**INTRODUCTION**

**A.1. Objective**

The purpose of this application is to use the methods of classification in machine learning and predict coronary heart disease in males. This application is conducted using a dataset which describes several conditions these males have, this dataset comes from a heart-disease high-risk region of Western Cape, South Africa. We will apply Logistic Regression to predict coronary disease in these patients.

**IMPLEMENTATION**

**B. Exploratory Data Analysis**

**B.1. Percent of Class 0 and 1**

```
p = df1["chd"].value_counts()
print("Patients who don't have it: " + str(p[0]))
print("Patients who have it: " + str(p[1]))
t = (p[0] + p[1])
print("Total Patients")
per1 = ((p[1]/t)*100)
per2 = ((p[0]/t)*100)

print('Percent of 0: ' + str(per2) + '%')
print('Percent of 1: ' + str(per1) + '%')
```

```
Patients who don't have it: 302
Patients who have it: 160
Total Patients
Percent of 0: 65.36796536796537%
Percent of 1: 34.63203463203463%
```

In order to find the percentage of each class 0 and 1, we take the values within the “chd” column and store them into p. We use t to place the total amount of both classes, using p[0] and p[1]. We implement the percentage equation for both class 0 and class 1, this gives us the percentage of class 0 and percentage of class 1, as shown in ***Figure B.1***.

**B.2. Missing Values**

```
df1.dtypes
df1.info()   # no missing values
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 462 entries, 0 to 461
Data columns (total 11 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   row.names  462 non-null    int64  
 1   sbp        462 non-null    int64  
 2   tobacco    462 non-null    float64
 3   ldl        462 non-null    float64
 4   adiposity  462 non-null    float64
 5   famhist    462 non-null    object 
 6   typea      462 non-null    int64  
 7   obesity    462 non-null    float64
 8   alcohol    462 non-null    float64
 9   age        462 non-null    int64  
 10  chd        462 non-null    int64  
dtypes: float64(5), int64(5), object(1)
```

In order to check for missing values within the dataset, we can use the info() function. This returns any null columns. As seen above, we can conclude that our dataset has zero missing values.

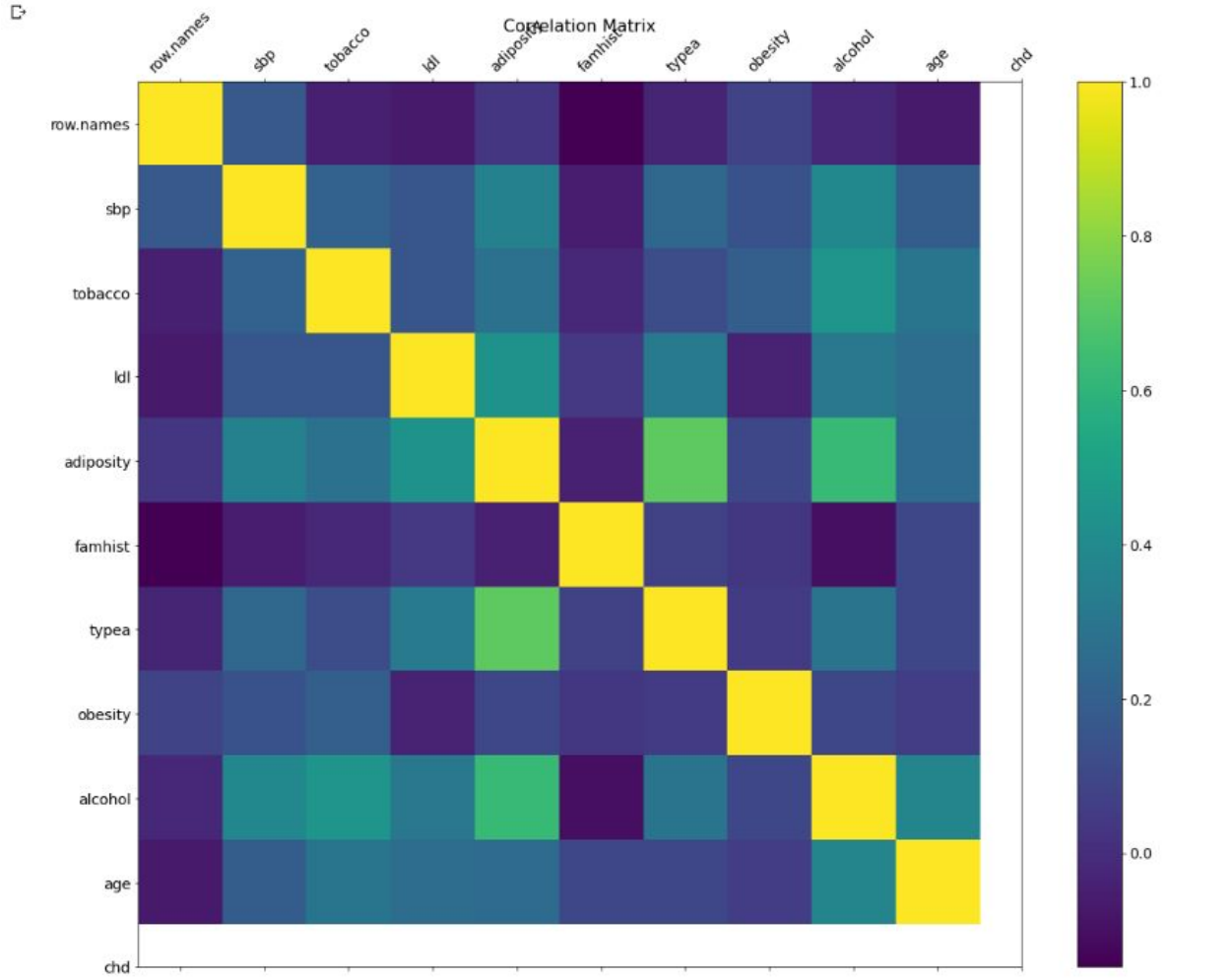
**B.3. Categorical Variables in Features**

Categorical values are non-numerical values in our dataset. We only have two categorical values: Present and Absent within the family history column.

