

Python TASK 4 (All exercises in one)

Name: Jatin Karthik

Matriculation no.:313301

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
```

EXERCISE 0

In [2]:

```
#reading the data file as csv
#naming the columns
ttt=pd.read_csv('tic-tac-toe-data',delimiter=',')
ttt.columns=['top-left-square','top-middle-square','top-right-square','middle-left-square','middle-middle-square','middle-right-square','bottom-left-square','bottom-middle-square','bottom-right-square','Classification']
ttt
```

Out [2]:

	top-left-square	top-middle-square	top-right-square	middle-left-square	middle-middle-square	middle-right-square	bottom-left-square	bottom-middle-square	bottom-right-square	Classification	
0	x	x	x	x	x	o	o	o	o	positive	
1	x	x	x	x	x	o	o	o	x	positive	
2	x	x	x	x	x	o	o	o	b	positive	
3	x	x	x	x	x	o	o	b	o	positive	
4	x	x	x	x	x	o	o	b	b	o	positive
...	...	...	...	...	...	...	...	...	...	...	
952	o	x	x	x	x	o	o	o	x	x	negative
953	o	x	x	x	x	o	o	x	o	x	negative
954	o	o	x	o	x	o	x	x	o	x	negative
955	o	x	o	o	x	x	x	o	x	negative	
956	o	o	o	x	x	o	o	x	x	negative	

957 rows × 10 columns

1. Convert any non-numeric values to numeric values. For example you can replace a country name with an integer value or more appropriately use hot-one encoding. (Hint: use hashmap (dict) or pandas.get\_dummies). Please explain your solution.

def encode\_and\_bind(original\_df, feature\_to\_encode):  
dummies = pd.get\_dummies(original\_df[[feature\_to\_encode]])  
original\_df = pd.concat([original\_df, dummies], axis=1)  
return(original\_df)  
  
A1=encode\_and\_bind(ttt, 'top-left-square')  
A2=encode\_and\_bind(A1, 'top-middle-square')  
A3=encode\_and\_bind(A2, 'top-right-square')  
A4=encode\_and\_bind(A3, 'middle-left-square')  
A5=encode\_and\_bind(A4, 'middle-middle-square')  
A6=encode\_and\_bind(A5, 'middle-right-square')  
A7=encode\_and\_bind(A6, 'bottom-left-square')  
A8=encode\_and\_bind(A7, 'bottom-middle-square')  
A9=encode\_and\_bind(A8, 'bottom-right-square')  
  
##Make Classification as a numerical positive or negative  
A9.loc[A9['Classification'] == 'positive', 'Classification\_num'] = '1'  
A9.loc[A9['Classification'] == 'negative', 'Classification\_num'] = '0'  
A9=A9.drop(['Classification'],axis=1)  
A9

Out [3]:

	top-left-square	top-middle-square	top-right-square	middle-left-square	middle-middle-square	middle-right-square	bottom-left-square	bottom-middle-square	bottom-right-square	top-left-square_b	...	bottom-left-square_b	bottom-middle-square_b	bottom-left-square_x	bottom-middle-square_b	bottom-middle-square_o	bottom-middle-square_x	bottom-middle-square_b	bottom-middle-square_o	bottom-middle-square_x	bottom-right-square_b	bottom-right-square_o	bottom-right-square_x	Classification_num
0	x	x	x	x	x	o	o	o	x	o	o	o	o	1	0	0	0	0	0	1	0	1	0	1
1	x	x	x	x	x	o	o	o	o	x	o	o	o	1	0	0	1	0	0	1	0	0	1	1
2	x	x	x	x	x	o	o	o	b	b	o	o	o	1	0	0	1	0	0	1	0	0	1	1
3	x	x	x	x	x	o	o	b	o	b	o	o	o	1	0	0	0	1	0	1	0	0	1	1
4	x	x	x	x	x	o	o	b	b	b	o	o	o	1	0	0	1	0	0	1	0	1	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
952	o	x	x	x	x	o	o	o	x	x	o	o	o	1	0	0	0	0	1	1	0	0	1	0
953	o	x	o	x	x	o	x	o	x	o	o	o	o	1	0	1	0	1	0	0	0	0	1	0
954	o	x	o	o	x	x	x	o	x	o	o	o	1	0	1	0	1	0	0	0	0	1	0	0
955	o	x	o	o	x	x	x	o	x	o	o	o	1	0	1	0	1	0	0	0	0	1	0	0
956	o	o	x	x	x	o	o	x	x	x	o	o	1	0	0	0	1	0	0	1	0	0	1	0

957 rows × 37 columns

Explanation- I used pd.getdummies here to have a substitute column for each of the pre-existing non-numeric columns. This is done because the classification algorithm cannot interpret the non-numeric(in this case x,o and b) as it is . pd.getdummies helps us to create a 3 subsequent columns for each column (because 3 possible values exist) consisting of subsequent 1's wherever the class has that specific value come up.

Since each row represents one whole tic-tac-toe game with three possibilities , so there will be 3\*3=27 new columns added when hot encoding is used.

2. This dataset is unbalanced, (show how we can confirm this). Explain what is stratified sampling and implement a stratified sampler.

In [4]:

```
P=(ttt['Classification']).value_counts().value_counts()  
P  
#print(ttt['Classification'] == 'positive')  
#print(ttt['Classification'] == 'negative')
```

Out [4]:

```
625    1  
332    1  
Name: Classification, dtype: int64
```

This shows us that there are 625 positive classes and 332 negative ones, which clearly shows that the dataset is imbalanced because they dont have similar/equal number of classification results for the machine to learn both the classifications properly

Balanced Dataset: — If the number of positive values and negative values is approximately same.

Unbalanced Dataset: — If there is the very high difference between the positive values and negative values.

Oversampling is when one class is randomly chosen more than the other one by a huge margin and that repetitive features are used from that class. Undersampling on the other hand is the opposite of oversampling.

Out [5]:

```
#making a df of only the numerical values by dropping the rest  
final=A9.drop(['top-left-square','top-middle-square','top-right-square','middle-left-square','middle-middle-square','middle-right-square','bottom-left-square','bottom-middle-square','bottom-right-square'],axis=1)  
final
```

Out [5]:

	top-left-square_b	top-left-square_o	top-left-square_x	top-middle-square_b	top-middle-square_o	top-middle-square_x	top-right-square_b	top-right-square_o	top-right-square_x	middle-left-square_b	...	bottom-left-square_b	bottom-left-square_o	bottom-left-square_x	bottom-middle-square_b	bottom-middle-square_o	bottom-middle-square_x	bottom-middle-square_b	bottom-middle-square_o	bottom-middle-square_x	bottom-right-square_b	bottom-right-square_o	bottom-right-square_x	Classification_num
0	o	o	1	o	o	0	1	o	0	1	o	o	1	o	0	1	0	0	0	1	0	1	0	1
1	o	o	1	o	o	1	o	o	1	o	o	1	o	o	1	0	0	1	0	0	0	1	0	1
2	o	o	1	o	o	1	o	o	1	o	o	1	o	o	1	0	0	1	0	0	1	0	0	1
3	o	o	1	o	o	1	o	o	1	o	o	1	o	o	1	0	0	1	0	0	1	0	0	1
4	o	o	1	o	o	1	o	o	1	o	o	1	o	o	1	0	0	1	0	0	1	0	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
952	o	1	o	o	o	1	o	o	1	o	o	o	1	o	0	0	1	0	0	0	1	0	0	1
953	o	1	o	o	o	1	o	1	o	o	o	o	1	o	1	0	1	0	0	0	0	1	0	0
954	o	1	o	o	o	1	o	1	o	o	o	o	1	o	1	0	1	0	0	0	0	1	0	0
955	o	1	o	o	o	1	o	1	o	o	o	o	1	o	1	0	1	0	0	0	0	1	0	0
956	o	1	o	o	1	o	o	o	1	o	o	o	1	o	0	0	1	0	0	0	1	0	0	0

957 rows × 28 columns

In statistical surveys, when subpopulations within an overall population vary, it could be advantageous to sample each subpopulation (stratum) independently. Stratification is the process of dividing members of the population into homogeneous subgroups before sampling. The strata should define a partition of the population. That is, it should be collectively exhaustive and mutually exclusive: every element in the population must be assigned to one and only one stratum. Then simple random sampling is applied within each stratum. The objective is to improve the precision of the sample by reducing sampling error. It can produce a weighted mean that has less variability than the arithmetic mean of a simple random sample of the population.

Source- [https://en.wikipedia.org/wiki/Stratified\\_sampling](https://en.wikipedia.org/wiki/Stratified_sampling)

In [6]:

```
#stratification  
final['Stratify'] = final['Classification_num']  
final['Stratify'].value_counts() / len(final) #sort_values(ascending=False)
```

Out [6]:

```
0    0.653883  
1    0.346117  
Name: Stratify, dtype: float64
```

Stratification by simply using groupby attribute w.r.t the target column and sampling it by fraction

In [11]:

```
final.groupby('Classification_num', group_keys=True).apply(lambda x: x.sample(frac=0.5, random_state=3116))
```

Out [11]:

	top-left-square_b	top-left-square_o	top-left-square_x	top-middle-square_b	top-middle-square_o	top-middle-square_x	top-right-square_b	top-right-square_o	top-right-square_x	middle-left-square_b	...	bottom-left-square_b	bottom-left-square_o	bottom-left-square_x	bottom-middle-square_b	bottom-middle-square_o	bottom-middle-square_x	bottom-middle-square_b	bottom-middle-square_o	bottom-middle-square_x	bottom-right-square_b	bottom-right-square_o	bottom-right-square_x	Classification_num	Stratify
0	943	0	0	1	o	o	0	1	o	0	1	o	0	1	o	0	1	0	0	0	0	1	0	0	0
927	1	o	0	1	o	o	1	o	0	0	1	o	o	1	o	0	1	0	0	0	1	0	0	0	0
774	0	1	o	0	o	0	1	o	1	o	1	o	o	0	1	1	0	0	0	0	1	0	0	0	0
826	o	1	o	o	0	1	o	o	1	o	o	o	o	0	1	0	0	1	1	0	0	0	0	0	0
852	o	1	o	1	o	1	o	o	0	1	o	o	o	1	o	o	1	0	0	1	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1	485	1	o	0	o	0	1	o	0	1	o	0	1	o	o	1	0	0	0	1	1	0	0	0	1
409	o	1	o	o	0	1	o	1	o	0	0	o	o	1	o	0	0	1	1	0	0	0	0	1	1
3	o	o	1	o	o	0	1	o	0	1	o	0	1	o	o	0	0	1	0	1	0	0	0	1	1
576	1	o	0	o	0	1	o	1	o	0	0	o	o	1	o	0	0	1	0	1	0	1	0	1	1
336	o	1	o	1	o	o	0	1	o	0	1	o	1	o	o	0	0	1	0	1	0	1	0	1	1

478 rows × 29 columns

OR-----

In [8]:

```
def stratify_data(df_data, stratify_column_name, stratify_values, stratify_proportions, random_state=None):  
df_stratified = pd.DataFrame(columns = df_data.columns) # Create an empty DataFrame with columns  
  
pos = -1  
for i in range(len(stratify_values)):  
pos += 1  
if pos == len(stratify_values) - 1:  
ratio_len = len(df_data) - len(df_stratified)  
else:  
ratio_len = int(len(df_data) * stratify_proportions[i]) # Calculate the number of rows to match the desired proportion  
  
df_temp = df_data[df_data[stratify_column_name] == stratify_values[i]] # Filter the source data based on the currently selected stratify value  
df_temp = df_temp[sampled.replace(True, np.nan).len, random_state=random_state] # Sample the filtered data using the calculated ratio  
df_stratified = pd.concat([df_stratified, df_temp]) # Add the sampled / stratified datasets together to produce the final result  
  
return df_stratified
```

The above code is referenced from the references mentioned in the end, but I have used the group by notation.

In [9]:

```
#stratify_values = ['positive', 'negative']  
stratify_values = ['1', '0']  
stratify_proportions = [0.58, 0.58]  
stratified = stratify_data(final, 'Stratify', stratify_values, stratify_proportions, random_state=3116)  
stratified
```

Out [9]:

	top-left-square_b	top-left-square_o	top-left-square_x	top-middle-square_b	top-middle-square_o	top-middle-square_x	top-right-square_b	top-right-square_o	top-right-square_x	middle-left-square_b	...	bottom-left-square_b	bottom-left-square_o	bottom-left-square_x	bottom-middle-square_b	bottom-middle-square_o	bottom-middle-square_x	bottom-middle-square_b	bottom-middle-square_o	bottom-middle-square_x	bottom-right-square_b	bottom-right-square_o	bottom-right-square_x	Classification_num	Stratify
274	o	o	1	1	o	o	1	o	0	0	o	o	1	o	0	1	0	0	0	1	0	0	1	0	1
290	o	o	1	1	o	o	1	o	0	1	o	o	o	o	0	1	0	0	0	0	1	0	0	1	1
12	o	o	1	1	o	1	o	0	1	o	o	1	o	o	0	1	0	0	1	0	0	1	0	1	1
219	o	o	1	o	1	o	1	o	0	1	o	o	1	o	1	0	0	0	0	0	0	1	1	1	1
379	o	1	o	0	1	o	0	0	1	o	o	0	0	1	o	0	0	1	0	0	0	1	1	1	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
956	o	1	o	0	1	o	0	0	1	o	o	1	o	o	0	1	0	0	0	1	0	0	1	0	0
641	o	o	1	o	o	1	o	1	o	0	o	1	o	1	o	0	0	0	0	0	1	0	0	0	0
908	1	o	0	0	0	1	1	o	0	o	o	1	o	0	1	0	0	0	0	1	0	0	0	0	0
945	o	o	1	o	1	o	0	0	1	o	o	1	o	o	0	1	0	0	1	0	1	0	0	0	0
676	o	o	1	o	1	o	0	0	0	1	o	o	0	0	1	0	0	1	0	1	0	0	0	0	0

957 rows × 29 columns

3. Split the data into a train(80%) and test(20%).

In [12]:

```
training_data = stratified.sample(frac=0.8, random_state=3116) #80% Train  
testing_data = stratified.drop(training_data.index) #20% test  
testing_data = stratified.sample(frac=0.2, random_state=3116)  
  
print(f"%o. of training examples: (training_data.shape[0])")  
print(f"%o. of testing examples: (testing_data.shape[0])")  
  
x=np.array(training_data, dtype=int)  
xtest=np.array(testing_data, dtype=int)  
  
y=np.array(y, dtype=int)  
ytest=np.array(y, dtype=int)  
print(np.shape(ytest))  
  
No. of training examples:768  
No. of testing examples: 191  
(191, 1)
```

Exercise 1: Logistic Regression with Gradient Descent

In [1]:

```
#defining the sigmoid function  
def sigmoid(f):  
S= 1/(1+np.exp(-1*f))  
return S
```

In [36]:

```
def minimise_GA(X, y, Xtest,ytest,mu,iterations,e):  
(n, n) = np.shape(X)  
  
loss_history = [0] * iterations #absolute loss history list  
log_history = [0] * iterations #log loss history list  
loss_history2 = [0] * iterations  
B=np.zeros([27,1], dtype=int)  
B2=np.zeros([27,1], dtype=int)  
y_hat=X.dot(B)  
y_hat2=X.dot(B2)  
y_hat2=X.dot(B2) - np.log(1+np.exp(y_hat)) #L_cond defined in slides  
#np.sum((y.T).dot(y_hat2) - np.log(1+np.exp(y_hat2)))  
  
