# Lab Course Machine Learning
# Exercise 4

## 1.1 Data preprocessing:

**Function name** : read_data
**Parameter** : Filename and column names

This function reads the data and returns the data in the format of data frame along with the headings.

```
def read_data(filename,columns):
    if columns == None:
        data = pd.read_csv(filename, sep='\s+', delimiter = ";")
        return data
    data = pd.read_csv(filename, sep='\s+',header = None)
    data.columns = columns
    return data
```

**Output:**

**Bank data**

| | age | job | marital | education | default | balance | housing | loan | contact | day | month | duration | campaign | pdays | previous | poutcome | y |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 30 | unemployed | married | primary | no | 1787 | no | no | cellular | 19 | oct | 79 | 1 | -1 | 0 | unknown | no |
| 1 | 33 | services | married | secondary | no | 4789 | yes | yes | cellular | 11 | may | 220 | 1 | 339 | 4 | failure | no |
| 2 | 35 | management | single | tertiary | no | 1350 | yes | no | cellular | 16 | apr | 185 | 1 | 330 | 1 | failure | no |
| 3 | 30 | management | married | tertiary | no | 1476 | yes | yes | unknown | 3 | jun | 199 | 4 | -1 | 0 | unknown | no |
| 4 | 59 | blue-collar | married | secondary | no | 0 | yes | no | unknown | 5 | may | 226 | 1 | -1 | 0 | unknown | no |

**Occupancy data**

| | date | Temperature | Humidity | Light | CO2 | HumidityRatio | Occupancy |
|---|---|---|---|---|---|---|---|
| 1 | 2015-02-04 17:51:00 | 23.18 | 27.2720 | 426.0 | 721.25 | 0.004793 | 1 |
| 2 | 2015-02-04 17:51:59 | 23.15 | 27.2675 | 429.5 | 714.00 | 0.004783 | 1 |
| 3 | 2015-02-04 17:53:00 | 23.15 | 27.2450 | 426.0 | 713.50 | 0.004779 | 1 |
| 4 | 2015-02-04 17:54:00 | 23.15 | 27.2000 | 426.0 | 708.25 | 0.004772 | 1 |
| 5 | 2015-02-04 17:55:00 | 23.10 | 27.2000 | 426.0 | 704.50 | 0.004757 | 1 |

**Function name** : check_na_null
**Parameter** : Dataframe and column names

This function checks the presence of null values or NA or nan values within the dataset and removes the same. This function also generates dummies for non-numeric values and returns the same.

```
def check_na_null(data,column_dummies):
    data.isnull().values.any()
    data.dropna(inplace = True)
    data = pd.get_dummies(data=data, columns=column_dummies)
    return data
```

**Output:**

Dummies and nan or NA value check for Bank data

| | age | balance | campaign | pdays | previous | y | job_admin. | job_blue-collar | job_entrepreneur | job_housemaid | ... | education_unknown | default_no | default_yes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **712** | 28 | 102 | 2 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 1 | 0 |
| **3177** | 32 | 11797 | 2 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 1 | 0 |
| **4029** | 37 | 487 | 15 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 1 | 0 |
| **1074** | 28 | 594 | 3 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 1 | 0 |
| **4117** | 25 | 8 | 2 | -1 | 0 | 1 | 0 | 0 | 0 | 0 | ... | 0 | 1 | 0 |

5 rows × 34 columns

**Function name** : text_to_number
**Parameter** : Dataframe and column names

This function converts the text within a column to numbers. For example, a city with name ABQ will be converted to a specific number for all the available instances.

```
def text_to_number(data,column):
    for i in column:
        Airdata[i] = pd.Categorical(Airdata[i])
        Airdata[i] = Airdata[i].cat.codes
    return data
```

**Output:**

| | age | job | marital | education | default | balance | contact | campaign | pdays | previous | poutcome | y |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1143** | 57 | housemaid | married | primary | no | 501 | cellular | 2 | -1 | 0 | unknown | 0 |
| **4471** | 59 | management | married | unknown | no | 3534 | cellular | 4 | -1 | 0 | unknown | 0 |
| **4078** | 30 | management | married | tertiary | no | 562 | cellular | 4 | -1 | 0 | unknown | 0 |
| **1646** | 38 | technician | single | tertiary | no | 2273 | cellular | 1 | -1 | 0 | unknown | 0 |
| **869** | 56 | technician | married | secondary | no | 150 | cellular | 6 | 349 | 1 | failure | 0 |

**Function name** : create_Test_Train_data
**Parameter** : Data of input variables and target

This function splits the input variables to Train and Test data with 80% and 20% of the entire data respectively.

```
def create_Test_Train_data(X_data,Y_data):
    Y_train = Y_data[:math.ceil(0.8*len(Y_data))]
    Y_test = Y_data[math.ceil(0.8*len(Y_data)):]
    X_train = X_data[:math.ceil(0.8*len(X_data))]
    X_test = X_data[math.ceil(0.8*len(X_data)):]
    return Y_train,Y_test,X_train,X_test
```

**Output:**

Bank data

```
Xtrain:  (3617, 33)
Ytrain:  (3617,)
Xtest:  (904, 33)
Ytest:  (904,)
```

Occupancy data

```
Xtrain:  (6515, 5)
Ytrain:  (6515,)
Xtest:  (1628, 5)
Ytest:  (1628,)
```

## Linear Classification with Gradient Descent:

### Bank dataset:

To begin with the problem, first we've to find the correlation between the input variables and the client subscription of a term deposit. As the dataset contains both categorical and numerical values, we can use spearman correlation coefficient which uses ranked variables and access them with pearson co-efficient correlation between two variables.

**Function name** : spearman_rank_coefficient
**Parameter** : Data for correlation

*Reference:* *https://statistics.laerd.com/statistical-guides/spearmans-rank-order-correlation-statistical-guide.php*

The Spearman's rank-order correlation is the nonparametric version of the Pearson product-moment correlation. Spearman's correlation coefficient, (ρ, also signified by rs) measures the strength and direction of association between two ranked variables.

Assumption: You need two variables that are either ordinal, interval or ratio

$$r_s = \rho_{rg_X, rg_Y} = \frac{cov(rg_x, rg_y)}{\sigma_{rg_X, rg_Y}}$$

**Function name** : pearson_coefficient
**Parameter** : Ranked data for correlation

*Reference:* *https://statistics.laerd.com/statistical-guides/pearson-correlation-coefficient-statistical-guide.php*

The Pearson correlation coefficient (PCC) is a measure of the linear correlation between two variables X and Y. PCC or 'r' has a value between +1 and −1.

Assumptions:

1.The variables must be either interval or ratio measurements.
2.The variables must be approximately normally distributed.
3.There is a linear relationship between the two variables
4.Outliers are either kept to a minimum or are removed entirely.
5.There is homoscedasticity of the data.