**B.4. Maximum Correlation**

The maximum correlation between each column is shown in the figure below, the color scale displayed in the colorbar legend denotes maximum to minimum correlation between two features using color schemes. Yellow being used for maximum and dark purple for minimum correlation. Features, when compared to themselves, have the most correlation, this is an obvious outcome. But we also have features ‘typea’ and ‘adiposity’ with an estimated correlation value of 80%. Alcohol and adiposity come next with an estimated correlation of 60%. The others share shades of blue and dull green, these have a much lower correlation value than the ones mentioned, ranging below 40%.

```
f = plt.figure(figsize=(19, 15))
plt.matshow(df1.corr(), fignum=f.number)
plt.xticks(range(df1.shape[1]), df1.columns, fontsize=14, rotation=45)
plt.yticks(range(df1.shape[1]), df1.columns, fontsize=14)
cb = plt.colorbar()
cb.ax.tick_params(labelsize=14)
plt.title('Correlation Matrix', fontsize=16);
```




**C. One-Hot-Encoding**

One-Hot-Encoding is used to turn categorical data into numerical, this makes implementing logistic regression easier. Codewise, if we were to keep these values categorical, we would consistently get non-float variable errors. We need to convert these values into a proper format so they can be used successfully.

As mentioned before, our categorical variables come from column ‘famhist’, these values are ‘absent’ and ‘present’. We will use one-hot-encoding to build two columns () in place of this categorical column. We will denote ‘0’ for ‘absent’ and ‘1’ for ‘present’ in the famhist_Present column and vice versa for the famhist_Absent column as shown below. To produce one-hot-encoding, we use the pandas’s function get_dummies().

```
c1 = ['famhist']
cx = df1[c1] # Features
print(cx)

fam_en = pd.get_dummies(df1.famhist, prefix='famhist')
print(fam_en)
```

```
     famhist
0    Present
1     Absent
2    Present
3    Present
4    Present
..       ...
457   Absent
458   Absent
459   Absent
460   Absent
461  Present

[462 rows x 1 columns]
     famhist_Absent  famhist_Present
0                 0                1
1                 1                0
2                 0                1
3                 0                1
4                 0                1
..              ...              ...
457               1                0
458               1                0
459               1                0
460               1                0
461               0                1

[462 rows x 2 columns]
```

**D. Standardization**

In order to make normalizing easy and considering the fact that we have 9 features to normalize, we can use pandas to standardize all 9 features in one line of code, as shown in the figure below.

```
X1 = df_final.iloc[:, 0:9]
y = df1.iloc[:, 10]

X = X1.iloc[:,0:9].apply(lambda x: (x-x.mean())/ x.std(), axis=0)
print(X)
```

```
     famhist_Present       sbp   tobacco  ...     typea   obesity   alcohol
0           1.184570  1.057417  1.821099  ... -0.418017 -0.176594  3.274189
1          -0.842361  0.276789 -0.789382  ...  0.193134  0.670646 -0.612081
2           1.184570 -0.991731 -0.774141  ... -0.112441  0.734723 -0.540597
3           1.184570  1.545310  0.841352  ... -0.214300  1.411091  0.294742
4           1.184570 -0.211103  2.169453  ...  0.702427 -0.012842  1.645991
..               ...       ...       ...  ...       ...       ...       ...
457        -0.842361  3.692037 -0.704470  ...  1.109862  0.570971 -0.696228
458        -0.842361  2.130781  0.122871  ... -0.112441  0.608942  0.068445
459        -0.842361 -1.479624 -0.138395  ... -1.334744 -1.413043  0.391960
460        -0.842361 -0.991731  0.384137  ...  1.109862  0.309916  0.282897
461         1.184570 -0.308682 -0.791559  ...  0.906144 -2.692210 -0.696228

[462 rows x 9 columns]
```

**E. Defining Functions for Logistic Regression**

**E.1. Hypothesis (Model)**

```
def sigmoid(x, theta):
    z= np.dot(x, theta)
    return 1/ (1 + np.exp(-z))# testing the sigmoid function

#sigmoid(X, theta)
```

The hypothesis for a logistic regression model can be denoted using the sigmoid function. The figure above defines this sigmoid function using z as the dot product between x and theta.

**E.2. Cost Function**

```
def costFunction(theta, X, y):
    m=len(y)

    predictions = sigmoid(np.dot(X,theta))
    error = (-y * np.log(predictions)) - ((1-y)*np.log(1-predictions))
    cost = 1/m * sum(error)

    grad = 1/m * np.dot(X.transpose(),(predictions - y))

    return cost[0] , grad
```

The cost function is displayed above. This function takes in a numpy array of theta, x and y and returns the logistic regression cost function and gradient. This function uses Andrew Ng's implementation, [1].

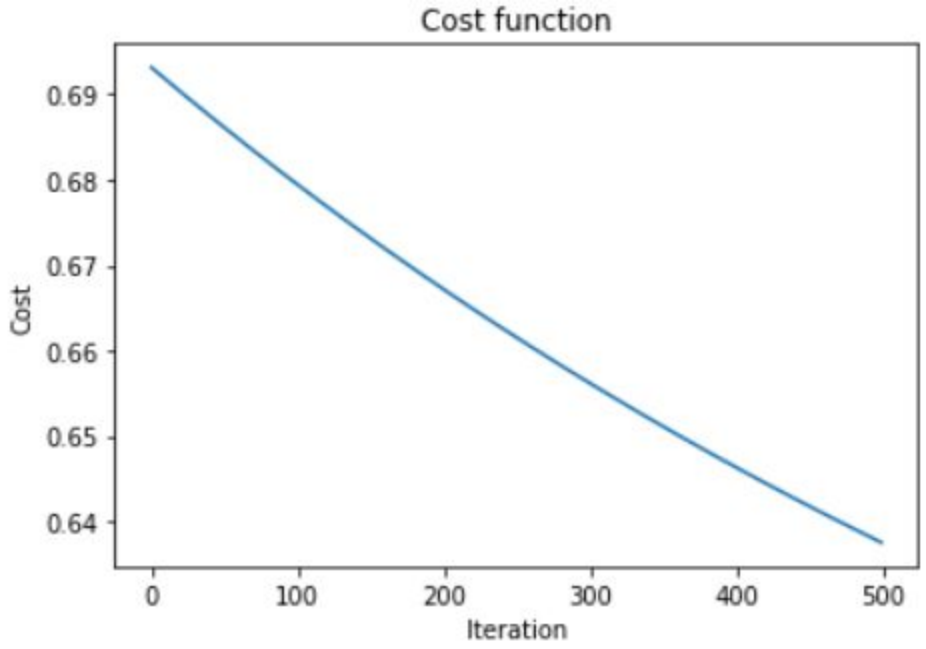
**E.3. Batch Gradient Descent Iteration**

```
def gradientDescent(X,y,theta,alpha,num_iters):

    m=len(y)
    J_history =[]

    for i in range(num_iters):
        cost, grad = costFunction(theta,X,y)
        theta = theta - (alpha * grad)
        J_history.append(cost)

    return theta , J_history
```

