MACHINE LEARNING LAB - TUTORIAL 4

Juan Fernando Espinosa

303158

1. DATA PROCESSSING

IMPORT DATASETS

```
import pandas as pd
import numpy as np
%matplotlib inline
import math
import matplotlib.pyplot as plt
from google.colab import files
from google.colab import drive
```

```
drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 4/tic-tac-toe.data"
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
'/content/drive/My Drive/Colab Notebooks/LAB/tutorial 4/tic-tac-toe.data'
```

```
column_names_tic_tac = ['top-left-square','top-middle-square','top-right-square','middle-left-square','middle-middle-square','middle
missing_values = ['-','na','Nan','nan','n/a','?']
tic_tac = pd.read_csv('/content/drive/My Drive/Colab Notebooks/LAB/tutorial 4/tic-tac-toe.data', names=column_names_tic_tac, na_valu
```

```
tic_tac.head()
```

	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	Class
0	x	x	x	x	o	o	x	o	o	positive
1	x	x	x	x	o	o	o	x	o	positive
2	x	x	x	x	o	o	o	o	x	positive
3	x	x	x	x	o	o	o	b	b	positive
4	x	x	x	x	o	o	b	o	b	positive

> Check for missing values

```
# Check for missing or incongruent values
check = tic_tac.empty
print('checking missing values:',check)
print('Sum of errors:',tic_tac.isnull().sum())
```

```
checking missing values: False
Sum of errors: top-left-square         0
top-middle-square       0
top-right-square        0
middle-left-square      0
middle-middle-square    0
middle-right-square     0
bottom-left-square      0
bottom-middle-square    0
bottom-right-square     0
Class                   0
dtype: int64
```

1. 1. Convert any non-numeric values to numeric values.

The structure of the dataset is the following: 9 out of the 10 features in the dataset contains 3 attribute informtion:

- **x:** x player x has taken
- **o:** o player o has taken
- **b:** Blank

The last column of **Class** has two attribute information:

- **Positive**
- **Negative**

Therefore, instead of using **Dummies** and increase the number of columns, making hard to process it is easier to build a custom binary encoding. This binary code will work as follows:

- 0.1 = x
- 0.2 = o
- 0.3 = b
- 1 = positive
- 0 = negative

```
# Check general variables of columns.
print(tic_tac["Class"].value_counts())
print(tic_tac["bottom-right-square"].value_counts())
```

```
positive    626
negative    332
Name: Class, dtype: int64
x    418
o    335
b    205
Name: bottom-right-square, dtype: int64
```

```
# Replacing values and create a new dataframe.
tic_tac_encoded = tic_tac.replace(to_replace=['x','o','b', 'positive', 'negative'], value=[0.1,0.2,0.3,1,0])
tic_tac_encoded.head()
```

	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	Class
0	0.1	0.1	0.1	0.1	0.2	0.2	0.1	0.2	0.2	1
1	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.1	0.2	1
2	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.2	0.1	1
3	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.3	0.3	1
4	0.1	0.1	0.1	0.1	0.2	0.2	0.3	0.2	0.3	1

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

```
tic_tac_encoded.groupby(['top-left-square']).sum()
```

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	Class
0.1	78.1	76.9	78.1	74.2	84.9	76.9	84.9	75.4	295
0.2	62.2	57.9	62.2	54.6	57.9	57.9	57.9	58.9	189

```
print(tic_tac["Class"].value_counts())
```

```
positive    626
negative    332
Name: Class, dtype: int64
```

```
tic_tac_encoded.groupby(['Class']).sum()
```

	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
Class									
0	60.4	58.9	60.4	58.9	62.0	58.9	60.4	58.9	60.4
1	109.9	119.9	109.9	119.9	99.8	119.9	109.9	119.9	109.9

The dataset is unbalanced because there are more results for 'x' than for 'o' and 'b' in all the columns. Moreover, if we check our *categorical column Class*,there are a huge difference between **positive** and **negative** answers which makes the dataset unbalanced. In a perfect world a balanced dataset would be 50% - 50% each value. In addition, it is important to mention that a 60% - 40% distribution is good enough to work on.