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

```
def pearson_coefficient(x,y):
    x = x - np.mean(x)
    y = y - np.mean(y)
    return (np.sum(x*y)/np.sqrt(np.sum(x*x)*np.sum(y*y)))
```

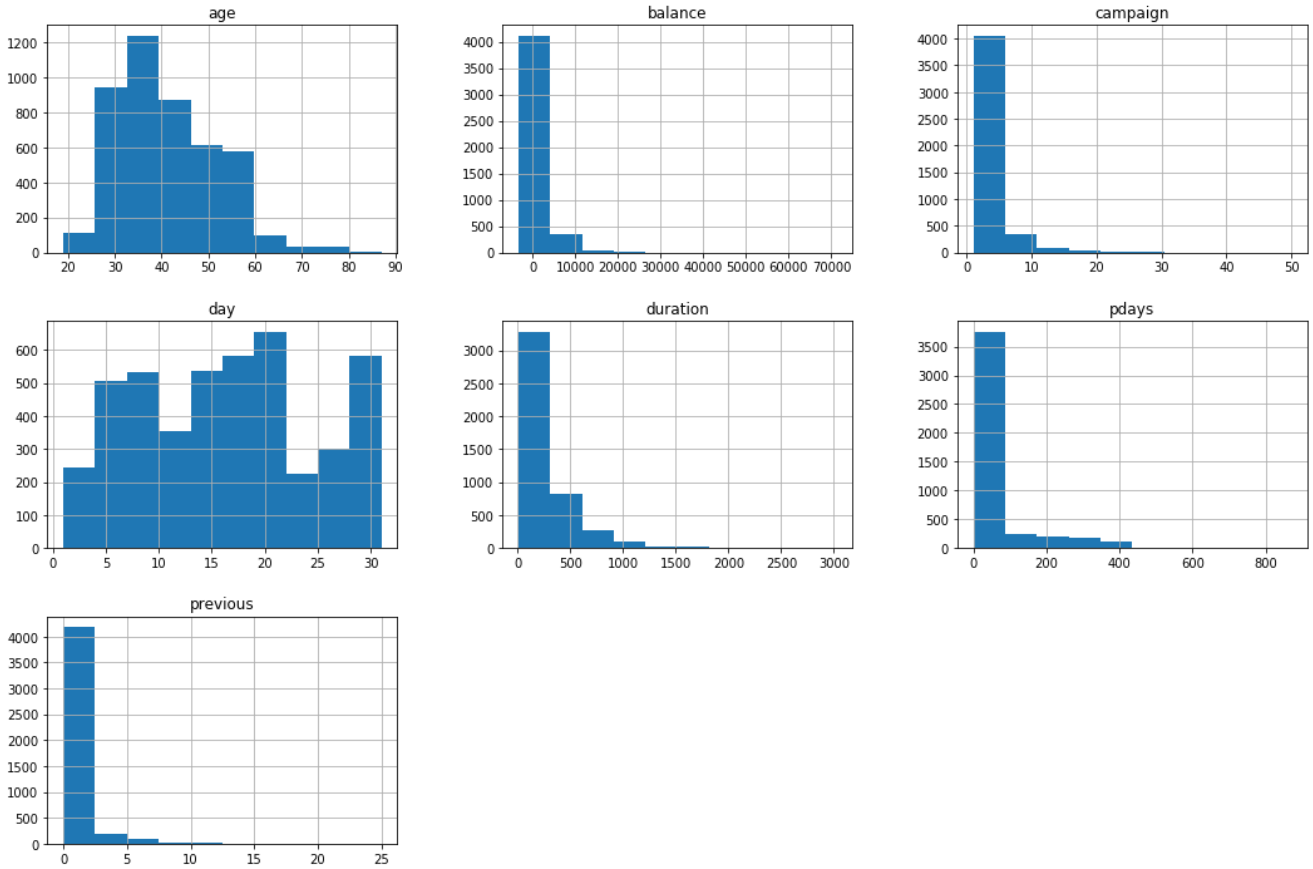
**Output:**

Correlation between subscription of term deposit and other variables for the bank data

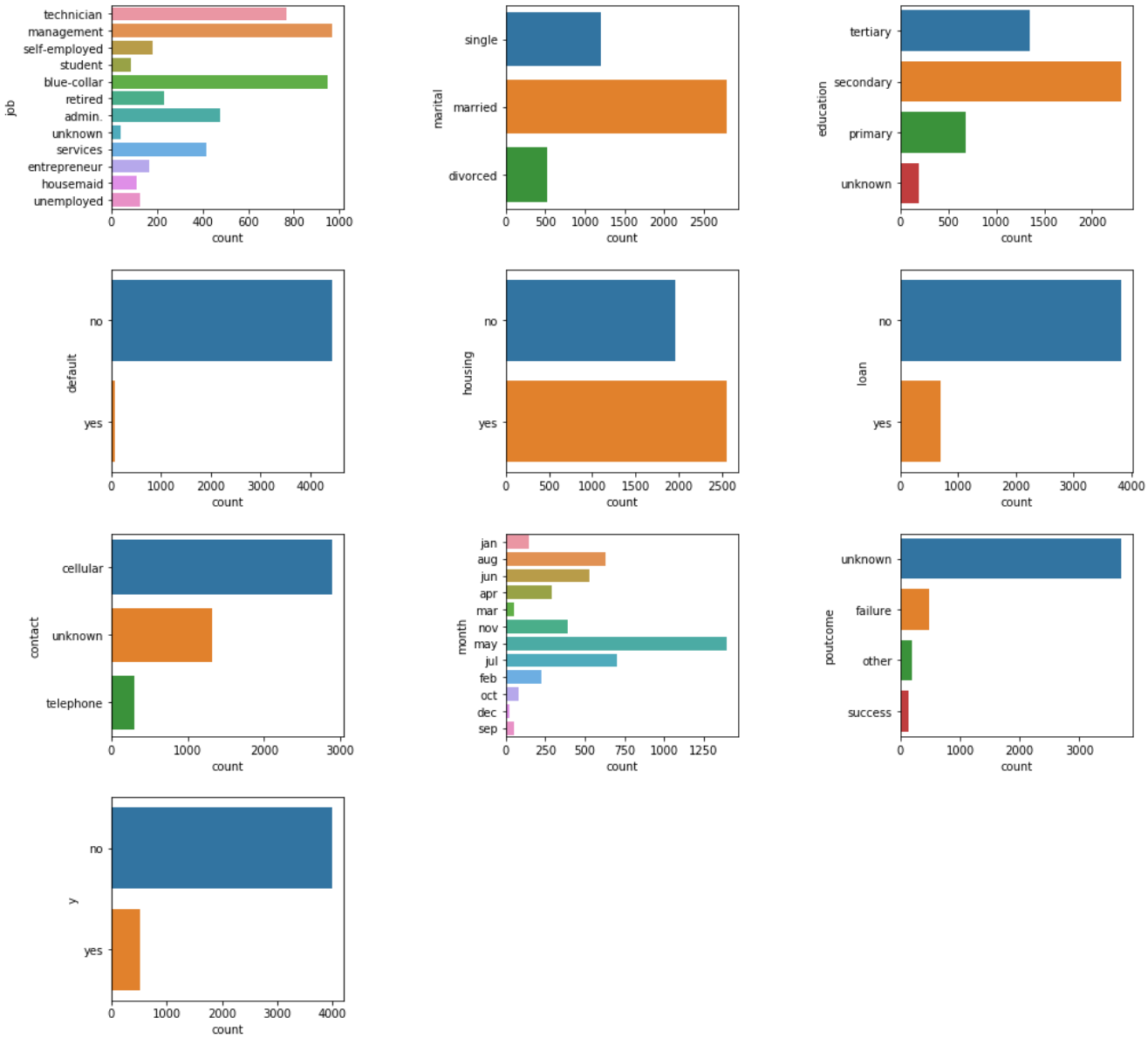
| | age | job | marital | education | default | balance | housing | loan | contact | day | month | duration | campaign | pdays | previous | poutcome | y |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.0163 | 0.03 | 0.0193 | 0.0481 | 0.0013 | 0.079 | -0.1047 | -0.0705 | -0.1295 | -0.0118 | -0.0326 | 0.3484 | -0.0646 | 0.1502 | 0.1653 | -0.1413 | NaN |