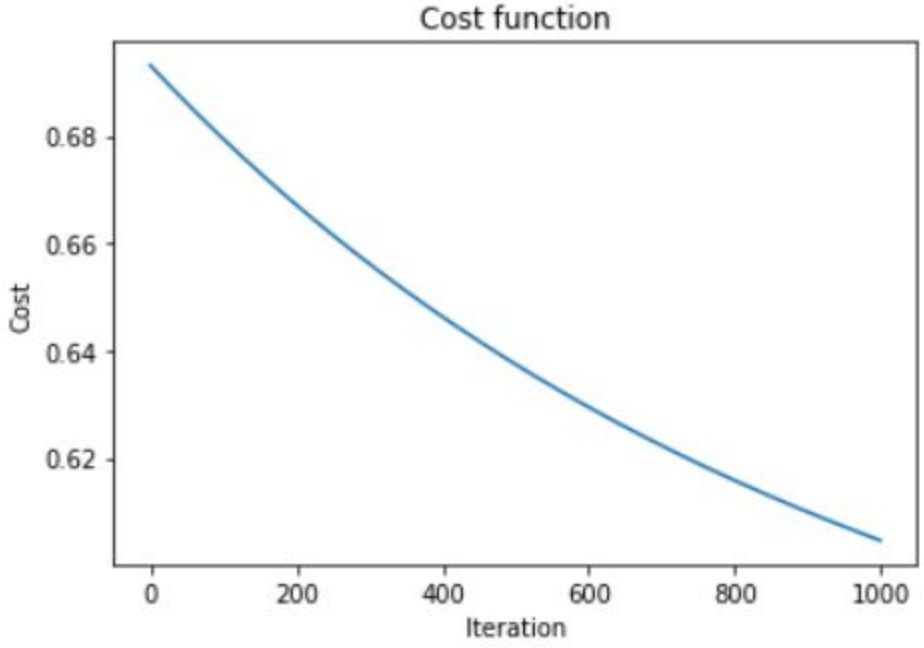
The function above displays the gradient descent function. This function uses Andrew Ng's implementation, [1]. Here, this function takes in a numpy array of theta, x and y and updates theta for each iteration using the updated theta formula. It then returns a list of the updated theta values and cost for each iteration. In order to update to new thetas, we call the gradient function from costFunction. This function outputs the dot product of X with (predictions - y). It's harder to implement this function within the batch gradient descent function itself, seeing as we need to find the transpose of X to find the dot product. Otherwise, if we try a simple summation, it will result in dimensional errors. Our numpy theta arrays' dimensions will not match X.

**F.4. Learning Curves**

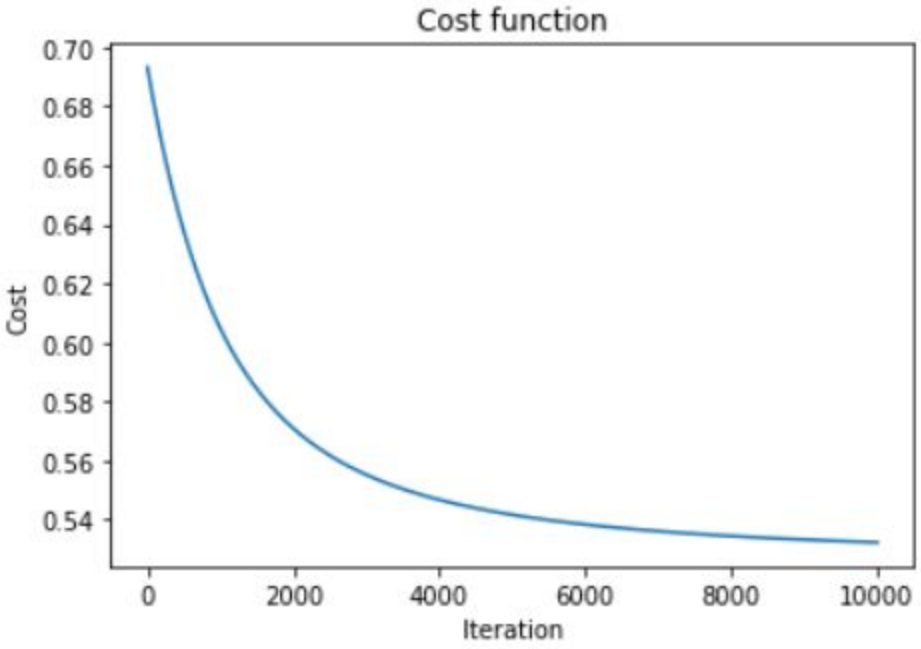
When learning rate = 0.001 and iterations = 500