> **Stratified sampling:** The data is splitted into homogeneous subgroups and the exactly number of instances in that homogeneous subgroup. Then, samples are extracted from this subgroups called *strata* to guarantee that the test set represents the whole population and then perform analysis to make inferences on the population of interest. Stratified sampling reduce the bias on test sets.

```
# First, I splitted the dataset in regards to the two main categorical classes:
# Positive and Negative.

tic_tac_positive = tic_tac_encoded.groupby(['Class']).get_group(1)
print(tic_tac_positive.head())
print(tic_tac_positive.shape)
tic_tac_negative = tic_tac_encoded.groupby(['Class']).get_group(0)
print(tic_tac_negative.head())
print(tic_tac_negative.shape)
```

```
   top-left-square  top-middle-square  ...  bottom-right-square  Class
0              0.1                0.1  ...                  0.2      1
1              0.1                0.1  ...                  0.2      1
2              0.1                0.1  ...                  0.1      1
3              0.1                0.1  ...                  0.3      1
4              0.1                0.1  ...                  0.3      1

[5 rows x 10 columns]
(626, 10)
     top-left-square  top-middle-square  ...  bottom-right-square  Class
626              0.1                0.1  ...                  0.2      0
627              0.1                0.1  ...                  0.2      0
628              0.1                0.1  ...                  0.2      0
629              0.1                0.1  ...                  0.3      0
630              0.1                0.1  ...                  0.2      0

[5 rows x 10 columns]
(332, 10)
```

```
# The next step is to select the optimal set of data by stratifying it.

tic_tac_positive_stratify = tic_tac_positive.sample(frac=0.65)
tic_tac_negative_stratify = tic_tac_negative.sample(frac=0.35)

# Finally I am going to concatenate both independent stratified dataframes.

frames = [tic_tac_positive_stratify, tic_tac_negative_stratify]
tic_tac_strafified = pd.concat(frames)
tic_tac_strafified.shape
```

```
(523, 10)
```

1. 3. Split the data into Train (80&) and test (20%)

```
tic_tac_train = tic_tac_strafified.sample(frac=0.8)
tic_tac_test = tic_tac_strafified.drop(tic_tac_train.index)
```