From the correlation values between the subscription term deposit and other variables, I'm dropping the columns of day, month, housing, loan and duration from the dataset. The reason why I am dropping duration column is that at the end of the phone call, it is obvious that we get the class of the subscription. Hence it can be dropped.

Visualization of numerical columns of the bank dataset



Visualization of categorical values in the bank dataset




```
Required_bankdata = Bankdata.drop(columns = ['day','month','housing','loan','duration'])
```

**Output:**

Data after dropping the poorly correlated columns

| | age | job | marital | education | default | balance | contact | campaign | pdays | previous | poutcome | y |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1143** | 57 | housemaid | married | primary | no | 501 | cellular | 2 | -1 | 0 | unknown | 0 |
| **4471** | 59 | management | married | unknown | no | 3534 | cellular | 4 | -1 | 0 | unknown | 0 |
| **4078** | 30 | management | married | tertiary | no | 562 | cellular | 4 | -1 | 0 | unknown | 0 |
| **1646** | 38 | technician | single | tertiary | no | 2273 | cellular | 1 | -1 | 0 | unknown | 0 |
| **869** | 56 | technician | married | secondary | no | 150 | cellular | 6 | 349 | 1 | failure | 0 |

To begin with the model generation, we have to initialise zero values for beta depending on the number of input variables

```
beta = np.zeros(X_data.shape[1] + 1)
```

**Output:**

```
(array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]),
 (34,))
```

**Function name** : loglikelihood
**Parameter** : Y data and dot product of beta and X

This function returns the loglikelihood of the function to be minimized. This is the function that we've to check for convergence.

$$logL_D(\hat{\beta}) = \sum_{n=1}^{N} y_n(x_n.\hat{\beta}) - log(1 + e^{(x_n.\hat{\beta})})$$

```
def loglikelihood(Y,betaX):
    return  np.sum(np.dot(Y,betaX) - np.log(1 + np.exp(betaX)))
```

**Function name** : logloss
**Parameter** : Test data of X,Y and beta values

This function returns the logarithmic loss of the test data from the predicted beta values from one of the classification methods.

*Reference:* [http://wiki.fast.ai/index.php/Log_Loss#Binary_Classification](http://wiki.fast.ai/index.php/Log_Loss#Binary_Classification)

In a binary classification (M=2), the formula equals:

$$-y(log(p)) + (1 - y)(log(1 - p))$$

```
def logloss(Ytest,Xtest,beta):
    X = np.insert(Xtest, 0, 1, axis = 1)
    betaX = np.dot(beta,X.T)
    Y_pred = 1/(1 + np.exp(-betaX))
    return -np.sum(Ytest * np.log(Y_pred) + (1 - Ytest)
                   * np.log(1 - Y_pred))/len(Ytest)
```

**Function name** : logreg_GD
**Parameter** : training and testing data of X and Y, steplength

This function performs the linear classification using Gradient Descent algorithm and calls the function for the same.

```
def logreg_gd(Y_train,Y_test,X_train,X_test,alpha):
    beta = np.zeros(X_data.shape[1] + 1)
    diff_graph,logloss_graph,beta = logisticregression(alpha,beta,Y_train,Y_test,X_train,X_test)
```

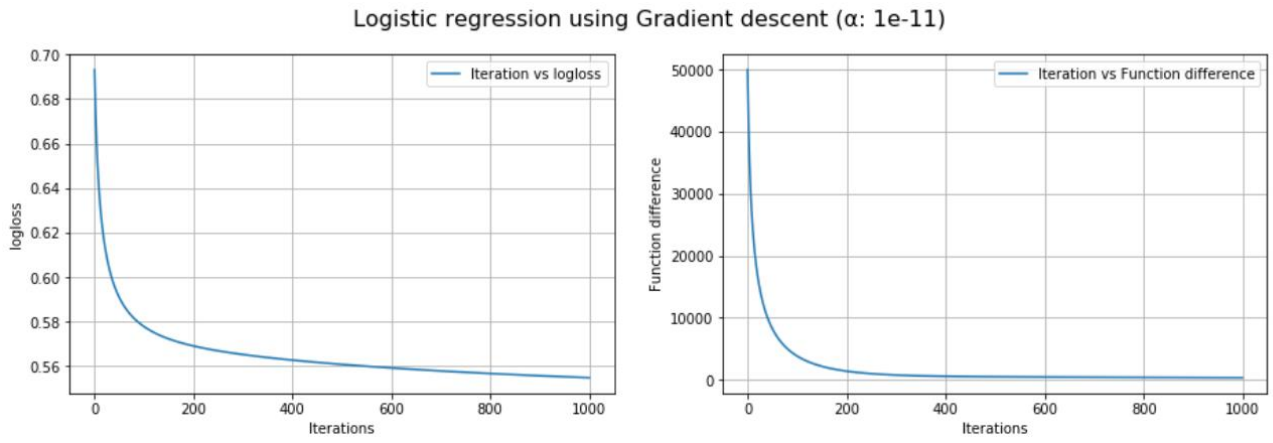
```
def logisticregression(alpha,beta,Y_train,Y_test,X_train,X_test):
    X = np.insert(X_train, 0, 1, axis = 1)
    logloss_graph = []
    diff_graph = []
    betaX = np.dot(beta,X.T)
    X_transpose = X.T
    likelihood = loglikelihood(Y_train,betaX)
    logloss_graph.append(logloss(Y_test,X_test,beta))
    iterations = 1000
    for i in range(0,iterations):
        func_gradient = - np.dot(X_transpose,Y_train - 1/(1 + np.exp(-betaX)))
        beta = beta - alpha * func_gradient
        betaX = np.dot(beta,X.T)
        likelihood_new = loglikelihood(Y_train,betaX)
        logloss_graph.append(logloss(Y_test,X_test,beta))
        diff_graph.append(abs(likelihood_new - likelihood))
        if abs(likelihood_new - likelihood) < 1e-10:
            return diff_graph,logloss_graph, beta
        likelihood = likelihood_new
    return diff_graph,logloss_graph,beta
```

This function generates the model using linear classification with gradient descent with constant step length. I have used a step length of 1e-11 which is used constantly till the likelihood function converges. This function finds the logloss, likelihood of the function for every iteration and makes a list of the same for plotting the graphs. It returns the list of difference between functions, logarithmic losses and beta values.

