
```
def sigmoid(z):
    return 1/(1+np.exp(-z))
sigmoid(0)
```

Out[11]: 0.5

Loss

Define a function `cost` that computes loss given prediction and labels.

In [12]:
```
def cost(probability, label):
    return (-label*np.log(probability)-(1-label)*np.log(1-probability)).mean()
```

Gradient

Define `gradient` function that computes gradient with respect to each parameter given features, labels and parameters.

In [13]:
```
def gradient(features, parameters, labels):
    h = sigmoid(np.dot(features, parameters))
    return np.dot(features.transpose(), h-labels)/labels.shape[0]
```

Gradient Descent

`gradient_descent` function that performs iterations of gradient descent and returns parameters given features, labels, learning rate and number of iterations.

In [14]:
```
def gradient_descent(features, labels, learning_rate, number_of_iterations):
    parameters = np.random.normal(0, 1, features.shape[1])
    for i in range(number_of_iterations):
        grad = gradient(features, parameters, labels)
        parameters -= learning_rate*grad
        '''print(i+1, cost(sigmoid(np.dot(features, parameters)), labels), grad)'''
    return parameters
```

Predict

`predict` function that returns the predicted labels given features, parameters and threshold.

In [15]:
```
def predict(features, parameters, threshold):
    return sigmoid(np.dot(features, parameters)) >= threshold
```

Misclassification Error

`misclassification_error` function returns misclassification error.

In [16]:
```
def misclassification_error(predictions, labels):
    return (predictions!=labels).mean()
```

Driver Loop

Do gradient descent to have an idea of the number of iterations at which cost from test set starts going up. Decrease learning rate gradually to avoid local minima and improve learning speed.

In [17]:
```
Y = train_Y.iloc[:, 0]
X_t = (test_X-test_X.mean())/test_X.std()
X_t.insert(0, 'bias', 1)
Y_t = test_Y.iloc[:, 0]

theta = np.zeros(6)
theta_i = np.zeros(6)
alpha = 0.1
test_cost = 1000
i=1
while 1:
    theta_i -= alpha*gradient(X, theta, Y)
    train_cost = cost(sigmoid(np.dot(X, theta_i)), Y)
    temp = cost(sigmoid(np.dot(X_t, theta_i)), Y_t)
    if temp>test_cost:
        if alpha>0.000001:
            alpha/=10
            continue
        else:
            break
    test_cost=temp
    theta = theta_i
    print(i, train_cost, test_cost)
    i+=1
```

```
1632 0.2607241751931474 0.28092426348015925
1633 0.2607238170484276 0.2809242528705977
1634 0.26072346004264413 0.28092424282368694
1635 0.26072310417203287 0.2809242333367292
1636 0.2607227494328424 0.2809242244070373
1637 0.26072239582133483 0.28092421603193496
1638 0.2607220433337847 0.28092420820875647
1639 0.2607216919664801 0.2809242009348467
1640 0.26072134171572175 0.2809241942075615
1641 0.2607209925778233 0.28092418802426683
```

In [18]:
```python
print("Train error : ", misclassification_error(predict(X, theta, 0.5), Y))
print("Test error : ", misclassification_error(predict(X_t, theta, 0.5), Y_t))
print(theta)
```

```
Train error :  0.05142857142857143
Test error :  0.09
[-1.01870462  0.1953828   0.66090329  0.63696202 -3.55495908  0.09227634]
```

Perform iterations of `gradient_descent` with multiple initialization and print misclassification error.