# Two loops to run because we are implementing SGD where epochs are run  
for iteration in range(iterations):  
y_cap=sigmoid(y_hat)  
y_cap2=sigmoid(X.dot(B2))  
for j in range(n):  
error=y-y_cap  
gradient=(np.dot(X.T,error))  
B = B + mu*gradient  
B2=B2 - mu*gradient  
  
#l=np.sum((y.T).dot(X.dot(B)) - np.log(1+np.exp(y_hat)))  
l_old=l  
l=np.sum((y.T).dot(X.dot(B)) - np.log(1+np.exp(y_hat)))  
#l=np.sum((y.T).dot(X.dot(B2)) - np.log(1+np.exp(y_hat2)))  
mu=steplength_bolddriver(1,l_old)  
y_cap2=sigmoid(X.dot(B2))  
loss=abs_loss(y_cap,y_cap2)  
loss2=abs_loss(l_old,l)  
loss_history[iteration]=loss  
loss_history2[iteration]=loss2  
  
z= Xtest.dot(B)  
y_pred=sigmoid(z)  
#print(ytest)  
log_loss=logloss(ytest,y_pred)  
log_history[iteration]=log_loss  
#if (1-l_old)<e:  
return B,loss_history,loss_history2,log_history
```

In [128]:

```
def logloss(y_true,y_pred):  
if this function, we will compute log loss ***  
log_loss = -((1/y_true * np.log(y_pred)) + (1-y_true) * np.log(1-y_pred)).mean()  
return log_loss
```

In [65]:

```
def Abs_loss(l_old,l): # Function for absolute loss i.e |(x-1)-f(x)|  
#np.absolute(l_old,l))  
return l
```

In [83]:

```
def steplength_bolddriver(l_old,l_new,mu_old=0.00001, mu_plus=0.5,mu_minus=0.0001):  
if (l-l_old)/l_new>0:  
mu=mu_old*mu_minus  
else:  
mu=mu_old*mu_plus  
#while(fun_x(X,y) + f_grad_sbd(mu,n) <= 0):  
#mu=mu_old*mu_minus  
return mu,n  
  