**Output:**

```
β:  [-8.97545547e-06 -3.61941444e-04 -4.16479051e-04 -8.88951297e-04
 -2.78447677e-05 -2.22754007e-04 -2.16653068e-06 -9.57416973e-07
 -2.34774594e-06 -3.74415481e-07 -2.02049427e-07 -1.54804114e-06
 -2.18457536e-07 -3.82578242e-07 -9.29094832e-07 -7.19702120e-08
 -1.58225822e-06 -3.05960888e-07 -5.54665796e-08 -9.91402082e-07
 -6.01696460e-06 -1.96708880e-06 -1.44410013e-06 -5.04322387e-06
 -2.11173127e-06 -3.76400213e-07 -8.75731261e-06 -2.18142867e-07
 -4.99233207e-06 -4.74286551e-07 -3.50883685e-06 -8.74058068e-07
 -2.16990223e-07  2.97836787e-07 -8.18224397e-06]

Logloss: 0.554900151391623
```



As we can see from these graphs plotted between iteration and logloss and function difference for a step length of 1e-11, we get a log loss of 0.555 and the function is converging near to 0 which with more iterations will be completely converged.

**Function name** : stochasticlogisticregression
**Parameter** : steplength, beta, training and testing data of X and Y and step length type

```
def stochasticlogisticregression(alpha,beta,Y_train,Y_test,X_train,X_test,steplength_type):
    epochs = 50

    X = np.insert(X_train, 0, 1, axis = 1)
    logloss_graph = []
    diff_graph = []
    betaX = np.dot(beta,X.T)
    likelihood = loglikelihood(Y_train,betaX)
    logloss_graph.append(logloss(Y_test,X_test,beta))
    index = np.arange(0,len(X))
    history = [0] * len(beta)
    for epoch in range(0,epochs):
        np.random.shuffle(index)
        for i in index:
            xtr = X[i]
            ytr = Y_train[i]
            betaX = np.dot(beta,xtr.T)
            if steplength_type == "Adagrad":
                beta, history = adagrad_steplength(alpha,beta,xtr,ytr,history)
            else:
                func_gradient = - np.dot(xtr.T,ytr - 1/(1 + np.exp(-betaX)))
                beta = beta - alpha * func_gradient
        betaX = np.dot(beta,X.T)
        likelihood_new = loglikelihood(Y_train,betaX)
        diff_graph.append(abs(likelihood_new - likelihood))
        logloss_graph.append(logloss(Y_test,X_test,beta))
        likelihood = likelihood_new
        if (steplength_type == "Bold"):
            alpha = bold_steplength(alpha,likelihood_new,likelihood)
    return logloss_graph,diff_graph,beta
```

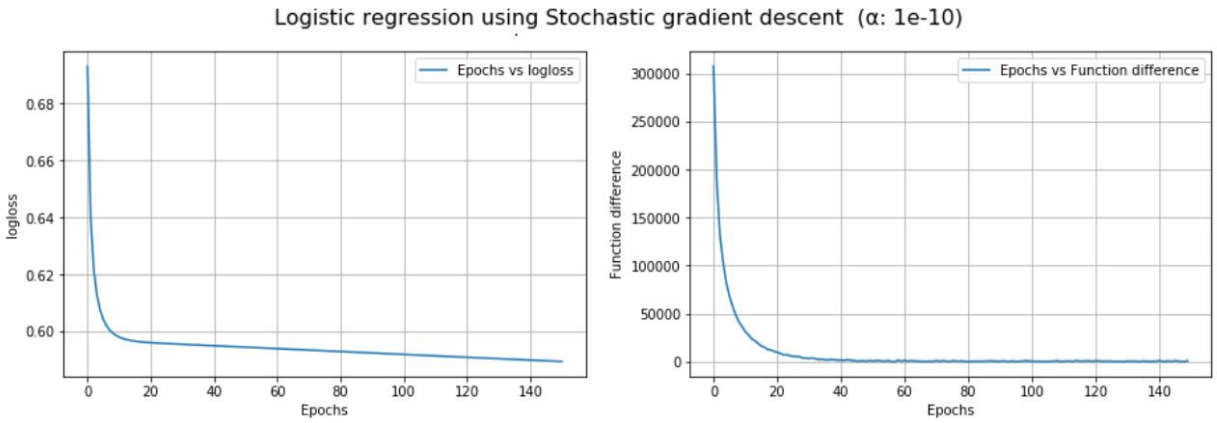
This function performs the linear classification algorithm with Stochastic Gradient Descent method. This method shuffles the enter data and picks a row randomly from the data and learns the value of beta. This constantly updates the value of beta until the entire data is utilised for learning. The process is repeated for 50 epochs and at the end of every epoch, the likelihood of a function and logarthmic loss are found and made into a list for the plotting of graph.

```
def logreg_sgd(Y_train,Y_test,X_train,X_test,alpha):
    beta = np.zeros(X_data.shape[1] + 1)
    ## Step length type is set to None for constant step length
    logloss_graph,diff_graph,beta = stochasticlogisticregression(alpha,beta,Y_train,Y_test,X_train,X_test,"None")
```

**Output:**

```
β:  [-1.41840276e-05 -5.71177970e-04 -5.48258048e-04 -4.44373964e-05
 -3.53001076e-04 -3.71983202e-06 -1.51705514e-06 -3.67678146e-06
 -5.60310307e-07 -3.18738859e-07 -2.49831517e-06 -3.13015419e-07
 -5.77028778e-07 -1.49257513e-06 -1.33753124e-07 -2.54693507e-06
 -4.60814261e-07 -8.87048524e-08 -1.53748836e-06 -9.49058457e-06
 -3.15595465e-06 -2.19833217e-06 -7.94415749e-06 -3.46460808e-06
 -5.76929829e-07 -1.38507139e-05 -3.33313676e-07 -8.09396292e-06
 -7.93008870e-07 -5.29705579e-06 -1.28666151e-06 -4.19447715e-07
  4.59809346e-07 -1.29377277e-05]

Logloss: 0.589407706463278
```




As we can see from these graphs plotted between iteration and logloss and function difference for a step length of 1e-10, we get a log loss of 0.589 and the function is converging near to 0 which with more iterations will be completely converged. The logarthmic loss is slightly more than that of the normal gradient descent method.

