Lab 4 - Logistic Regression

December 3, 2021

Machine Learning Lab

Lab 04

0.0.1 Exercise 0: Dataset preprocessing

Importing Packages

[1]:
```
import pandas as pd                         #Importing Pandas
import numpy as np                          #Importing Numpy
import seaborn as sns                       #Importing Seaborn
import matplotlib.pyplot as plt             #Importing Matplotlib
```

Reading Tic Tac Toe Dataset

[2]:
```
tic_tac_toe = pd.read_csv('tic-tac-toe.data',header=None)

#Replacing the Column names with more descriptive names
tic_tac_toe.columns =␣
↪['top-left-square','top-middle-square','top-right-square','middle-left-square','middle-midd

                    ␣
↪'middle-right-square','bottom-left-square','bottom-middle-square','bottom-right-square','Cl
tic_tac_toe.head()
```

[2]:

	top-left-square	top-middle-square	top-right-square	middle-left-square	\
0	x	x	x	x	
1	x	x	x	x	
2	x	x	x	x	
3	x	x	x	x	
4	x	x	x	x	

	middle-middle-square	middle-right-square	bottom-left-square	\
0	o	o	x	
1	o	o	o	
2	o	o	o	
3	o	o	o	
4	o	o	b	

	bottom-middle-square	bottom-right-square	Class
0	o	o	positive

```
1                    x                    o  positive
2                    o                    x  positive
3                    b                    b  positive
4                    o                    b  positive
```

Converting any non-numeric values to numeric values.

[3]:
```
#Converting the Class column to numeric column using label transformation
tic_tac_toe.loc[:,'Class'] = pd.factorize(tic_tac_toe['Class'])[0].reshape(-1,1)
tic_tac_toe.head()
```

[3]:
```
  top-left-square top-middle-square top-right-square middle-left-square  \
0               x                 x                x                   x
1               x                 x                x                   x
2               x                 x                x                   x
3               x                 x                x                   x
4               x                 x                x                   x

  middle-middle-square middle-right-square bottom-left-square  \
0                    o                   o                  x
1                    o                   o                  o
2                    o                   o                  o
3                    o                   o                  o
4                    o                   o                  b

  bottom-middle-square bottom-right-square  Class
0                    o                   o      0
1                    x                   o      0
2                    o                   x      0
3                    b                   b      0
4                    o                   b      0
```

The Reason why I used Label transformation for the Class column is because the class can have only two specified values i.e. Positive and Negative.So it is good approach to map each class to specific number, in this case 0 and 1

[4]:
```
#Converting all the other columns except Class to numeric column using Hot one␣
↪encoding
tic_tac_toe = pd.get_dummies(tic_tac_toe,columns=tic_tac_toe.columns[:-1])
tic_tac_toe.head()
```

[4]:
```
   Class  top-left-square_b  top-left-square_o  top-left-square_x  \
0      0                  0                  0                  1
1      0                  0                  0                  1
2      0                  0                  0                  1
3      0                  0                  0                  1
4      0                  0                  0                  1
```

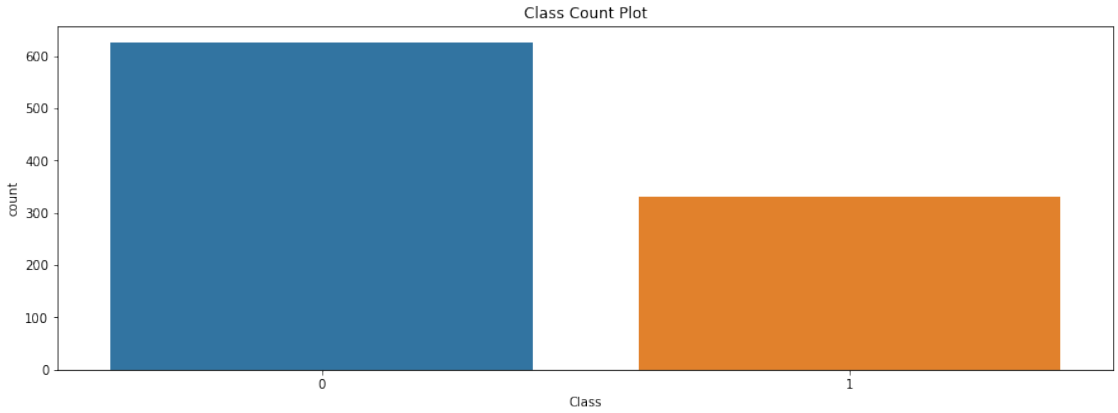
```
   top-middle-square_b  top-middle-square_o  top-middle-square_x  \
0                    0                    0                    1
1                    0                    0                    1
2                    0                    0                    1
3                    0                    0                    1
4                    0                    0                    1

   top-right-square_b  top-right-square_o  top-right-square_x  …  \
0                   0                   0                   1  …
1                   0                   0                   1  …
2                   0                   0                   1  …
3                   0                   0                   1  …
4                   0                   0                   1  …

   middle-right-square_x  bottom-left-square_b  bottom-left-square_o  \
0                      0                     0                     0
1                      0                     0                     1
2                      0                     0                     1
3                      0                     0                     1
4                      0                     1                     0

   bottom-left-square_x  bottom-middle-square_b  bottom-middle-square_o  \
0                     1                       0                       1
1                     0                       0                       0
2                     0                       0                       1
3                     0                       1                       0
4                     0                       0                       1

   bottom-middle-square_x  bottom-right-square_b  bottom-right-square_o  \
0                       0                      0                      1
1                       1                      0                      1
2                       0                      0                      0
3                       0                      1                      0
4                       0                      1                      0

   bottom-right-square_x
0                      0
1                      0
2                      1
3                      0
4                      0

[5 rows x 28 columns]
```