2. LOGISTIC REGRESSION

Algorithm learn Logreg GA with Gradient Ascent and Bold Driver

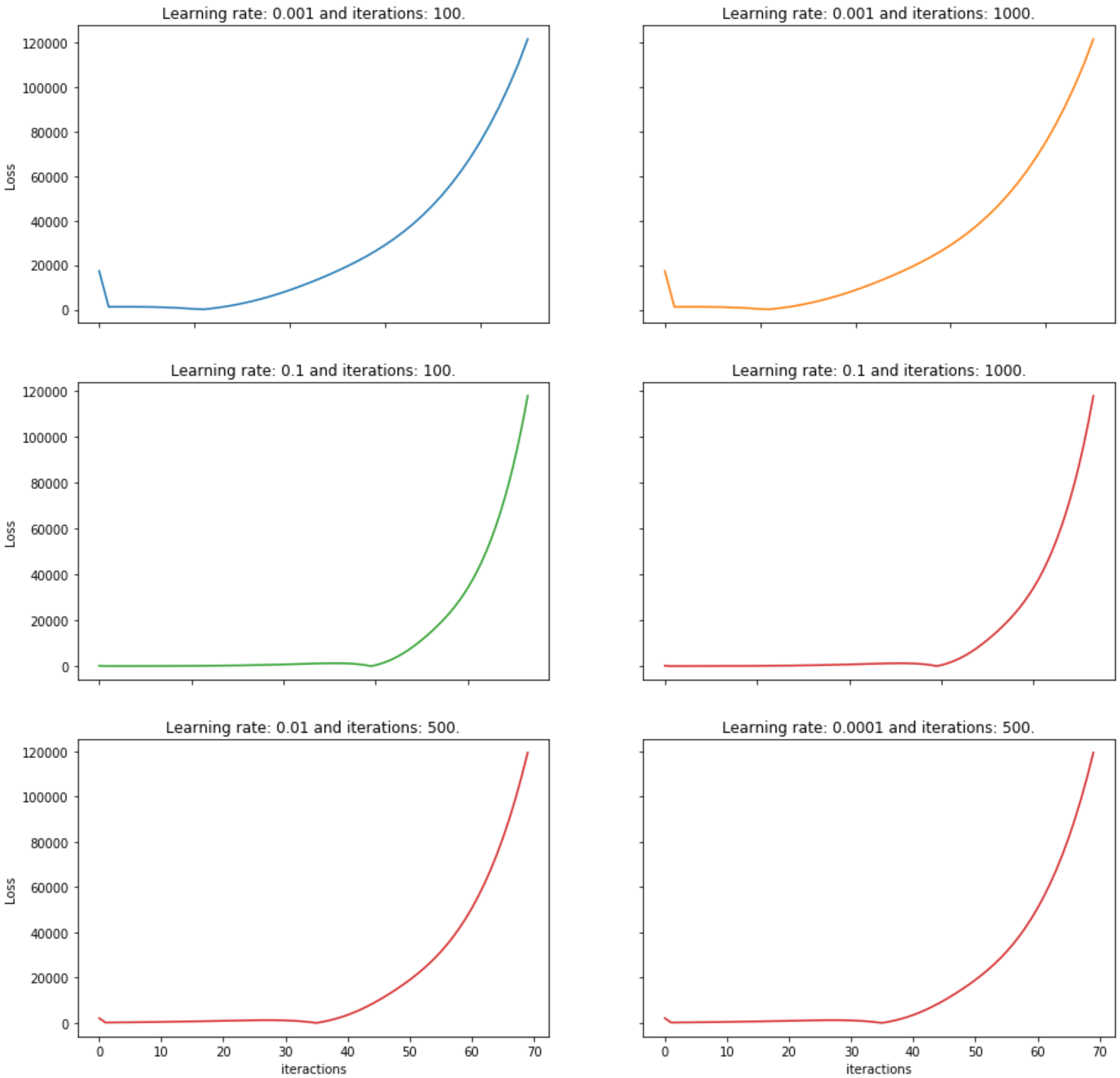
```
X = tic_tac_train.drop(['Class'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
y = tic_tac_train['Class'].values
y = np.reshape(y, (len(y),1))

# Sigma function
def sigma(X, beta):
  sigma = 1/(1+np.exp(-np.matmul(X, beta)))
  return sigma


def loss_general(X, y, beta):
  loss = np.sum(y@(np.log(sigma(X, beta))).T + (1-y)@(np.log(1-sigma(X, beta))).T)
  return loss

# Function for the Loss to plot it.
def loss_function(lossOld, loss):
  loss_total = abs(lossOld - loss)
  return loss_total

def bold_driver_step(X, y, u_old, u_plus, u_minus, beta, y_hat):
  u = u_old * u_plus
  while loss_general(X, y, beta) - loss_general(X, y, beta + u*np.dot(X.T, (y - y_hat))) <= 0:
    u * u_minus
  return u

def learn_logreg_GA(X, y, num_iters, u_old, u_plus, u_minus):
  loss_decrease = []
  X = X
  y = y
  n = X.shape[1]
  beta = np.zeros(n)
  beta = np.reshape(beta, (len(beta),1))
  loss = loss_general(X, y, beta)
  for i in range(num_iters):
    y_hat = (1 / 1 + np.exp(np.dot(X, -beta)))
    u = bold_driver_step(X, y, u_old, u_plus, u_minus, beta, y_hat)
    beta_hat = beta + u*np.dot(X.T, (y - X@beta))
    lossOld = loss
    loss = loss_general(X, y, beta_hat)
    loss_calculation = loss_function(lossOld, loss)
    loss_decrease.append(loss_calculation)

    u_old = u
  return beta_hat, loss_decrease




betas1, loss_decrease = learn_logreg_GA(X, y, 100, 0.0000001, 1.1, 0.5)
Beta_train = learn_logreg_GA(X, y, 100, 0.0000001, 1.1, 0.5)[0]
print('Betas', betas1, '\n', 'Loss', loss_decrease)
```

```
Betas [[0.44511378]
 [0.08103   ]
 [0.08530199]
 [0.07937633]