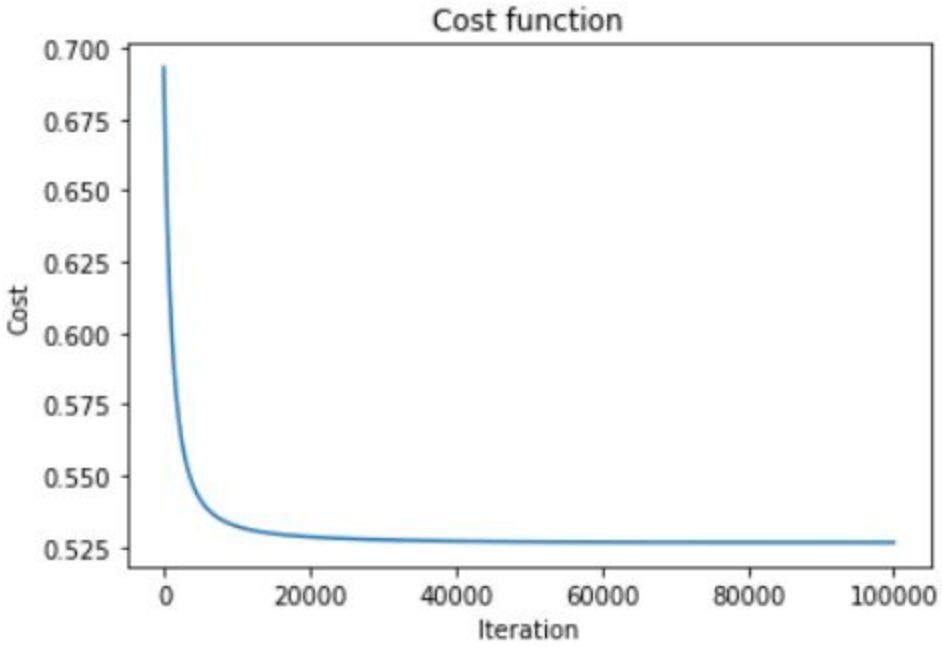

When learning rate = 0.001 and iterations = 1000



When learning rate = 0.001 and iterations = 10,000

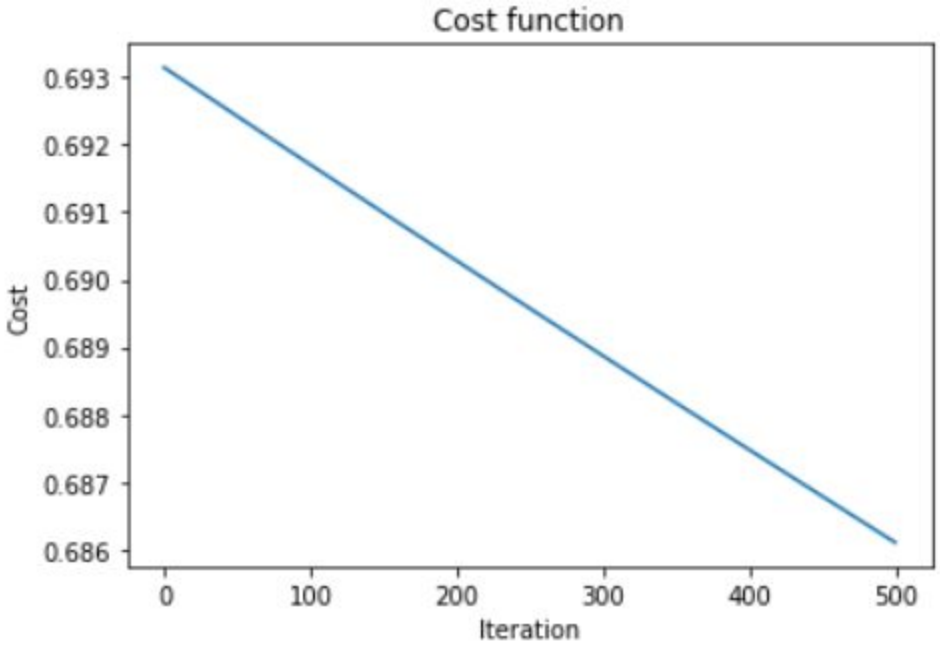



The graphs above display cost vs iteration graphs for learning rate 0.001. A good gradient descent model means decreasing values of cost while iterations increase. When we run our model for 500 iterations, we can see that although our cost decreases, our model does not converge. Similarly with 1000 iterations, there's however a slight curve this time. Our model is slowly reaching towards the minimum value but doesn't quite get there yet. When we run it for 10, 000 iterations, our model shows a bigger curve, we can see it reaching the minimum error more. But it still doesn't fully converge nor does it reach the minimum error rate. Let's try running it for 100K iterations and see if it converges. Below is a graph indicating 100K iterations for learning rate of 0.001.

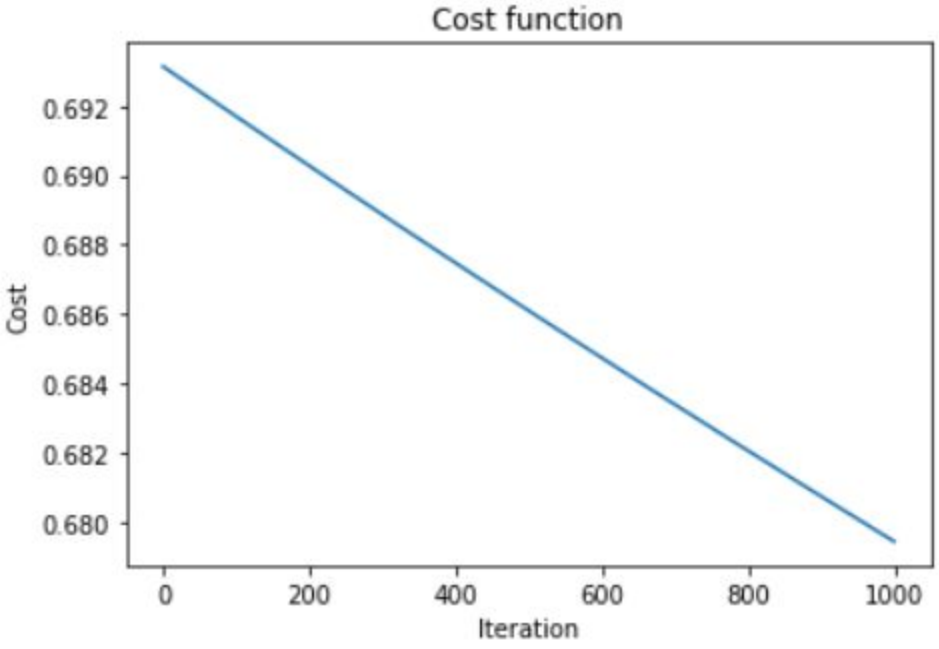



We can see that our model properly converges now, and our minimum error rate is approximately 0.525. When we ran it for 10, 000 iterations, we had an idea that our model would converge properly soon. But we couldn't estimate our minimum error. But compared to 500 iterations and 1000 iterations, 10,000 iterations brings us closer to the minimum error. We then try running the same iterations for 0.0001 value of learning rate, these graphs are shown below.