Since all the columns are non numeric, so I am using Hot one encoding to transform each column into multiple numeric columns so that we can train our model more easily.

Checking the Data Imbalance and Implementing Stratified Sampling Methodology

Plotting the Count Plot of different classes to show how much data imbalance is present

[5]:
```
#Using the Count plot from Seaborn to plot frequency of different classes
plt.figure(figsize=(15,5))
sns.countplot(x='Class',data=tic_tac_toe)
plt.title('Class Count Plot')
plt.show()
```



The imbalance data reflects how one class dominate the whole dataset and as a result there is unequal distribution of classes.From the above Tic Tac Toe dataset, we can see that Class 0 (Which is Positive class) has more datap points than the negative class.So this is a imbalance dataset and we have to balance it using stratified sampling.

Function to Stratify the dataset and Split it into Train and Test Sets Stratified Sampling is a techinque which is used to convert imbalance dataset into a balance dataset with each class having the equal distribution.In stratified Sampling we divide our all data points into different groups called Strata and then we pick equally distributed samples from each class to achieve balance in our final dataset.

[6]:
```
def stratified_sample_split(df,label,train_size):
    #Finding the minimum number of rows for classes
    min_sample_size = min(df[label].value_counts())

    #Sampling Positive and Negative Samples from the dataframe with minimum␣
↪number of rows for each class
    positive_samples = df[df[label] == 1].sample(n = min_sample_size , replace␣
↪= False , ignore_index = True)
    negative_samples = df[df[label] == 0].sample(n = min_sample_size , replace␣
↪= False , ignore_index = True)

    #Combining both Positive and Negative classes to a single stratified␣
↪Dataframe
    stratified_df = pd.
↪concat([positive_samples,negative_samples],ignore_index=True)
```

```
    #Shuffling the rows randomly in the stratified dataframe to make data␣
↪Randomized
    stratified_df = stratified_df.sample(frac = 1).reset_index(drop = True)

    #Creating X matrix by removing the label/Target column
    X = stratified_df.drop(label,axis=1)

    #Creating Y vector which include only the Label/Target Column
    Y = stratified_df[label].to_numpy()

    #Adding a Bias Column in the X Matrix
    X = np.append(np.ones(shape=(len(X),1)),X,axis=1)

    #Calculating number of rows to be copied in the Training Set according to␣
↪specific ratio
    total_rows_train = int(len(X)*train_size)

    #Splitting the Dataset into Training set and Test Set based on calculated␣
↪rows
    X_train , X_test = X[:total_rows_train,:] , X[total_rows_train:,:]
    Y_train , Y_test = Y[:total_rows_train].reshape(-1,1) , Y[total_rows_train:
↪].reshape(-1,1)

    return X_train, Y_train, X_test, Y_test
```

Spliting Tic Tac Toe Dataset on 80:20 percent Ratio

[7]:
```
X_train, Y_train, X_test, Y_test =␣
↪stratified_sample_split(tic_tac_toe,'Class',0.8)
```