**Function name** : sgd_bold
**Parameter** : steplength, beta, training and testing data of X and Y

This function sends the input parameter to SGD function of type "Bold" and generates a model with continuously updating the value of the steplength. It controlls the step length for better values of beta and logarthmic loss.

```
def sgd_bold(Y_train,Y_test,X_train,X_test,alpha):
    beta = np.zeros(X_data.shape[1] + 1)
        ## Step length type is set to "Bold" for Bold driver step length
    logloss_graph,diff_graph,beta = stochasticlogisticregression(alpha,beta,Y_train,Y_test,X_train,X_test,"Bold")
```

**Function name** : bold_steplength
**Parameter** : alpha, new and old loglikelihood function values

This function is also for generating dynamic step length for generating a model with gradient descent. The step length is updated depending on the likelihood function values.

If the new function value is lessthan the old function value then the step length is incremented and vice versa.

$$\mu = \mu . \mu^{+}$$
$$\mu = \mu . \mu^{-}$$

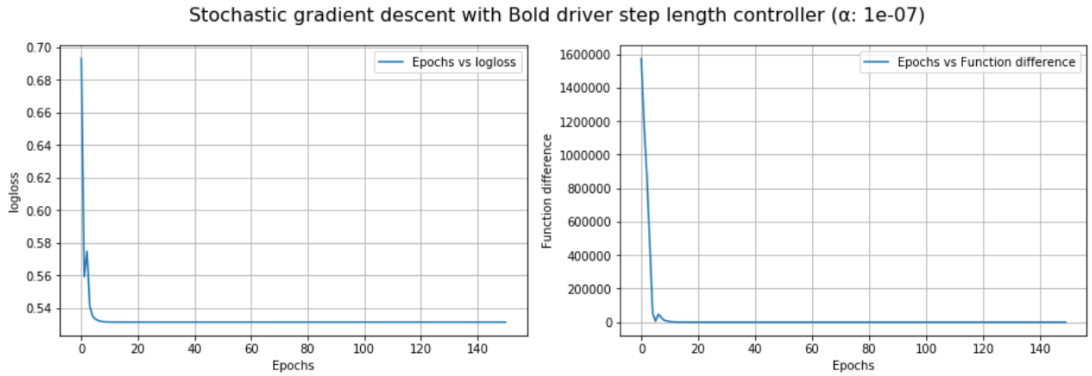
```
def bold_steplength(alpha,f_new,f_old):

    alpha_plus = 1.1
    alpha_minus = 0.5
    if f_new < f_old:
        alpha = alpha * alpha_plus
    else:
        alpha = alpha * alpha_minus
    return alpha
```

**Output:**

```
β:  [-1.62237303e-04 -6.47327294e-03 -4.04754868e-04 -5.20158071e-04
 -2.11374419e-03 -2.00437232e-05 -1.70818979e-05 -4.27788703e-05
 -6.53294919e-06 -3.56958009e-06 -2.74708815e-05 -2.75061061e-06
 -6.91365484e-06 -1.77248975e-05 -1.63324562e-06 -2.94701570e-05
 -5.37798645e-06 -9.32571657e-07 -1.70827415e-05 -1.09128695e-04
 -3.60258663e-05 -2.48869029e-05 -9.21492411e-05 -3.85101357e-05
 -6.69102292e-06 -1.58182329e-04 -4.05497360e-06 -8.87880583e-05
 -8.81693960e-06 -6.46323048e-05 -1.07418735e-05 -3.39070694e-06
  7.46121708e-06 -1.55565939e-04]

Logloss: 0.5312012305941279
```




From these graphs plotted between iteration and logloss and function difference for a step lengt of 1e-7, we get a log loss of 0.531 and the function is converging near to 0 which with more iterations will be completely converged. The logarthmic loss is compartively better than using constant step length methods.

**Function name** : sgd_adagrad
**Parameter** : steplength, beta, training and testing data of X and Y

This function sends the input parameter to SGD function of type "Adagrad" and generates a model with continuously updating the value of the steplength. It controlls the step length by dividing the previous step length by square root of the history of gradient squares.

```
def sgd_adagrad(Y_train,Y_test,X_train,X_test,alpha):
    beta = np.zeros(X_data.shape[1] + 1)
    ## Step length type is set to "Adagrad" for Adaptive gradient step length
    logloss_graph,diff_graph,beta = stochasticlogisticregression(alpha,beta,Y_train,Y_test,X_train,X_test,"Adagrad")
```

**Function name** : adagrad_steplength
**Parameter** : alpha,beta,training value of x and y and history of gradients.

This function performs the adaptive step length for linear classification using stochastic gradient descent or ascent problems. This function keeps in track of the previous history values and returns the new beta.

$$\beta_{new} = \beta_{old} - \frac{\mu}{\sqrt{h}} G$$
$$G = X^T(y - p)$$
$$h = h + G * G$$

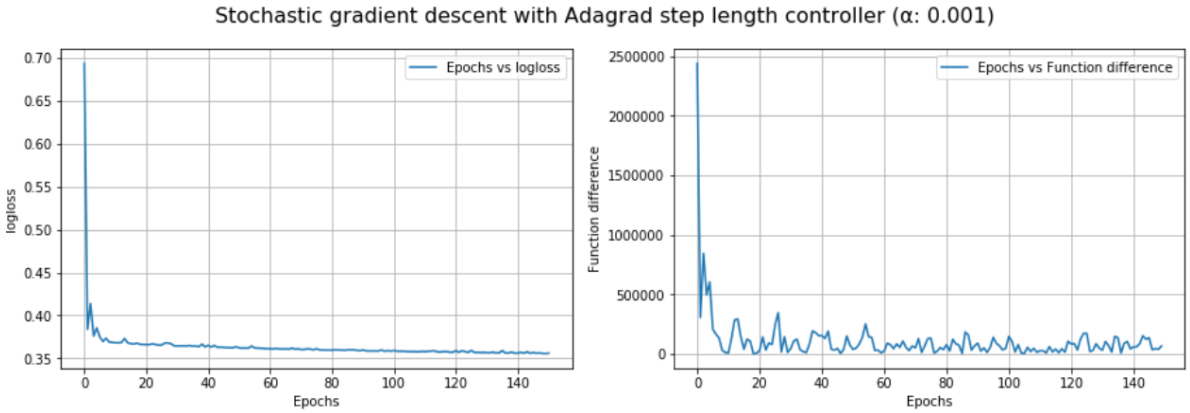
```
def adagrad_steplength(alpha,beta,xtr,ytr,history):
    error =  ytr - 1/(1 + np.exp(-np.dot(beta,xtr)))
    grad = - (xtr.T * error)
    grad_sq = grad * grad
    for i in range(0,len(history)):
        history[i] += float(grad_sq[i]) + 1e-10
        beta[i] = beta[i] - float((alpha/np.sqrt(history[i]))
                                  * grad[i])

    return beta,history
```