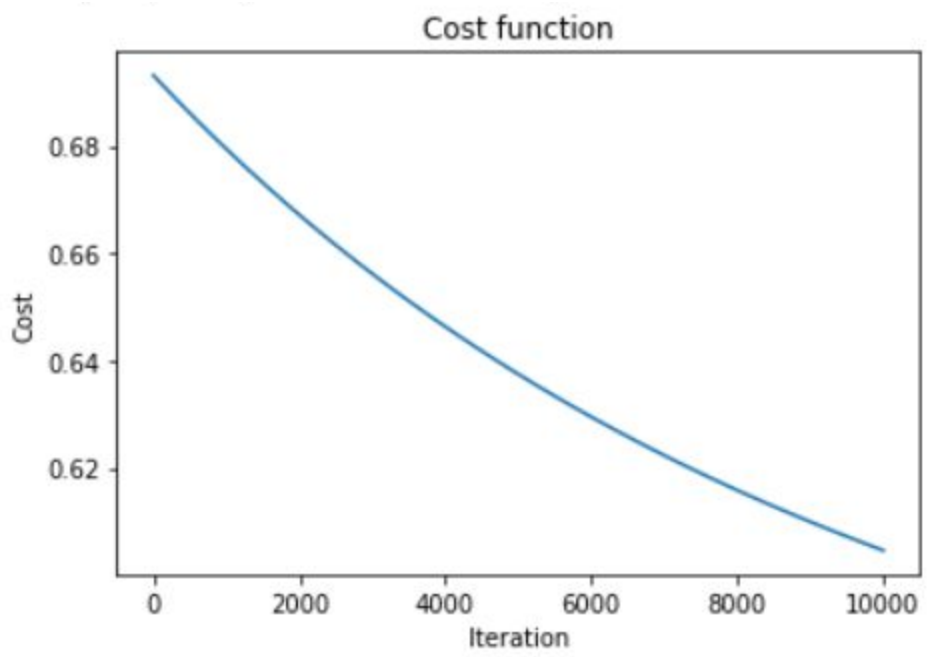
When learning rate = 0.0001 and iterations = 500



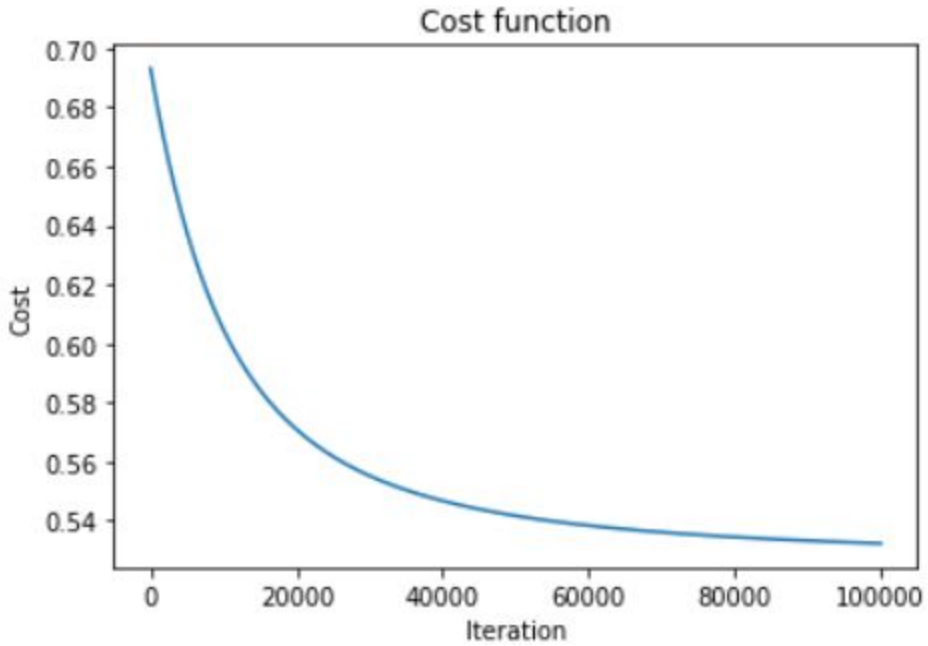

When learning rate = 0.0001 and iterations = 1000




When learning rate = 0.0001 and iterations = 10,000

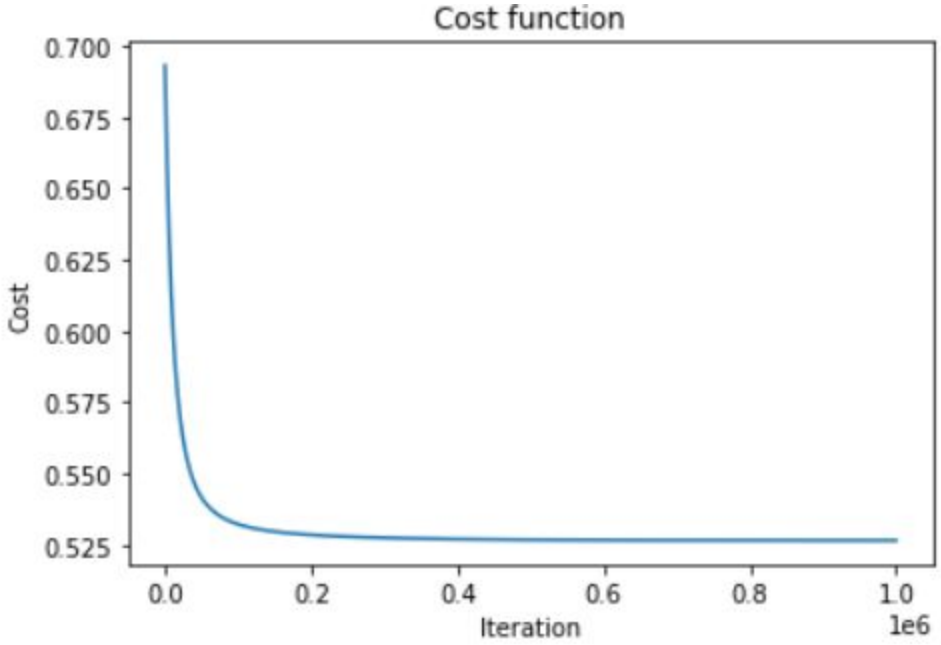



When we use a learning rate of 0.0001 and run it for 500 iterations, we can see that our graph produces a straight decreasing line. There is no curve here, our cost is linearly decreasing by a large amount. It is the same for when we use 1000 iterations. This means we are far from distinguishing our minimum error rate. When we run it for 10,000 iterations, we see a small curve, our cost is still consistently decreasing but it is decreasing by a smaller amount now. We are getting closer but our graph does not converge. So let's try to run it for 100K iterations and see if our graph converges.