#sbd_mu=steplength_bolddriver(X,y,mu_old=0.00001)  
#print("The mu value which we get from stepsize bolddriver is",sbd_mu)  
#beta_loss_hist= minimise_GA(X,y, sbd_mu , iterations=100, e=1e-3)
```

You will use bolddriver as the weight controller. - In each iteration of the algorithm calculate |(xi-1) - f(xi)| and at the end of learning, plot it against iteration number i. Explain the graph. - In each iteration step also calculate logloss on test set (see <https://www.kaggle.com/wiki/LogarithmicLoss>), plot it against iteration number i. Explain the graph.

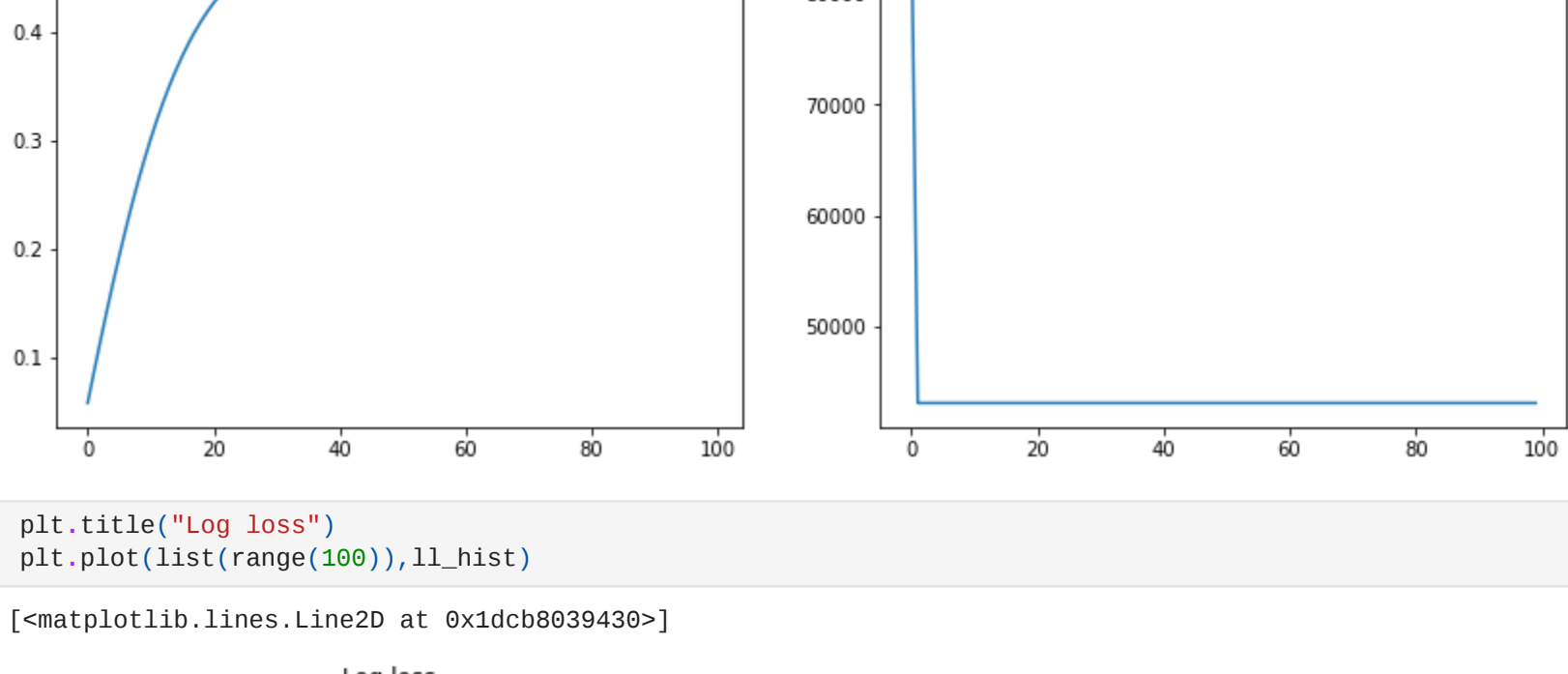
Out [85]:

```
w_loss_hist1,loss_hist2,ll_hist= minimise_GA(X,y, Xtest,ytest,mu=0.000001, iterations=100, e=1e-1)  
#print(ll_hist)
```

```
fig, (ax1,ax2) = plt.subplots(1, 2,figsize=(14,5))  
fig.suptitle('absolute loss Plot for 2 varities')  
ax1.plot(list(range(100)),loss_hist1)  
ax1.set_title('|f(x)-1|')  
ax2.plot(list(range(100)),loss_hist2)  
ax2.set_title('|y(x)-y(x-1)|')
```

Out [85]:

Text(0.5, 1.8, '|y(x)-y(x-1)|')



In [86]:

```
plt.title('Log loss')  
plt.plot(list(range(100)),ll_hist)
```

Out [86]:

```
[<matplotlib.lines.Line2D at 0x1cd8039430>]
```

Explanation- For Gradient Ascent, is used for finding the maximum point and here, we were asked to plot absolute loss of two previous iterations. As we can see the y(i) i.e the Lcond given in slides decreases rapidly within an iteration which means the GA has very less difference between its iterations. About the 2nd graph which is |f(x)-1|, this is the predicted value difference of every iteration, be. In my view, this graph hasd to decrease to 0, no. of iterations but I think the difference is 0.5 where it saturates and that is not what Dig. Talking about Log Loss, the ideal log loss for Logistic regression GA is the one where the log likelihood increases, so the log loss should automatically decrease which is what is happening but the trend differs with different values of alpha, alpha\_plus and minus i.e the step used.

Exercise 2: Implement Newton Algorithm for Logistic Regression

$$\beta'(0)$$