**Output:**

```
β:  [-8.76154564e-02 -3.23481361e-02  2.13647465e-05 -8.48501535e-02
  9.68757037e-04  4.05933409e-02 -2.75890673e-02 -1.33415565e-01
 -3.10174611e-02  1.19308874e-02  2.08474373e-03  1.05858802e-01
 -3.05846311e-02 -7.78637681e-02  1.11122538e-03 -6.13711882e-02
 -4.23626913e-02  8.59412159e-03  4.01744779e-02 -1.12712851e-01
 -3.75614338e-02 -1.32592741e-02 -1.04889750e-01 -8.28892435e-03
 -2.56570507e-02 -8.89536785e-02  2.69864546e-03 -3.85647965e-03
  2.10264256e-02 -2.24955857e-01 -5.78348287e-02 -3.39685349e-03
  1.62653121e-01 -1.45081701e-01]

Logloss: 0.3556536482051492
```




From these graphs plotted between iteration and logloss and function difference for a step lengt of 0.001 we get a logarthmic loss of 0.356 which is the best of all th step length controllers. The logarthmic loss is much better than using constant step length methods.

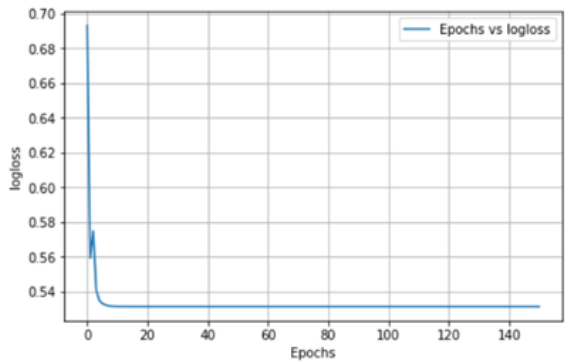
## Comparision between Adagrad and Bold driver step length controllers:

**Algorithm Comparision:**

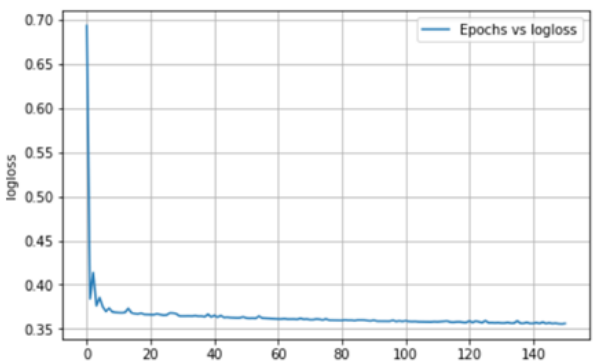
In bold driver step length algorithm, the step length is controlled only at the end of every epoch. Whereas in Adagrad, the step length is controlled while iterating every data row within the epoch. The step length within Bold driver is incremented when the new value of likelihood function is more than the old value and vice versa. In Adagrad, the step length is adapted to the change, that it is controlled by the square root of the history of previous gradient squares. For every epoch, the history value is retained and used for the next epoch.

**Logarthmic Loss graph comparision:**

Bold driver step length controller




Adagrad step length controller




It can be seen that the logarthmic loss reduces gradually near to a particular value and tries to reduce further in a very small rate. The logloss of Bold and Adagrad is 0.531 and 0.35. After 5 epochs, the logarthmic

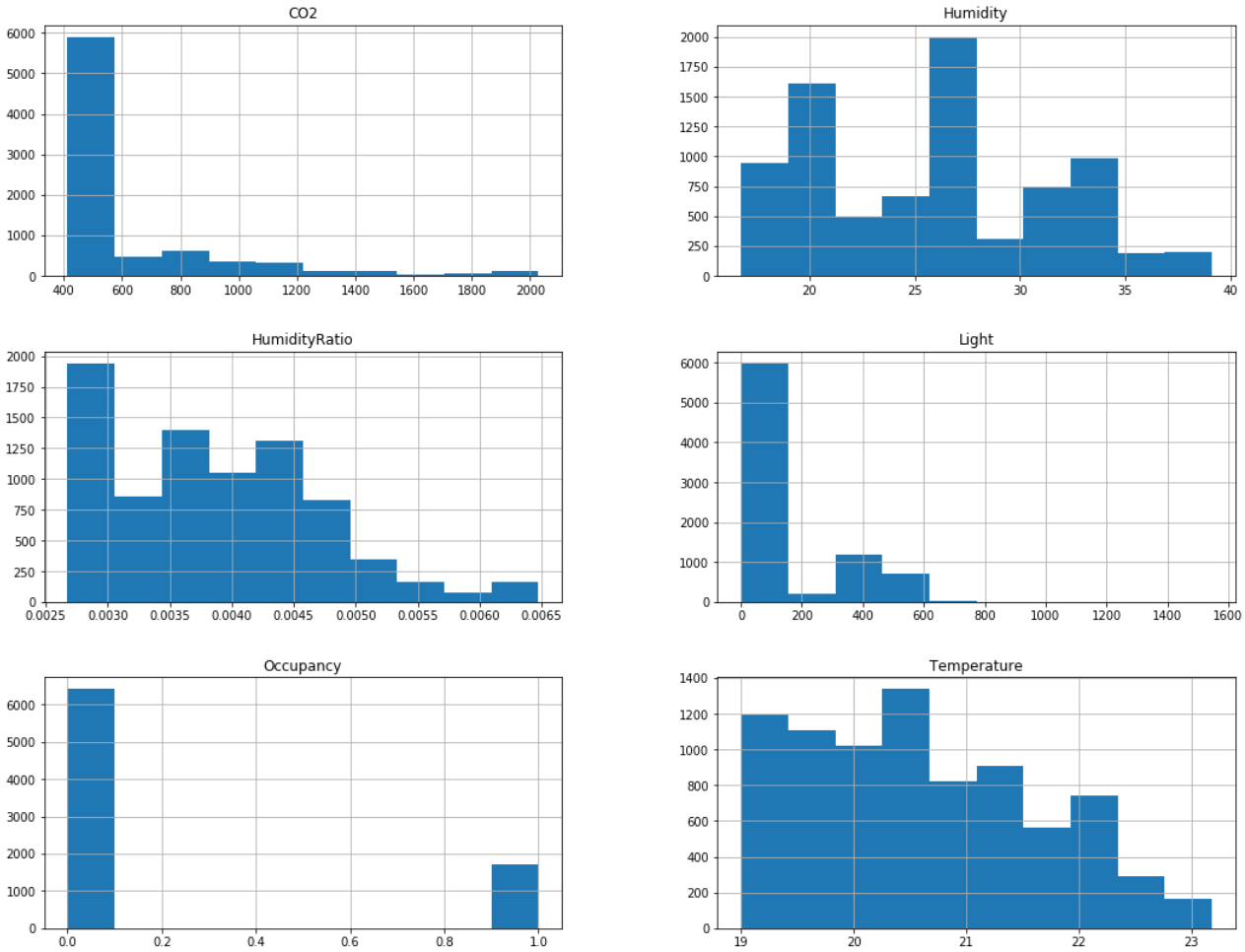
loss in Bold driver remains to be constant but in the Adagrad, it tries to adapt itself as the number of epochs which means that the logarithmic loss decreases as the epochs increases. Also, the model fit is good with Adagrad step length controller. So, Adagrad is better than Bold.