The graph above is produced when we run our model for 100K iterations. There's a bigger curve here, and our graph is going towards convergence. We produce a similar graph to when we ran 10,000

iterations for 0.001 learning rate, this is due to the difference between our learning rates. We are taking smaller steps when we use 0.0001 as opposed to using 0.001. So, we can estimate that we will reach convergence 10 times faster when using 0.001 than when using 0.0001. Our graph for 0.0001 learning rate for 10,000 iterations is the same as the graph for learning 0.001 for 1000 iterations, there is a difference of 10. To further explain this, let's run our learning rate of 0.0001 for 1 million iterations.

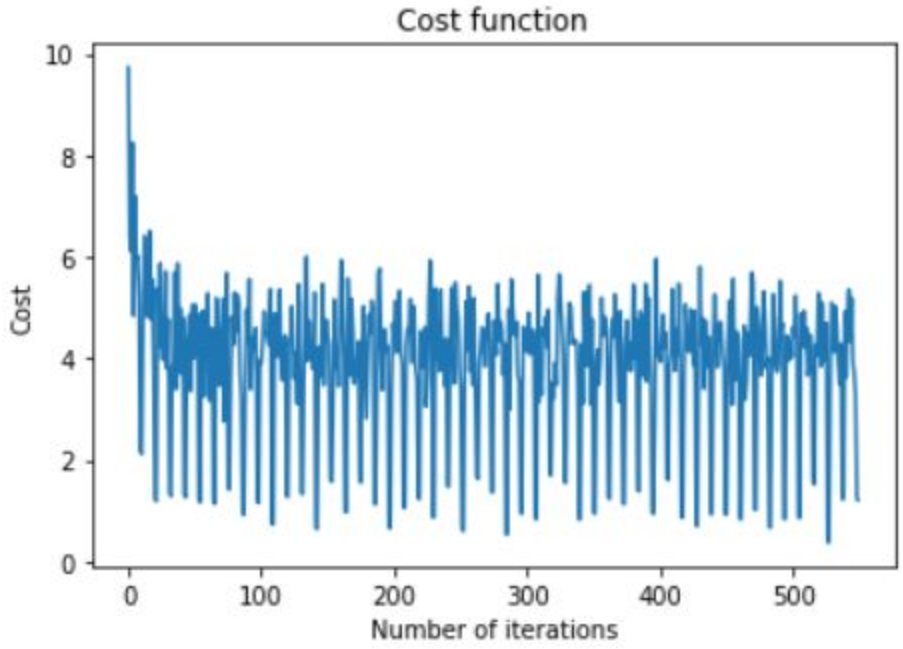



When we run our model for 1 million iterations, we get the above graph. We can see here that this graph is the same as when we ran 100K iterations using 0.001 as learning rate. This graph converges better and we can approximate our minimum error to be 0.525.

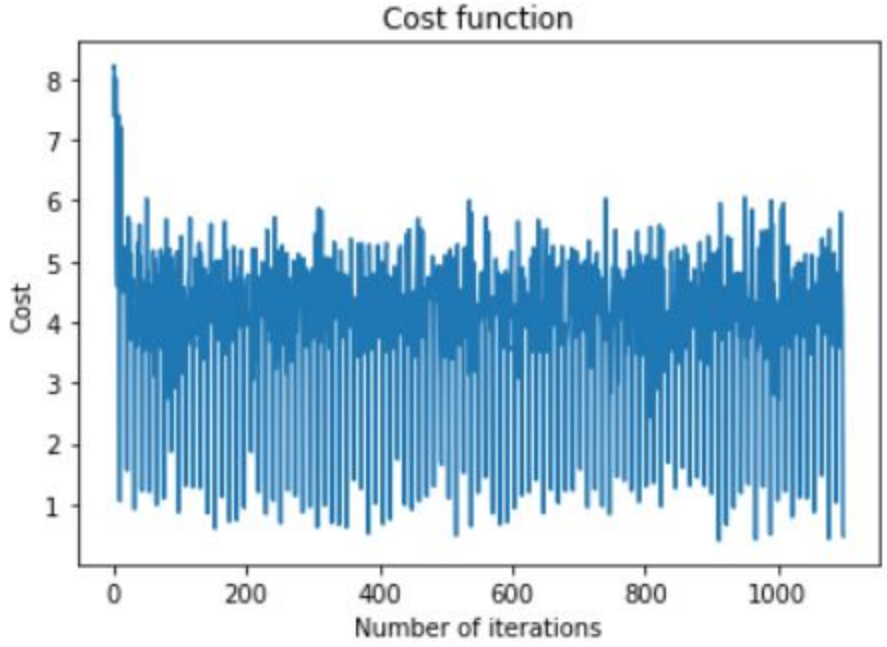
**G.1. Mini-Batch Gradient Descent Iteration**

```
[ ]  def get_batches(X,y,batch_size,i):
         X_new = X[i:i+batch_size,:]
         y_new = y[i:i+batch_size]
         return X_new, y_new

     num_iters = 500
     # Mini-Batch Gradient Descent

     def MiniBatchGradientDescent(X,y,theta,learning_rate=0.001,batch_size=10):
         num_batches = int(X.shape[0]/batch_size)
         gradient = 0
         J_history = []
         for i in range(0,num_batches):
             X_batch, y_batch = get_batches(X,y,batch_size,i)
             hypothesis = np.dot(X_batch,theta)
             loss = hypothesis - y_batch
             X_trans = X_batch.transpose()
             gradient = np.dot(X_trans,loss)/batch_size
             theta = theta - (learning_rate * gradient)
             J_history.append(cost)
         return theta, J_history

     theta, J_history = MiniBatchGradientDescent(X,y,initial_theta, 0.001,10)
```