In [19]:
```python
for i in range(20):
    theta_i = gradient_descent(X, Y, 0.01, 1600)
    mc_error_train = misclassification_error(predict(X, theta_i, 0.5), Y)
    mc_error_test = misclassification_error(predict(X_t, theta_i, 0.5), Y_t)
    print(i+1, round(mc_error_train, 4), round(mc_error_test, 4), theta_i)
```

```
1 0.0657 0.1033 [-5.03095568e-01 -1.70836523e-04  4.13488166e-01  4.74309265e-01
 -2.28682538e+00  7.13150376e-02]
2 0.0557 0.1067 [-0.50793752  0.15955294  0.38612939  0.2182976  -2.01009079  0.02718853]
3 0.0714 0.0967 [-0.73935068  0.25043545  0.82796664  0.7904199  -2.77355241  0.2764856 ]
4 0.0543 0.1133 [-0.44161326  0.16144658  0.33502178  0.29459462 -1.93068409  0.05319208]
5 0.0686 0.12 [-0.39112786  0.16357119  0.38356913  0.32158721 -1.9926219  -0.01844572]
6 0.06 0.1 [-0.52217244  0.16915082  0.28612531  0.28363751 -2.12382762 -0.07722074]
7 0.0986 0.1067 [-0.62453088  0.16510281  0.12756448  0.01144191 -2.32428645 -0.23065443]
8 0.06 0.1133 [-0.48234724  0.1757505   0.44621613  0.30844267 -1.96755287  0.02011711]
9 0.0771 0.11 [-0.43312925  0.06651961  0.26355579  0.43734356 -2.20894872  0.04208882]
10 0.0586 0.1167 [-0.48749015  0.14104629  0.43446645  0.27731126 -1.9194208   0.05177214]
11 0.0671 0.1067 [-0.47859503  0.01657322  0.46685402  0.37200293 -2.22441285 -0.00238295]
12 0.0671 0.1067 [-0.38397004  0.08867232  0.30797557  0.29898019 -1.89682828  0.07706732]
13 0.06 0.1 [-0.48244766  0.01191927  0.39897224  0.28682776 -2.18333109  0.07473783]
14 0.0686 0.1033 [-0.45565164  0.11617079  0.25989494  0.35206494 -2.12013462  0.00776301]
15 0.0771 0.1033 [-0.52386375 -0.09849776  0.35240341  0.27426714 -2.4493133  -0.01554331]
16 0.0514 0.0933 [-0.56893082  0.08973322  0.34965325  0.3992079  -2.13750049  0.04141368]
17 0.0614 0.11 [-4.57267220e-01 -1.15881266e-03  3.65244498e-01  2.37134707e-01
 -2.00484994e+00  8.26189217e-02]
18 0.0657 0.1167 [-0.36943317  0.06592287  0.37013719  0.29248337 -2.02331092  0.04618403]
19 0.0786 0.12 [-0.40130605 -0.00530211  0.23703365  0.43881198 -2.61162416  0.04134634]
20 0.0686 0.1 [-0.4205923   0.16365063  0.19949386  0.28583786 -1.83260377 -0.05965412]
```

Choose theta with best performance.

Confusion Matrix

`confusion_matrix` function which returns confusion matrix given predicted labels and original labels.

In [20]:
```python
def confusion_matrix(predictions, labels):
    true_positives = np.logical_and(predictions, labels).sum()
    false_positives = np.logical_and(predictions, np.logical_not(labels)).sum()
    false_negatives = np.logical_and(np.logical_not(predictions), labels).sum()
    true_negatives = np.logical_not(np.logical_or(predictions, labels)).sum()
    return pd.DataFrame(data={'Predicted 0': [true_negatives, false_negatives], 'Predicted 1': [false_positives, true_
positives]}, index=['Actual 0', 'Actual 1'])

X = (features-features.mean())/features.std()
X.insert(0, 'bias', 1)
Y = labels.iloc[:, 0]
CF = confusion_matrix(predict(X, theta, 0.5), Y)
CF
```

Out[20]:

	Predicted 0	Predicted 1
Actual 0	550	35
Actual 1	22	393

F1 Score

`f1_score` function calculates F1 score given confusion matrix.

In [21]:
```python
def f1_score(confusion_matrix):
    return (2*confusion_matrix["Predicted 1"]["Actual 1"])/(2*confusion_matrix["Predicted 1"]["Actual 1"]+confusion_ma
trix["Predicted 0"]["Actual 1"]+confusion_matrix["Predicted 1"]["Actual 0"])

f1_score(CF)
```

Out[21]: 0.9323843416370107

Question 2

Use the same code developed in Question 1 to fit a logistic regression model to the dataset described below.