## Occupancy dataset:

Correlation between occupancy and other variables for the occupancy data

| | date | Temperature | Humidity | Light | CO2 | HumidityRatio | Occupancy |
|---|---|---|---|---|---|---|---|
| **0** | -0.0982 | 0.5328 | 0.1292 | 0.8046 | 0.6567 | 0.2558 | NaN |

Visualization of the numercial columns in the occupancy dataset



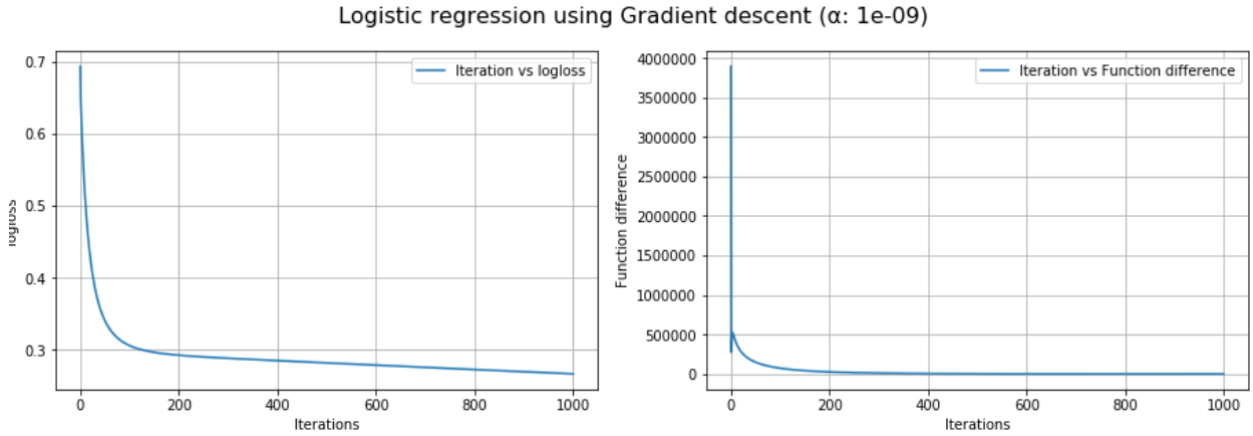

From the correlation values and data visualization, except the date column, all other column seem to be playing a vital role in determining the occupancy of the data. Hence all the columns are used.

| | Temperature | Humidity | Light | CO2 | HumidityRatio | Occupancy |
|---|---|---|---|---|---|---|
| **7904** | 20.100000 | 33.045 | 0.0 | 449.000000 | 0.004810 | 0 |
| **4227** | 21.033333 | 19.500 | 24.0 | 438.000000 | 0.002998 | 0 |
| **4371** | 20.200000 | 23.000 | 0.0 | 432.000000 | 0.003361 | 0 |
| **6198** | 19.390000 | 27.100 | 0.0 | 462.000000 | 0.003769 | 0 |
| **5950** | 19.390000 | 27.390 | 0.0 | 442.666667 | 0.003809 | 0 |

**Logistic regression with Gradient Descent:**

```
β:  [-5.92751926e-04 -1.20274379e-02 -1.13470530e-02  1.50360574e-02
 -3.89001645e-03 -1.59733774e-06]

Logloss: 0.2665711485375253
```

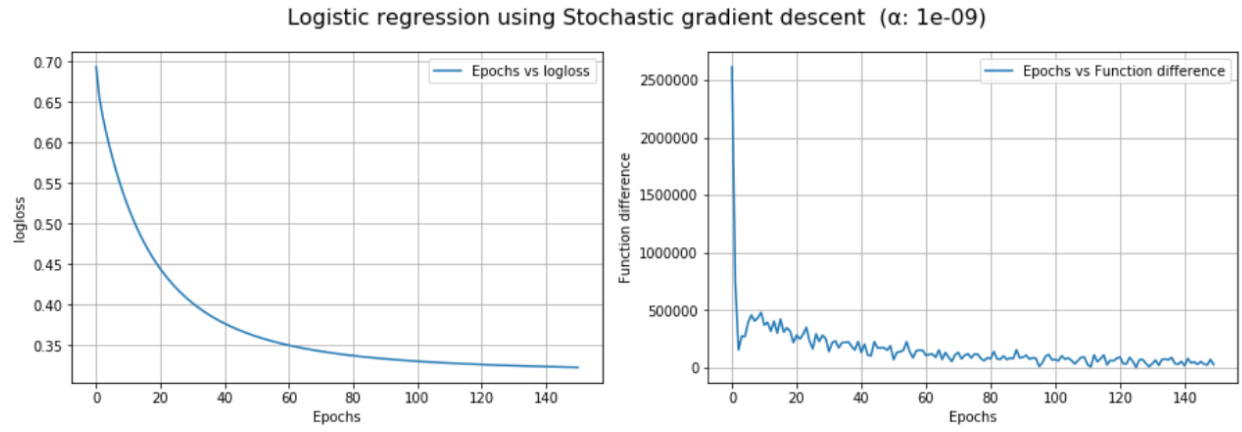



As we can see from these graphs plotted between iteration and logloss and function difference for a step length of 1e-9, I'm getting a log loss of 0.266 and the function is converging near to 0 which with more iterations will be completely converged.

**Logistic regression with Stochastic Gradient Descent:**

```
β:  [-1.16358809e-04 -2.31634743e-03 -2.42256106e-03  1.36452935e-02
 -4.17530366e-03 -3.35184733e-07]

Logloss: 0.3228199537752271
```