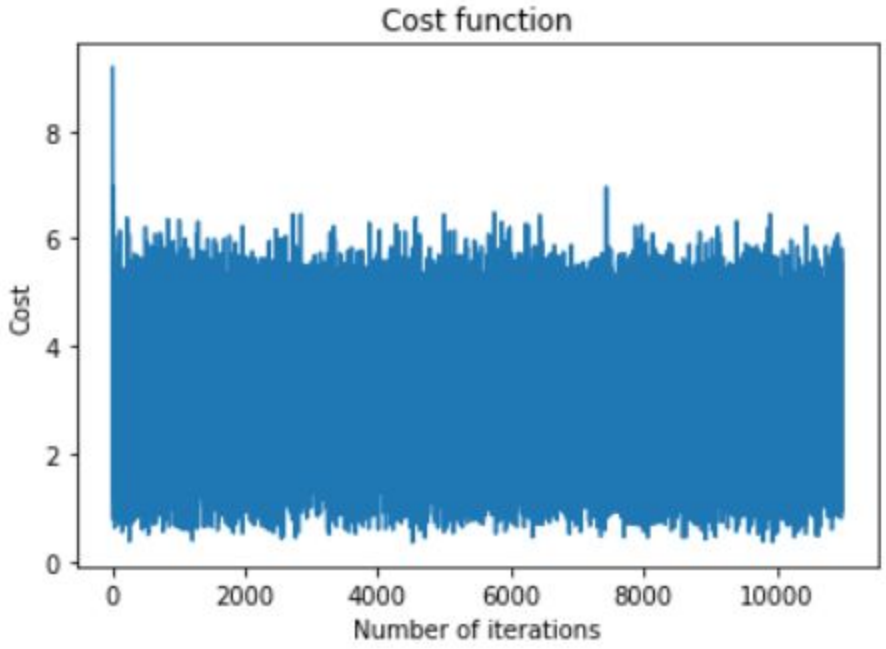
When learning rate = 0.001 and iterations = 500