0.0.2 Exercise 1: Logistic Regression with Gradient Ascent

[8]:
```
#Initializing arrays to store loss difference and Log loss values in different␣
↪number of Iterations
ga_loss_difference_values = np.array([])
ga_logloss_values = np.array([])
newton_loss_difference_values = np.array([])
newton_logloss_values = np.array([])
```

Function to calculate the sigmoid of the given vector

[9]:
```
def sigmoid_function(x):
    # sigmoid(x) = 1 / (1 + e^-x)
    return (1 / (1 + np.exp(np.negative(x))))
```

Function to calculate the Loglikelihood Loss

[10]:
```
def loglikelihood_loss(X,Y,B):
    # L = summation((Y - Y_pred) - log(1 + e^Y_pred))
```

```
    return np.sum((Y * sigmoid_function(X @ B)) - np.log(1 + np.
↪exp(sigmoid_function(X @ B))))
```

Function to calculate the Loglikelihood loss difference between current Beta and previous Beta

[11]:
```
def loss_difference(X,Y,B_old,B_new):
    # |L(B_old) - L(B_new)|
    return np.abs(loglikelihood_loss(X,Y,B_old) - loglikelihood_loss(X,Y,B_new))
```

Function to calculate the log absolute loss of predicted Y

[12]:
```
def log_loss(X,Y,B):
    # Log Loss = |loglokelihood_loss(X,Y,B)|
    return np.abs(loglikelihood_loss(X,Y,B))
```

Function which returns the gradient of Loss function

[13]:
```
def dL(X,Y,B):
    # Derivative of Loss = X^T * (Y - Y_pred)
    return X.T @ (np.subtract(Y,sigmoid_function(X @ B)))
```

Function to calculate optimum learning rate using Bold Driver Algorithm

[14]:
```
def steplength_bolddriver(X,Y,B,alpha_old,alpha_plus,alpha_minus):
    #Increasing alpha value using alpha+
    alpha = alpha_old*alpha_plus

    #Iterating until following condition is meet:
    #f(x + μd) - f(x) < 0
    while(loglikelihood_loss(X,Y,B + (alpha * dL(X,Y,B))) -␣
↪loglikelihood_loss(X,Y,B) < 0):
        #Slowly Increasing alpha using alpha-
        alpha = alpha * alpha_minus
    return alpha
```

Function to Maximize the Gradient Ascent based on Total Iterations and Learning Rate using Bold Driver Algorithm

[15]:
```
def learn_logreg_GA(X_train,Y_train,X_test,Y_test,imax,epsilon):
    #Using global arrays to store loss difference and rmse values for different␣
↪Iterations
    global ga_loss_difference_values , ga_logloss_values

    #Emptying both Loss difference and RMSE arrays
    ga_loss_difference_values , ga_logloss_values = np.array([]) , np.array([])

    #Initializing beta vector with zeros
    beta = np.zeros(shape=(len(X_train[0]),1))

    #Initializing initial alpha value for Bold Driver algorithm
```

```
    alpha = 5e-5
    for i in range(imax):
        #Calculating new alpha/step length value using Bold Driver Algorithm
        alpha = steplength_bolddriver(X_train,Y_train,beta,alpha,1,0.5)

        #Calculating new Beta values from previous beta values and gradient␣
↪ascent direction
        #Beta = Beta + learning_rate * gradient ascent based on Beta
        beta_ = beta + (alpha * dL(X_train,Y_train,beta))

         #Appending Loss difference between between previous and new Beta
        ga_loss_difference_values = np.
↪append(ga_loss_difference_values,loss_difference(X_train,Y_train,beta,beta_))

        #Appending RMSE loss between actual Y and Predicted Y
        ga_logloss_values = np.
↪append(ga_logloss_values,log_loss(X_test,Y_test,beta_))

        #Checking the Stopping Condition by comparing loss on previous and␣
↪current beta
        if loglikelihood_loss(X_train,Y_train,beta_) -␣
↪loglikelihood_loss(X_train,Y_train,beta) < epsilon:
            return beta_

        #Copying new Beta value to old Beta value for Further Calculation
        beta = np.copy(beta_)
    return beta_
```