Description of Data Set 2:

This data set contains data for credit card fraud detection.

- **q2_data_matrix.csv:** This file contains a 100 × 5 data matrix. The 5 features and their corresponding ranges are described below:
    1. **Age:** 18-100 years
    2. **Transaction Amount:** $ 0-5000
    3. **Total Monthly Transactions:** $ 0-50000
    4. **Annual Income:** $ 30000-1000000
    5. **Gender:** 0/1 (0 - Male, 1 - Female)
- **q2_labels.csv:** This file contains a 1000 × 1 vector of 0/1 labels for whether the transaction is fraudulent or not.
    - 0: The transaction is legitimate
    - 1: The transaction is fraudulent

1. Report the confusion matrix and the F1 Score for this data set.
2. Which data set gives better results better? Can you think of reasons as to why one data set gives better results than the other? (**Hint**: Think of assumptions behind the logistic regression model)
3. Can you suggest improvements to the logistic regression model to make it perform better on the unfavorable data set?
4. **Bonus Points!**: Implement your suggested improvement as a code and compare the performance of this with vanilla logistic regression.

Solution

Load Data

In [22]:
```
features = pd.read_csv('../data/q2_data_matrix.csv', header=None, names=['age', 'tram', 'tomotr', 'anin', 'gen'])
labels = pd.read_csv('../data/q2_labels.csv', header=None, names=['fra'])
```

Data Exploration

In [23]:
```
features.head()
```

Out[23]:

	age	tram	tomotr	anin	gen
0	31.0	2897.0	49741.0	339500.0	1.0
1	46.0	2087.0	23953.0	935000.0	1.0
2	23.0	1814.0	26056.0	191700.0	0.0
3	94.0	179.0	30250.0	715900.0	0.0
4	26.0	3995.0	39466.0	711900.0	0.0

In [24]:
```
labels.head()
```

Out[24]:

	fra
0	1.0
1	0.0
2	1.0
3	0.0
4	0.0

In [25]:
```
features.shape[0]
```

Out[25]:
```
1000
```

In [26]:
```
features.isnull().sum()
```

Out[26]:
```
age       0
tram      0
tomotr    0
anin      0
gen       0
dtype: int64
```

In [27]:
```
pd.concat([features, labels["fra"]], axis=1).groupby("fra").mean()
```

Out[27]:

	age	tram	tomotr	anin	gen
fra					
0.0	57.091483	2418.894322	22141.055205	643522.082019	0.479495
1.0	62.789617	2940.267760	30990.527322	311790.163934	0.543716

Split Data

In [28]:
```
[train_X, test_X] = np.split(features, [int(0.7*features.shape[0])], axis=0)
[train_Y, test_Y] = np.split(labels, [int(0.7*labels.shape[0])], axis=0)
```

Check distribution of data samples across classes.

In [29]:
```
train_Y["fra"].value_counts()
```

Out[29]:
```
0.0    438
1.0    262
Name: fra, dtype: int64
```

Data distribution is around 60%-40%.

Feature Scaling

In [30]:
```
X = (train_X-train_X.mean())/train_X.std()
X.head()
```

Out[30]:

	age	tram	tomotr	anin	gen
0	-1.212844	0.201788	1.676857	-0.634242	0.979496
1	-0.570286	-0.360923	-0.111253	1.534665	0.979496
2	-1.555542	-0.550577	0.034567	-1.172553	-1.019475
3	1.485901	-1.686419	0.325374	0.736668	-1.019475
4	-1.427031	0.964573	0.964401	0.722099	-1.019475

Bias Term

In [31]:
```
X.insert(0, 'bias', 1)
X.head()
```

Out[31]:

	bias	age	tram	tomotr	anin	gen
0	1	-1.212844	0.201788	1.676857	-0.634242	0.979496
1	1	-0.570286	-0.360923	-0.111253	1.534665	0.979496
2	1	-1.555542	-0.550577	0.034567	-1.172553	-1.019475
3	1	1.485901	-1.686419	0.325374	0.736668	-1.019475
4	1	-1.427031	0.964573	0.964401	0.722099	-1.019475