 [0.08461296]
 [0.07028112]
 [0.0854398 ]
 [0.07579337]
 [0.08612883]
 [0.08075439]]
 Loss [2.1984379470231943, 0.21983867824019399, 0.24182147170358803, 0.2660023187636398, 0.2926009776274441, 0.32185917189053725
```

```
a, b = learn_logreg_GA(X, y, 100, 0.001, 1.1, 0.5)
d, e = learn_logreg_GA(X, y, 1000, 0.001, 1.1, 0.5)
h, i = learn_logreg_GA(X, y, 100, 0.00001, 1.1, 0.5)
k, l = learn_logreg_GA(X, y, 1000, 0.00001, 1.1, 0.5)
m, n = learn_logreg_GA(X, y, 500, 0.0001, 1.1, 0.5)
o, p = learn_logreg_GA(X, y, 500, 0.0001, 1.1, 0.5)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), b)
axs[0, 0].set_title('Learning rate: 0.001 and iterations: 100.')
axs[0, 1].plot(range(1000), e, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.001 and iterations: 1000.')
axs[1, 0].plot(range(100), i, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.1 and iterations: 100.')
axs[1, 1].plot(range(1000), l, 'tab:red')
```

```
axs[1, 1].plot(range(1000), i, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.1 and iterations: 1000.')
axs[2, 0].plot(range(500), n, 'tab:red')
axs[2, 0].set_title('Learning rate: 0.01 and iterations: 500.')
axs[2, 1].plot(range(500), p, 'tab:red')
axs[2, 1].set_title('Learning rate: 0.0001 and iterations: 500.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='Loss')

for ax in axs.flat:
    ax.label_outer()
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:14: RuntimeWarning: divide by zero encountered in log

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:14: RuntimeWarning: invalid value encountered in matmul

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:9: RuntimeWarning: overflow encountered in exp
  if __name__ == '__main__':
```