Calculating new Beta values using gradient ascent function and 1000 iterations

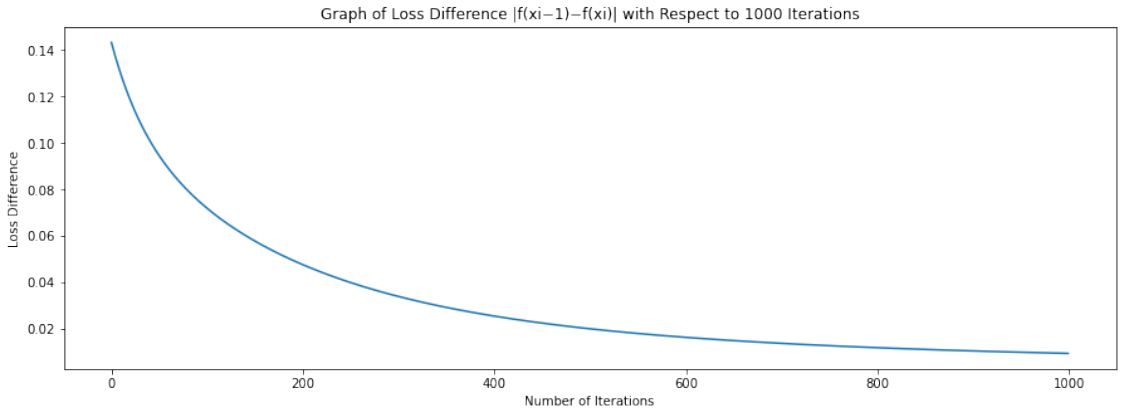
[16]:
```
beta = learn_logreg_GA(X_train,Y_train,X_test,Y_test,1000,10**-5)
```

Plotting Loss difference and Log loss after Gradient Ascent

Graph of Loss Difference

[17]:
```
fig = plt.figure(figsize=(15,5))
ax = fig.add_subplot(111)
ax.plot([i for i in␣
 ↪range(len(ga_loss_difference_values))],ga_loss_difference_values)
ax.set_title('Graph of Loss Difference |f(xi-1)-f(xi)| with Respect to 1000␣
 ↪Iterations')
ax.set_xlabel('Number of Iterations')
ax.set_ylabel('Loss Difference')
```

[17]:
```
Text(0, 0.5, 'Loss Difference')
```

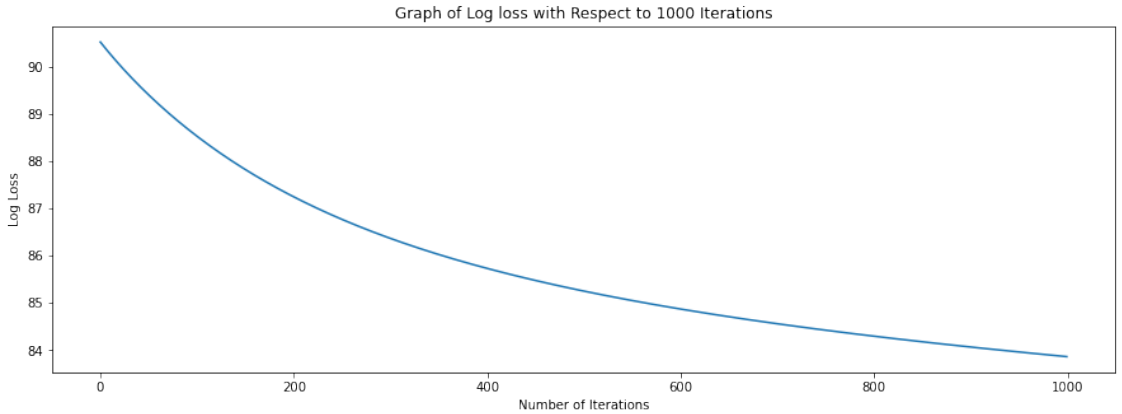


The above graph shows that our loss difference is decreasing as the number of iterations are increased.

Graph of Log Loss

[18]:
```
fig = plt.figure(figsize=(15,5))
ax = fig.add_subplot(111)
ax.plot([i for i in range(len(ga_logloss_values))],ga_logloss_values)
ax.set_title('Graph of Log loss with Respect to 1000 Iterations')
ax.set_xlabel('Number of Iterations')
ax.set_ylabel('Log Loss')
```

[18]:
```
Text(0, 0.5, 'Log Loss')
```



This graph is also showing that our log loss is decreasing as the number of iterations are increased.The Log loss is decreasing not increasing because we have taken the absolute value of our loglikelihood loss