Driver Loop

In [32]:
```
Y = train_Y.iloc[:, 0]
X_t = (test_X-test_X.mean())/test_X.std()
X_t.insert(0, 'bias', 1)
Y_t = test_Y.iloc[:, 0]

theta = np.zeros(6)
theta_i = np.zeros(6)
alpha = 0.1
test_cost = 1000
i=1
while 1:
    theta_i -= alpha*gradient(X, theta, Y)
    train_cost = cost(sigmoid(np.dot(X, theta_i)), Y)
    temp = cost(sigmoid(np.dot(X_t, theta_i)), Y_t)
    if temp>test_cost:
        if alpha>0.000001:
            alpha/=10
            continue
        else:
            break
    test_cost = temp
    theta = theta_i
    print(i, train_cost, test_cost)
    i+=1
```

```
10433 0.32503496086388933 0.3374106766366127
10434 0.3250349608638893 0.3374106766366127
10435 0.3250349608638892 0.3374106766366126
10436 0.32503496086388933 0.3374106766366124
10437 0.3250349608638893 0.3374106766366123
10438 0.3250349608638892 0.33741067663661206
10439 0.3250349608638893 0.3374106766366119
10440 0.3250349608638892 0.3374106766366118
10441 0.32503496086388933 0.3374106766366118
10442 0.3250349608638892 0.3374106766366117
10443 0.3250349608638892 0.3374106766366115
10444 0.3250349608638893 0.33741067663661134
10445 0.3250349608638892 0.33741067663661123
10446 0.3250349608638892 0.337410676636611
```

In [33]:
```
print("Train error : ", misclassification_error(predict(X, theta, 0.5), Y))
print("Test error : ", misclassification_error(predict(X_t, theta, 0.5), Y_t))
print(theta)
```

```
Train error :  0.1
Test error :  0.13
[-1.19057535  0.54762267  0.87412537  1.45826855 -2.45583991  0.12145785]
```

In [34]:
```
for i in range(20):
    theta_i = gradient_descent(X, Y, 0.01, 1000)
    mc_error_train = misclassification_error(predict(X, theta_i, 0.5), Y)
    mc_error_test = misclassification_error(predict(X_t, theta_i, 0.5), Y_t)
    print(i+1, round(mc_error_train, 4), round(mc_error_test, 4), theta_i)
```

```
1 0.1443 0.1267 [-0.80333384  0.30825477  0.27741166  0.73786549 -1.07156181 -0.02181775]
2 0.1143 0.1433 [-0.45982168  0.09345891  0.42228359  0.7881262  -1.31119965  0.08785956]
3 0.1329 0.15 [-0.29010296  0.21698243  0.23755198  0.93149722 -1.37764829  0.16540425]
4 0.1343 0.1267 [-0.83064526  0.07637762  0.37376378  0.95240723 -1.44414594 -0.15371178]
5 0.1414 0.13 [-0.95871433  0.65056088  0.26108463  1.12996367 -1.35267248  0.27247142]
6 0.1771 0.2133 [-0.70488941  0.65644252  1.24130969  1.17727218 -1.44424639  0.06864588]
7 0.1014 0.1033 [-0.60550231  0.18731487  0.25842826  0.6140183  -1.0639407  -0.04296883]
8 0.1343 0.1533 [-0.48325583  0.1011045   0.50227935  0.792766   -0.98406597  0.02987987]
9 0.1286 0.14 [-0.92440445  0.11582874  0.56008637  0.73457553 -1.58040824  0.25429075]
10 0.1286 0.1533 [-0.26472031  0.26068041  0.12954925  0.60965837 -0.99593543  0.27094449]
11 0.1229 0.1467 [-0.7212136   0.31425471  0.74593445  1.00714716 -1.63784677  0.0673217 ]
12 0.17 0.1833 [-1.1460129   0.92113154 -0.23181316  0.8679345  -1.85512561  0.78895311]
13 0.1357 0.1267 [-0.49686059  0.18045228  0.01858988  0.79172053 -1.01388701  0.13475207]
14 0.1343 0.1733 [-0.72801025  0.01210963  0.69415785  0.70146551 -1.36409794 -0.10412912]
15 0.1486 0.1667 [-1.43487692  0.58135254  0.95137246  1.05661641 -2.08565155 -0.67582128]
16 0.1157 0.14 [-1.15754051  0.77776456  0.62637834  0.9397739  -1.85715543  0.12604622]
17 0.1457 0.1733 [-1.07091286  0.4534306   0.9788532   0.66063116 -1.69308076  0.34908064]
18 0.1229 0.1533 [-0.86019305  0.0634375   0.66607954  0.65119064 -1.9813613   0.10771826]
19 0.21 0.19 [-0.94335564 -0.18789854 -0.05548611  0.75361015 -1.11273978  0.25070873]
20 0.13 0.1233 [-0.71981427  0.11143276  0.05148688  0.68826428 -1.36013398  0.09707957]
```