Observations:

> As it is possible to appreciate in the graphs presented above a lower learning rate reaches convergence faster.

> The less iterations the model counts helps to reach convergence faster. In other words, since the stepsize is being adjusted, the iterations has low influence on the final outcome.

Test set

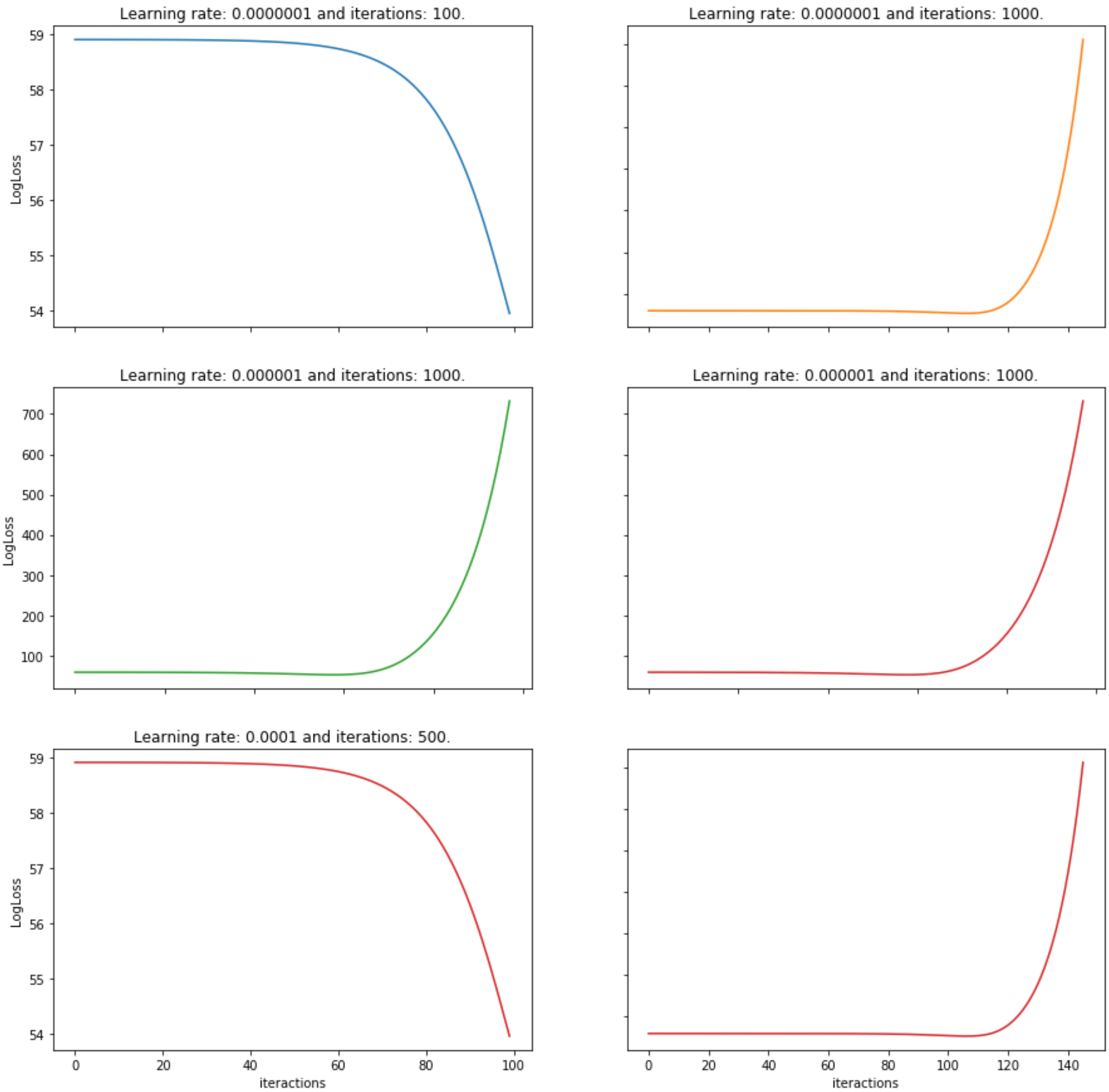
```
# LogLoss function
def LogLoss_function(y, X, beta_hat1):
  n1 = X.shape[0]
  a = (-1/n1)*(np.sum((y@(np.log(sigma(X, beta_hat1))).T) + ((1-y)@(np.log(1-sigma(X, beta_hat1))).T)))
```

```
  #b = np.sum((y@(np.log(sigma(X, beta_hat1))).T) + ((1-y)@(np.log(1-sigma(X, beta_hat1))).T))
  return a

def learn_logreg_GA(X, y, num_iters, u_old, u_plus, u_minus):
  logLoss_total = []
  X = X
  y = y
  n1 = X.shape[0]
  beta1 = Beta_train
  loss = loss_general(X, y, beta1)
  for i in range(num_iters):
    y_hat = (1 / 1 + np.exp(np.dot(X, -beta1)))
    u = bold_driver_step(X, y, u_old, u_plus, u_minus, beta1, y_hat)
    beta_hat1 = beta1 + u*np.dot(X.T, (y - X@beta1))
    lossOld = loss
    loss = loss_general(X, y, beta_hat1)
    logLoss = LogLoss_function(y, X, beta_hat1)
    logLoss_total.append(logLoss)
    u_old = u
  return beta_hat1, logLoss_total



X = tic_tac_test.drop(['Class'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
y = tic_tac_test['Class'].values
y = np.reshape(y, (len(y),1))

betas1, logLoss_total = learn_logreg_GA(X, y, 100, 0.0000001, 1.1, 0.5)
print('Betas', betas1, '\n', 'LogLoss', logLoss_total)
```

```
Betas [[0.47734346]
  [0.08626619]
  [0.09129938]
  [0.08561476]
  [0.09147371]
  [0.07429302]