0.0.3 Exercise 2: Implement Newton Algorithm for Logistic Regression

Function to find Local Minima Using Newton's Algorithm for Logistic Regression

[19]:
```
def minimize_newton(X,Y,imax,epsilon):
    #Using global arrays to store loss difference and rmse values for different␣
↪Iterations
    global newton_loss_difference_values , newton_logloss_values

    #Emptying both Loss difference and RMSE arrays
    newton_loss_difference_values , newton_logloss_values = np.array([]) , np.
↪array([])

    #Initializing beta with Zeros
    beta = np.zeros(shape=(len(X[0]),1))
    for i in range(imax):
        #Calculating the Inverse of Hessian Matrix
        #W = diagonal(Y_pred . (1 - Y_pred))
        W = np.diag(sigmoid_function(X @ beta.reshape(1,-1)[0]) * (1 -␣
↪sigmoid_function(X @ beta.reshape(1,-1)[0])))

        #H = X^T * W * X
        H = X.T.dot(W).dot(X)

        #H^-1 = inv(H)
        H_inv = np.linalg.inv(H)

        #Calculating new Beta values from previous beta values and Newton's␣
↪Hessian matrix
        #Beta = Beta - learning_rate * Hessian Matrix * gradient ascent based␣
↪on Beta
        beta_ = beta + (0.0001 * (H_inv @ dL(X,Y,beta)))

         #Appending Loss difference between between previous and new Beta
        newton_loss_difference_values = np.
↪append(newton_loss_difference_values,loss_difference(X,Y,beta,beta_))

        #Appending Log loss for Predicted Y
        newton_logloss_values = np.
↪append(newton_logloss_values,log_loss(X_test,Y_test,beta_))

        #Checking the Stopping Condition by comparing loss on previous and␣
↪current beta
        if loglikelihood_loss(X_train,Y_train,beta_) -␣
↪loglikelihood_loss(X_train,Y_train,beta) < epsilon:
            return beta_

        #Copying new Beta value to old Beta value for Further Calculation
        beta = np.copy(beta_)
    return beta_
```

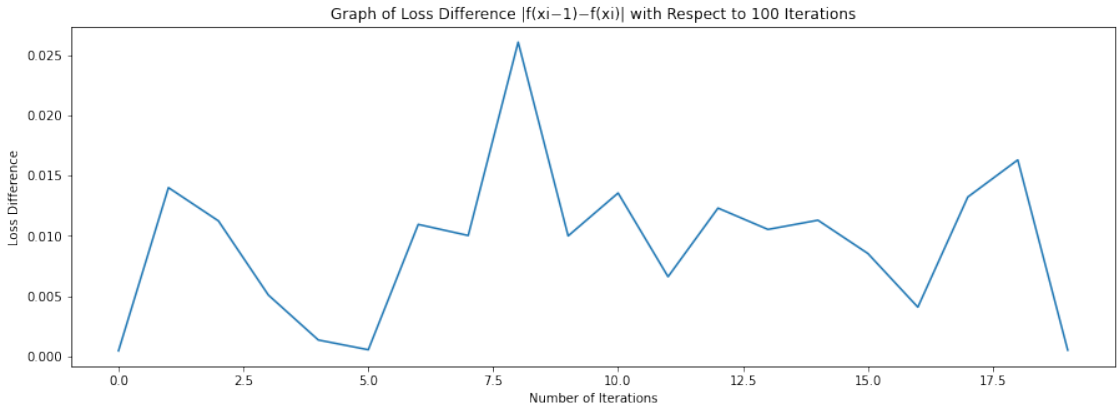
Calculating new Beta values using Newton's Algorithm and 100 iterations

[20]:
```
beta = minimize_newton(X_train,Y_train,100,10**-5)
```

Plotting Loss difference and Log loss after Newton's Algorithm

[21]:
```
fig = plt.figure(figsize=(15,5))
ax = fig.add_subplot(111)
ax.plot([i for i in␣
 ↪range(len(newton_loss_difference_values))],newton_loss_difference_values)
ax.set_title('Graph of Loss Difference |f(xi-1)-f(xi)| with Respect to 100␣
 ↪Iterations')
ax.set_xlabel('Number of Iterations')
ax.set_ylabel('Loss Difference')
```