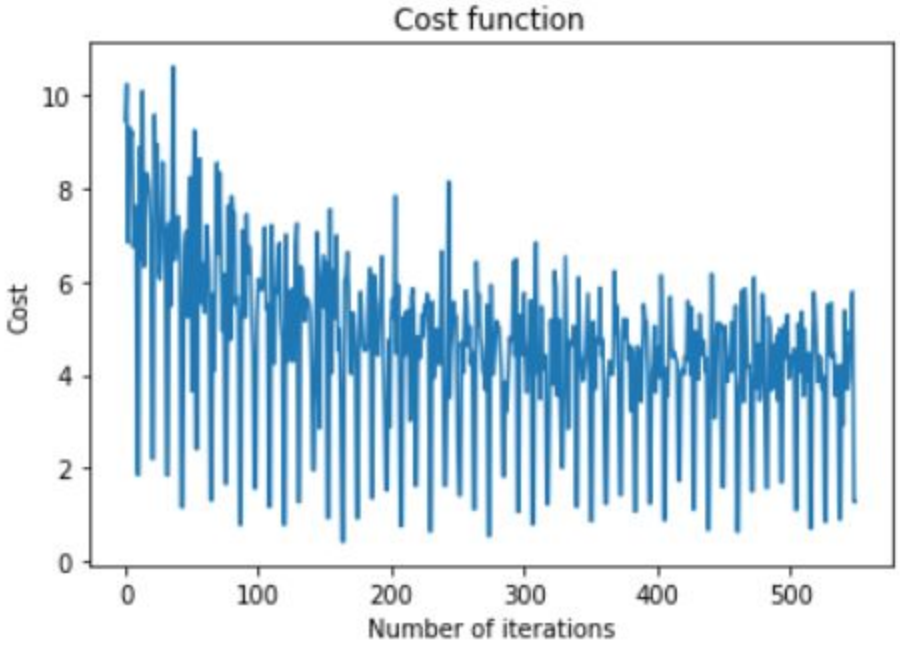

When learning rate = 0.001 and iterations = 1000



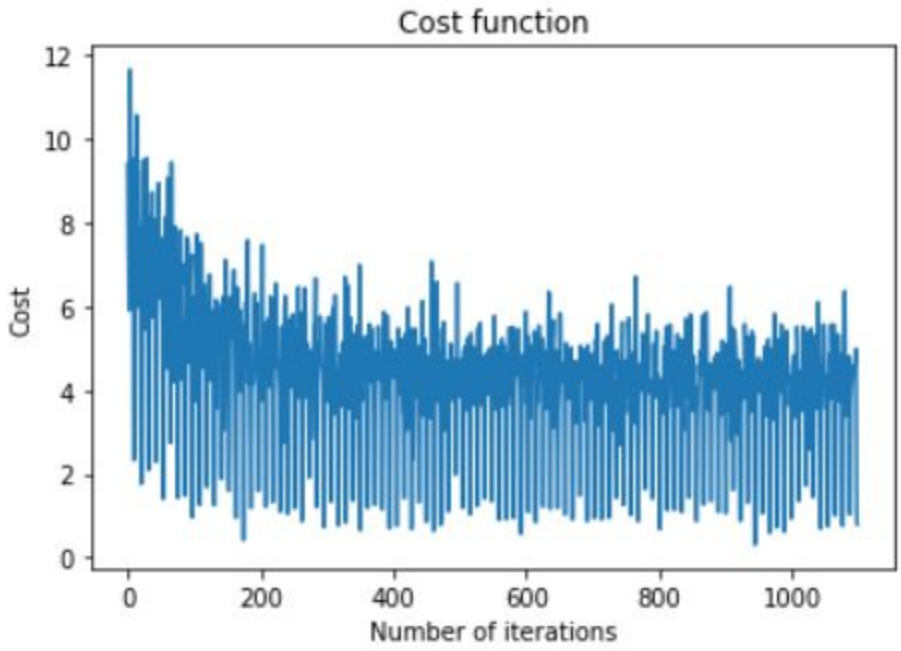

When learning rate = 0.001 and iterations = 10,000



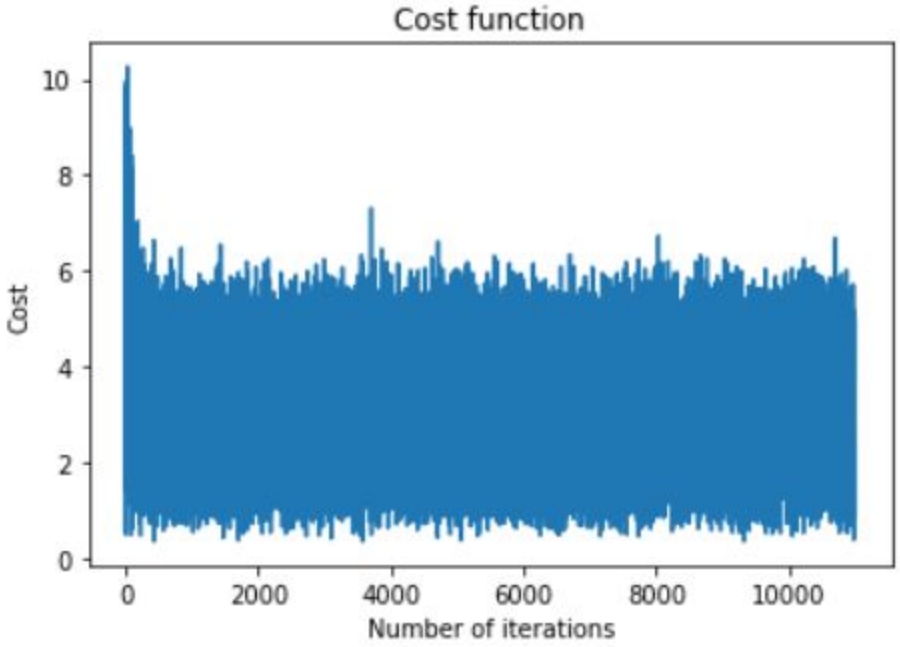

When learning rate = 0.0001 and iterations = 500




When learning rate = 0.0001 and iterations = 1000

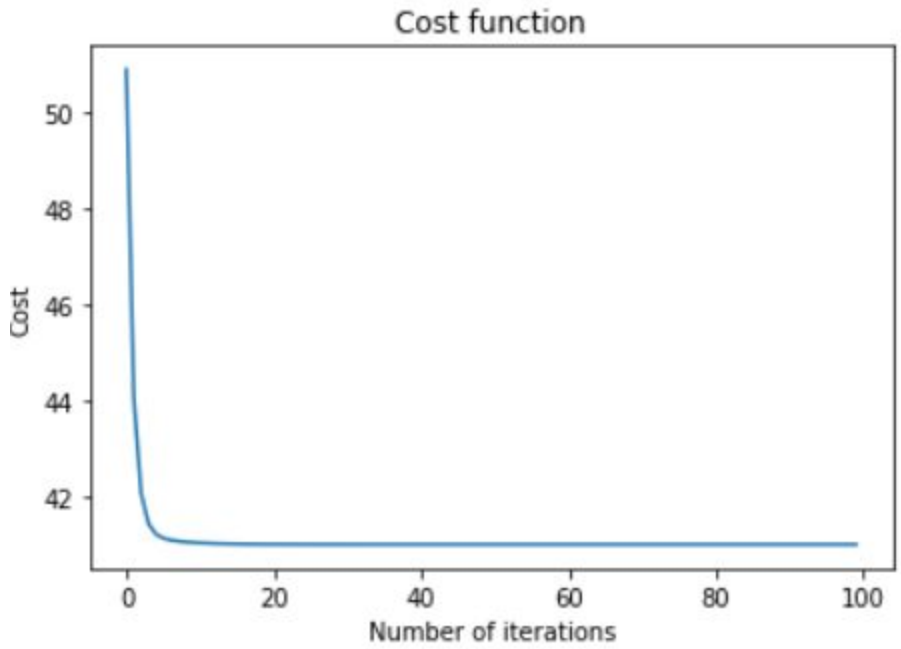



When learning rate = 0.0001 and iterations = 10,000




The six graphs of cost vs. iterations for mini-batch descent are shown above. As we can see, our graphs look very much messier compared to batch gradient descent. Batch gradient descent finds cost for one entire dataframe, whereas mini-batch breaks the dataframe into smaller amounts of samples. Here, we've tried 50 samples (as mentioned in the application manual). This produces graphs that are harder to understand. This can be due to the fact that using mini-batch gradient descent makes it harder

to reach convergence. Selection of a proper learning rate is difficult, this results in a noisier model. Let's try increasing the batch size and see what happens.




We've increased the batch size from 50 in the above image, and we can see that for 100 iterations, our model converges. When working with only 50 batch sizes, our graphs turn out noisy.

**H. Compare with Implemented Library**

```
model = LogisticRegression()
model.fit(X, y)
predicted_classes = model.predict(X)
accuracy = accuracy_score(y.flatten(),predicted_classes)
parameters = model.coef_

print("Coefficients of model: " + str(parameters))
print("accuracy of model: " + str(accuracy*100))
```

```
Index(['row.names', 'sbp', 'tobacco', 'ldl', 'adiposity', 'famhist', 'typea',
       'obesity', 'alcohol', 'age', 'chd'],
      dtype='object')
Coefficients of model: [[-8.32491425e-05  2.53411730e-01  2.23844730e-01  5.01753782e-01
   3.51666196e-01  5.50655686e-01 -2.53411730e-01  3.29891160e-01
  -4.54389266e-01 -3.42375343e-02]]
accuracy of model: 73.37662337662337
```

Using the sklearn implemented library to find our updated coefficients for logistic regression gives us different values of theta. This could be the result of noise and unsteady models when implementing logistic regression from scratch. The implementation of the model using sklearn library gives us an accuracy of 73%.

References

[1]. Benlau93/Machine-Learning-by-Andrew-Ng-in-Python, 2020. [Online].