  [0.09216535]
  [0.0810511 ]
  [0.0916039 ]
  [0.08603815]]
  LogLoss [58.91315056914709, 58.91314501721865, 58.91313891010046, 58.913132192274155, 58.91312480266972, 58.913116674110285, 5
```

```
a, b = learn_logreg_GA(X, y, 100, 0.000001, 1.1, 0.5)
d, e = learn_logreg_GA(X, y, 200, 0.000001, 1.1, 0.5)
h, i = learn_logreg_GA(X, y, 100, 0.0001, 1.1, 0.5)
k, l = learn_logreg_GA(X, y, 300, 0.0001, 1.1, 0.5)
m, n = learn_logreg_GA(X, y, 100, 0.000001, 1.1, 0.5)
o, p = learn_logreg_GA(X, y, 600, 0.000001, 1.1, 0.5)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), b)
axs[0, 0].set_title('Learning rate: 0.0000001 and iterations: 100.')
axs[0, 1].plot(range(200), e, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.0000001 and iterations: 1000.')
axs[1, 0].plot(range(100), i, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.000001 and iterations: 1000.')
axs[1, 1].plot(range(300), l, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.000001 and iterations: 1000.')
axs[2, 0].plot(range(100), n, 'tab:red')
axs[2, 0].set_title('Learning rate: 0.0001 and iterations: 500.')
axs[2, 1].plot(range(600), p, 'tab:red')
axs[2, 0].set_title('Learning rate: 0.0001 and iterations: 500.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='LogLoss')

for ax in axs.flat:
    ax.label_outer()
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by zero encountered in log
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: invalid value encountered in matmul
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:9: RuntimeWarning: overflow encountered in exp
  if __name__ == '__main__':
```