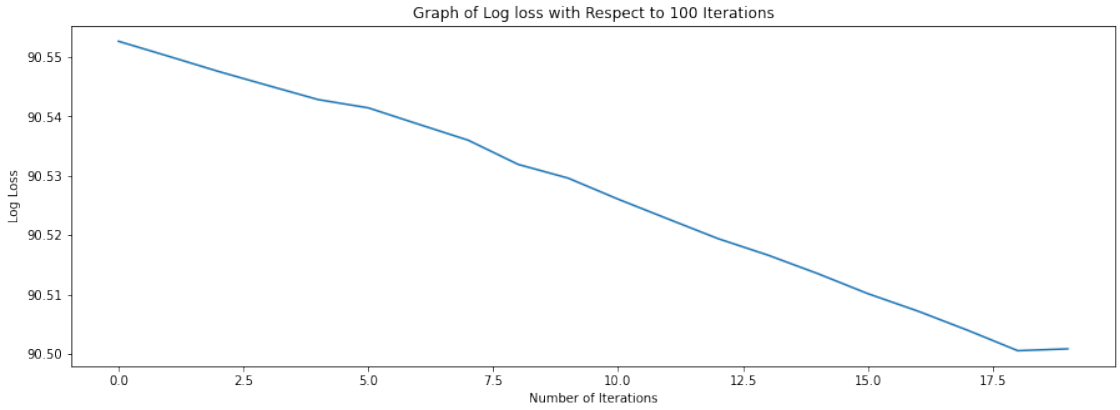
[21]:
```
Text(0, 0.5, 'Loss Difference')
```



The above graph shows that our loss difference is fluctuating but it reaches its lowest value only after 16th iteration which is very fast as compared to Gradient Ascent

[22]:
```
fig = plt.figure(figsize=(15,5))
ax = fig.add_subplot(111)
ax.plot([i for i in range(len(newton_logloss_values))],newton_logloss_values)
ax.set_title('Graph of Log loss with Respect to 100 Iterations')
ax.set_xlabel('Number of Iterations')
ax.set_ylabel('Log Loss')
```

[22]:
```
Text(0, 0.5, 'Log Loss')
```

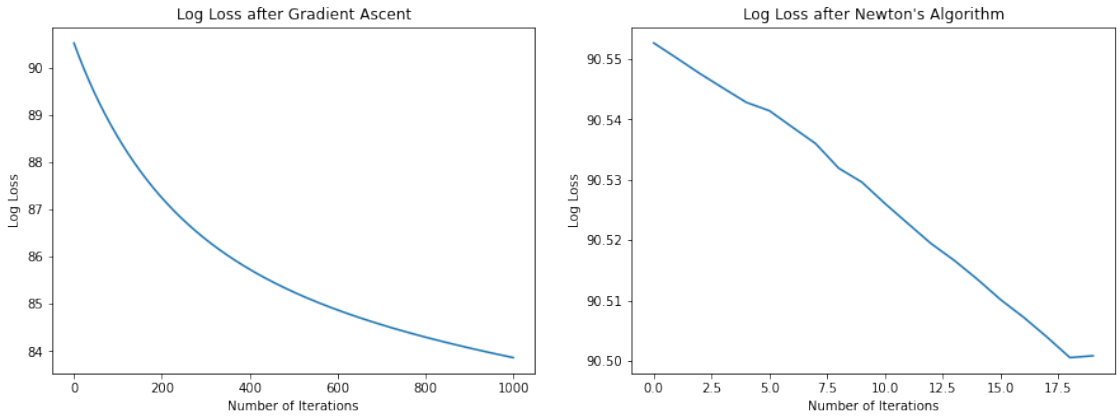


The above graph also shows that our log loss is decreasing very rapidly and after only 16th iteration it reaches its local minima which is really fast as compared to gradient ascent

Comments on the rate of convergence in the light of plots from above

[23]:
```
#Plotting Log Loss after Gradient Ascent
fig = plt.figure(figsize=(15,5))
ax = fig.add_subplot(121)
ax.plot([i for i in range(len(ga_logloss_values))],ga_logloss_values)
ax.set_title('Log Loss after Gradient Ascent')
ax.set_xlabel('Number of Iterations')
ax.set_ylabel('Log Loss')

#Plotting Log Loss after Newton's Algorithm
ax1 = fig.add_subplot(122)
ax1.plot([i for i in range(len(newton_logloss_values))],newton_logloss_values)
ax1.set_title('Log Loss after Newton\'s Algorithm')
ax1.set_xlabel('Number of Iterations')
ax1.set_ylabel('Log Loss')
```

[23]:
```
Text(0, 0.5, 'Log Loss')
```



From the above graph we can see that the gradient ascent takes more iterations to converge as compared to Newton's Algorithm which takes lesser Iterations.So Newton's Algorithm converges faster than Gradient Ascent because of Second Order Differentiation.