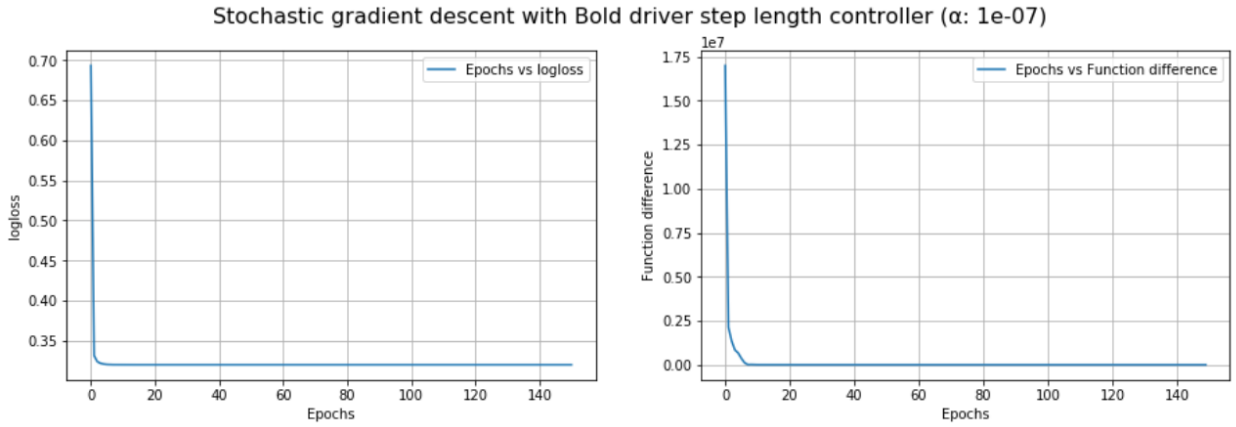



As we can see from these graphs plotted between iteration and logloss and function difference for a step length of 1e-9, I'm getting a log loss of 0.323 and the function is converging near to 0 which with more iterations will be completely converged. The model generated is not as good as the model generaged by the normal gradient descent method.

**Logistic regression with Stochastic Gradient Descent using Bold Driver step length:**

```
β:  [-1.45633777e-04 -2.90956416e-03 -2.98339937e-03  1.45988147e-02
 -4.38584361e-03 -4.14015998e-07]

Logloss: 0.31976859094732324
```

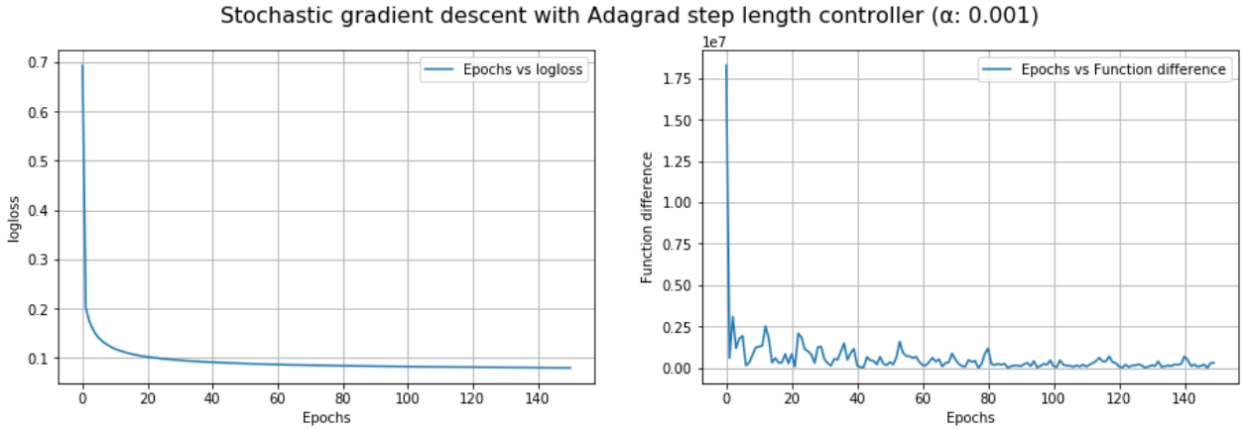



From the graph, it's evident that when the step length is controlled by a controller, the Bold Driver in this case, the learning gets better with the SGD. Here, the logarthmic loss is 0.31 for an initail step length of 1e-07 and the function is converging near to 0 which with more iterations will be completely converged. The model generated is good as compared to the model generated by the constant step length.

**Logistic regression with Stochastic Gradient Descent using Adagrad step length:**

```
β:  [-0.2073459  -0.20263141 -0.11520504  0.01412297  0.00382223 -0.10921685]

Logloss: 0.08010139130503528
```




From the graph, it can be seen that when the step length is controlled by an Adagrad adaptive steplength controller, the learning gets better with the SGD. Here, the logarthmic loss is as low as 0.08 for an initail step length of 0.001. In this, the step length is adaptively controlled depending on the gradient value and hence the better beta value is achieved.

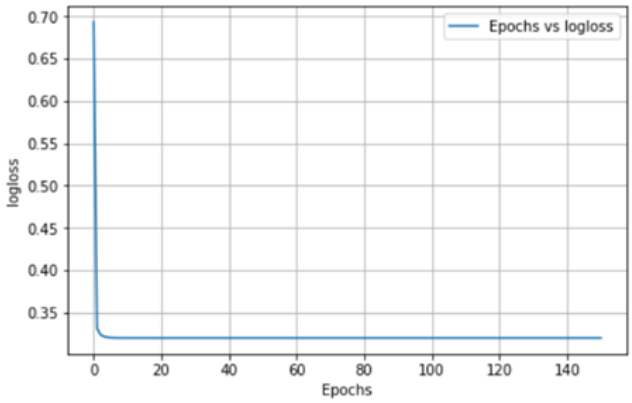
## Comparision between Adagrad and Bold driver step length controllers:

**Algorithm Comparision:**

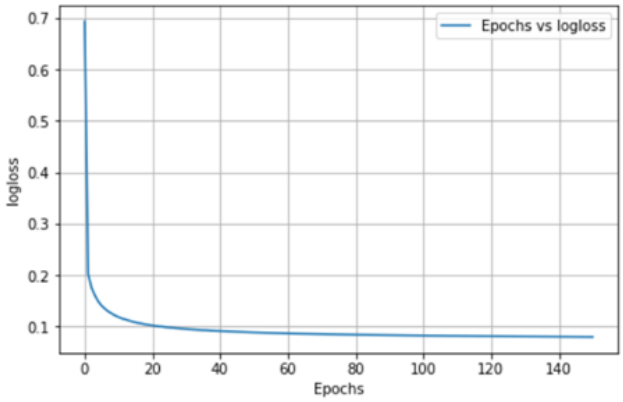
In bold driver step length algorithm, the step length is controlled only at the end of every epoch. Whereas in Adagrad, the step length is controlled while iterating every data row within the epoch. The step length within Bold driver is incremented when the new value of likelihood function is more than the old value and vice versa. In Adagrad, the step length is adapted to the change, that it is controlled by the square root of the history of previous gradient squares. For every epoch, the history value is retained and used for the next epoch.

**Logarthmic Loss graph comparision:**

Bold driver step length controller




Adagrad step length controller




It can be seen that the logarthmic loss reduces gradually near to a particular value and tries to reduce further in a very small rate. The logloss of Bold and Adagrad 0.31 and 0.08. The log loss after 3 epochs in Bold driver step length seem to be constant but in the case of Adagrad, it keeps on decreasing at an adaptive rate. The model generated by SGD using Adagrad step length fits well than the one generated by Bold driver step length. The values of beta is also good in the case of Adagrad. Hence, in this case learning is better in Adagrad.