Choose theta with least missclassification error.

Confusion Matrix

In [35]:
```
X = (features-features.mean())/features.std()
X.insert(0, 'bias', 1)
Y = labels.iloc[:, 0]
CF = confusion_matrix(predict(X, theta, 0.5), Y)
CF
```

Out[35]:

	Predicted 0	Predicted 1
Actual 0	583	51
Actual 1	56	310

F1 Score

In [36]:
```
f1_score(CF)
```

Out[36]:
```
0.8528198074277854
```

Inference

The first dataset performs better using this model.

Logistic regression can only classify linearly separable data. This might be the reason for poor performance of second data set. Data sample imbalance across classes also might be a reason.

Improvements

One possible solution is by expanding the basis to include higher order terms of features into dataset. For further improvements remove features that seems to be independent by trial and error to get an optimum set of features.

Bonus

Define function `basis_expansion` to introduce second order terms into features set.

In [37]:
```
def basis_expansion(X):
    for i in range(0, 5):
        for j in range(i, 5):
            X[str(i)+str(j)]=X.iloc[:, i]*X.iloc[:, j]
    return X
```

In [38]:
```
features = basis_expansion(features)
```

In [39]:
```
[train_X, test_X] = np.split(features, [int(0.7*features.shape[0])], axis=0)
[train_Y, test_Y] = np.split(labels, [int(0.7*labels.shape[0])], axis=0)

X = (train_X-train_X.mean())/train_X.std()
X.insert(0, 'bias', 1)
Y = train_Y.iloc[:, 0]
X_t = (test_X-test_X.mean())/test_X.std()
X_t.insert(0, 'bias', 1)
Y_t = test_Y.iloc[:, 0]

theta = np.zeros(21)
theta_i = np.zeros(21)
alpha = 0.1
test_cost = 1000
i=1
while 1:
    theta_i -= alpha*gradient(X, theta, Y)
    train_cost = cost(sigmoid(np.dot(X, theta_i)), Y)
    temp = cost(sigmoid(np.dot(X_t, theta_i)), Y_t)
    if temp>test_cost:
        if alpha>0.000001:
            alpha/=10
            continue
        else:
            break
    test_cost = temp
    theta = theta_i
    print(i, train_cost, test_cost)
    i+=1
```

```
2172 0.31701659943561133 0.33049671719211754
2173 0.3170137281318014 0.33049671378561624
2174 0.317010858048474 0.33049671066126163
2175 0.3170079891848145 0.33049670781857965
2176 0.3170051215400083 0.3304967052570955
2177 0.3170022551132422 0.3304967029763375
2178 0.3169993899037031 0.33049670097583295
2179 0.3169965259105795 0.3304966992551112
2180 0.3169936631330602 0.33049669781370267
2181 0.3169908015703349 0.330496696651138
2182 0.3169879412215944 0.33049669576695007
```

In [40]:
```
print("Train error : ", misclassification_error(predict(X, theta, 0.5), Y))
print("Test error : ", misclassification_error(predict(X_t, theta, 0.5), Y_t))
print(theta)
```

```
Train error :  0.09857142857142857
Test error :  0.13333333333333333
[-0.94683286  0.41616011  0.45021329  1.47993344 -0.74432487  0.12890708
 -0.09742759  0.12086925  0.48203783 -0.32741465  0.03540156  0.21571053
  0.41958038 -0.51383393  0.22243022 -0.09342475 -1.13939958 -0.06235577
 -0.13042047 -0.43062858  0.12890708]
```