Observations:

> The behavior of the logLogs in regards to the iterations have a big impact: The less iteractions the model has, the better shape the curve will have to reach convergence.

> If the number of iterations increase the curve will change to a right-up-left curve such as the presented in the graph 4 and 6.

> The learning rate directly affects the shape of the curve. A lower learning rate will change the direction of the curve, as one can see in graphs 1 and 3.

> Yet, the model is not reaching convergence in the desired direction.

Algorithm learn Logreg GA with Newton

```
# Sigma function
def sigma(X, beta):
  sigma = 1/(1+np.exp(-np.matmul(X, beta)))
  return sigma

# Function for the Loss.
```

```python
def loss_function(y, X, beta_hat, beta):
  loss_previous = np.sum(y@(np.log(sigma(X, beta))).T + (1-y)@(np.log(1-sigma(X, beta))).T)
  loss_actual = np.sum(y@(np.log(sigma(X, beta_hat))).T + (1-y)@(np.log(1-sigma(X, beta_hat))).T)
  loss_total = abs(loss_previous - loss_actual)
  return loss_total

def learn_logreg_Newton(X, y, u, num_iters):
  X = X
  y = y
  loss_values = []
  n = X.shape[1]
  beta = np.zeros(n)
  beta = np.reshape(beta, (len(beta),1))
  loss = np.sum(y@(np.log(sigma(X, beta))).T + (1-y)@(np.log(1-sigma(X, beta))).T)
  betas, loss_total = minimize_Newton(X, y, u, beta, num_iters)
  return betas, loss_total

def minimize_Newton(X, y, u, beta, num_iters):
  loss_decrease = []
  for i in range(num_iters):
    y_hat = 1 / (1 + np.exp(-(beta.T@X.T)))
    y_hat_vector = np.reshape(y_hat.T, (len(y_hat.T)))
    g = X.T@(y - y_hat.T)
    W = np.diag(y_hat_vector*(1-y_hat_vector))
    XW = X.T@W
    H = np.matmul(XW, X)
    inv = np.linalg.inv(H)
    beta_hat = beta + u*(((np.linalg.inv(H))@g))
    loss_calculation = loss_function(y, X, beta_hat, beta)
    loss_decrease.append(loss_calculation)
    beta = beta_hat
  return beta_hat, loss_decrease
```