In [41]:
```
X = (features-features.mean())/features.std()
X.insert(0, 'bias', 1)
Y = labels.iloc[:, 0]
CF = confusion_matrix(predict(X, theta, 0.5), Y)
CF
```

Out[41]:

	Predicted 0	Predicted 1
Actual 0	591	43
Actual 1	66	300

In [42]:
```
f1_score(CF)
```

Out[42]: 0.846262341325811

Feature Selection

Drop features that are independent or irrelevant by trial and error.

In [43]:
```
features = features.drop(columns=['44', '02', '04', '13', '14', '24', '34'])
features = features.drop(columns=['33', '22', '11', '12', '23'])
'''features = features.drop(columns=['03'])'''
features.head()
```

Out[43]:

	age	tram	tomotr	anin	gen	00	01	03
0	31.0	2897.0	49741.0	339500.0	1.0	961.0	89807.0	10524500.0
1	46.0	2087.0	23953.0	935000.0	1.0	2116.0	96002.0	43010000.0
2	23.0	1814.0	26056.0	191700.0	0.0	529.0	41722.0	4409100.0
3	94.0	179.0	30250.0	715900.0	0.0	8836.0	16826.0	67294600.0
4	26.0	3995.0	39466.0	711900.0	0.0	676.0	103870.0	18509400.0

In [44]:
```
[train_X, test_X] = np.split(features, [int(0.7*features.shape[0])], axis=0)
[train_Y, test_Y] = np.split(labels, [int(0.7*labels.shape[0])], axis=0)

X = (train_X-train_X.mean())/train_X.std()
X.insert(0, 'bias', 1)
Y = train_Y.iloc[:, 0]
X_t = (test_X-test_X.mean())/test_X.std()
X_t.insert(0, 'bias', 1)
Y_t = test_Y.iloc[:, 0]

theta = np.zeros(9)
theta_i = np.zeros(9)
alpha = 0.1
test_cost = 1000
i=1
while 1:
    theta_i -= alpha*gradient(X, theta, Y)
    train_cost = cost(sigmoid(np.dot(X, theta_i)), Y)
    temp = cost(sigmoid(np.dot(X_t, theta_i)), Y_t)
    if temp>test_cost:
        if alpha>0.000001:
            alpha/=10
            continue
        else:
            break
    theta = theta_i
    test_cost = temp
    print(i, train_cost, test_cost)
    i+=1
```

```
1773 0.3266623137173051 0.3375566504135404
1774 0.3266610703488829 0.337556641306663
1775 0.3266598276220403 0.3375566328860763
1776 0.3266585855363516 0.33755662515060625
1777 0.3266573440913919 0.337556618090825
1778 0.32665610328673683 0.3375566117303371
1779 0.32665486312196235 0.3375566060432051
1780 0.32665362359664524 0.33755660103652485
1781 0.32665238471036273 0.3375565967091372
1782 0.3266511464626926 0.3375565930598866
1783 0.32664990885321316 0.3375565900876198
```

In [45]:
```python
print("Train error : ", misclassification_error(predict(X, theta, 0.5), Y))
print("Test error : ", misclassification_error(predict(X_t, theta, 0.5), Y_t))
print(theta)
```

```
Train error :  0.09857142857142857
Test error :  0.12666666666666668
[-1.15077492  0.56165362  0.70445155  1.42492256 -1.94506542  0.12742219
  0.13461665  0.19359139 -0.60572495]
```

In [46]:
```python
X = (features-features.mean())/features.std()
X.insert(0, 'bias', 1)
Y = labels.iloc[:, 0]
CF = confusion_matrix(predict(X, theta, 0.5), Y)
CF
```

Out[46]:

	Predicted 0	Predicted 1
Actual 0	583	51
Actual 1	55	311

In [47]:
```python
f1_score(CF)
```

Out[47]: 0.8543956043956044