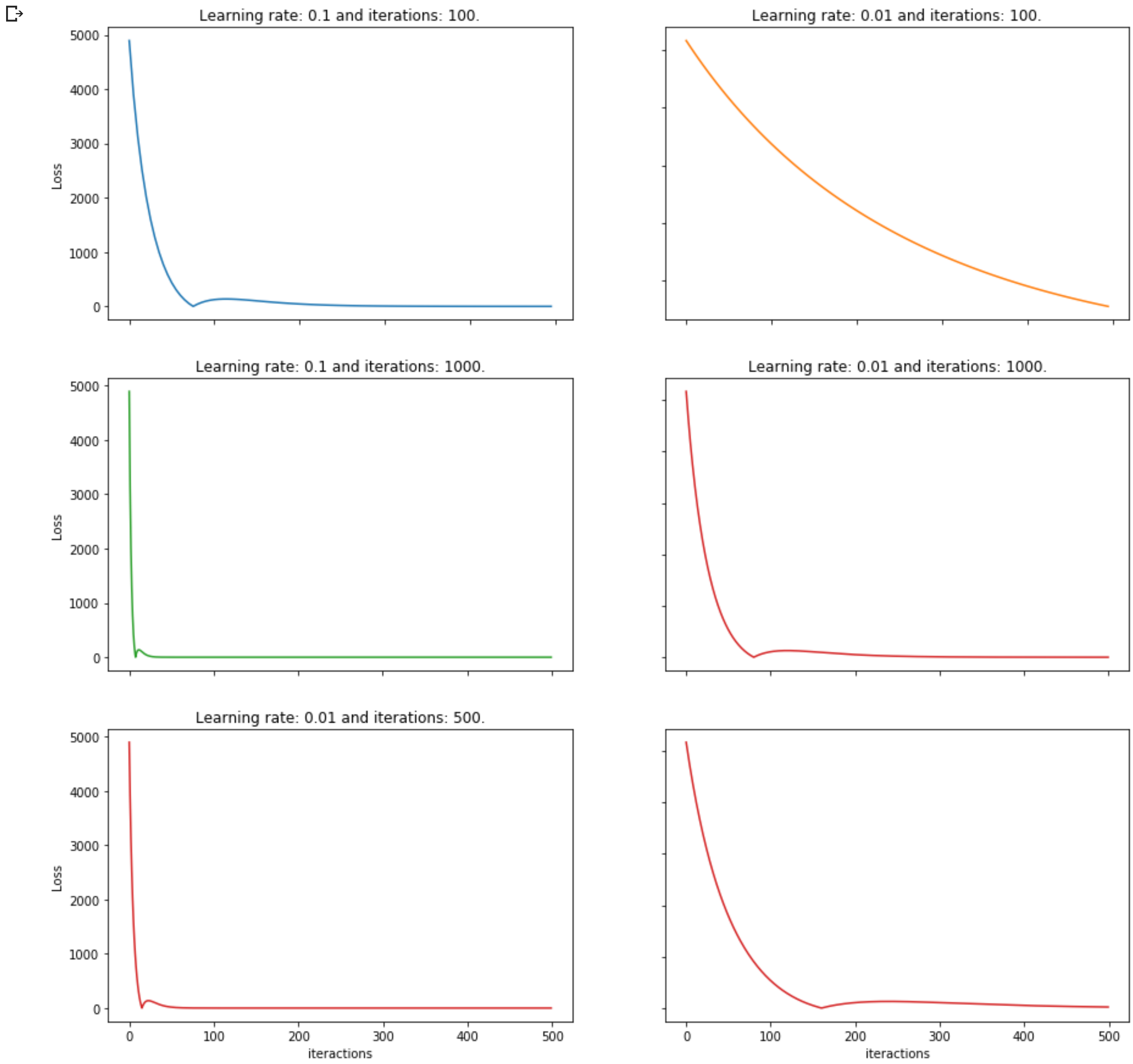
```python
X = tic_tac_train.drop(['Class'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
y = tic_tac_train['Class'].values
y = np.reshape(y, (len(y),1))
```

```python
Betas, loss_decrease = learn_logreg_Newton(X, y, 0.001, 100)
Beta = learn_logreg_Newton(X, y, 0.001, 100)[0] # We are going to take the betas for the testing set.
print('Betas', Betas, '\n', 'Loss', loss_decrease)
```

```
Betas [[ 0.10982896]
 [-0.08578824]
 [ 0.0966201 ]
 [ 0.05043292]
 [ 0.20019964]
 [-0.43506479]
 [ 0.1176824 ]
 [-0.15932727]
 [ 0.19769831]
 [-0.07962518]]
 Loss [51.95352146901132, 51.84072581243527, 51.728211376888794, 51.61597720746067, 51.5040223539545, 51.39234587021929, 51.280
```

```python
a, b = learn_logreg_Newton(X, y, 0.1, 100)
d, e = learn_logreg_Newton(X, y, 0.01, 100)
h, i = learn_logreg_Newton(X, y, 0.1, 1000)
k, l = learn_logreg_Newton(X, y, 0.01, 1000)
m, n = learn_logreg_Newton(X, y, 0.1, 500)
o, p = learn_logreg_Newton(X, y, 0.01, 500)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), b)
axs[0, 0].set_title('Learning rate: 0.1 and iterations: 100.')
axs[0, 1].plot(range(100), e, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.01 and iterations: 100.')
axs[1, 0].plot(range(1000), i, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.1 and iterations: 1000.')
axs[1, 1].plot(range(1000), l, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.01 and iterations: 1000.')
axs[2, 0].plot(range(500), n, 'tab:red')
axs[2, 0].set_title('Learning rate: 0.1 and iterations: 500.')
axs[2, 1].plot(range(500), p, 'tab:red')
axs[2, 0].set_title('Learning rate: 0.01 and iterations: 500.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='Loss')

for ax in axs.flat:
```

```
ax.label_outer()
```



Observations:

> As it is possible to appreciate in the graphs presented above a higher learning rate reaches convergence faster.

> The less iterations the model counts has means more iteractions to reach convergence. In other words, the size of the steps are going to be small, therefore, it takes more time/iterations to reach convergence.

> Small bias is experienced in the curves where there is a increase in the loss.

Test set

```
# Sigma function
def sigma(X, beta1):
  sigma = 1/(1+np.exp(-np.matmul(X, beta1)))
  return sigma

# LogLoss function
def LogLoss_function(y, X, beta_hat1, beta1):
  n2 = X.shape[0]
  #a = np.sum(-(y@(np.log(sigma(X, beta_hat1))).T + (1-y)@(np.log(1-sigma(X, beta_hat1))).T))
  a = (1/n2)*(np.sum(-(y@(np.log(sigma(X, beta_hat1))).T + (1-y)@(np.log(1-sigma(X, beta_hat1))).T)))
  return a

def learn_logreg_Newton(X, y, u, num_iters):
  X = X
  y = y
  loss_values = []
```

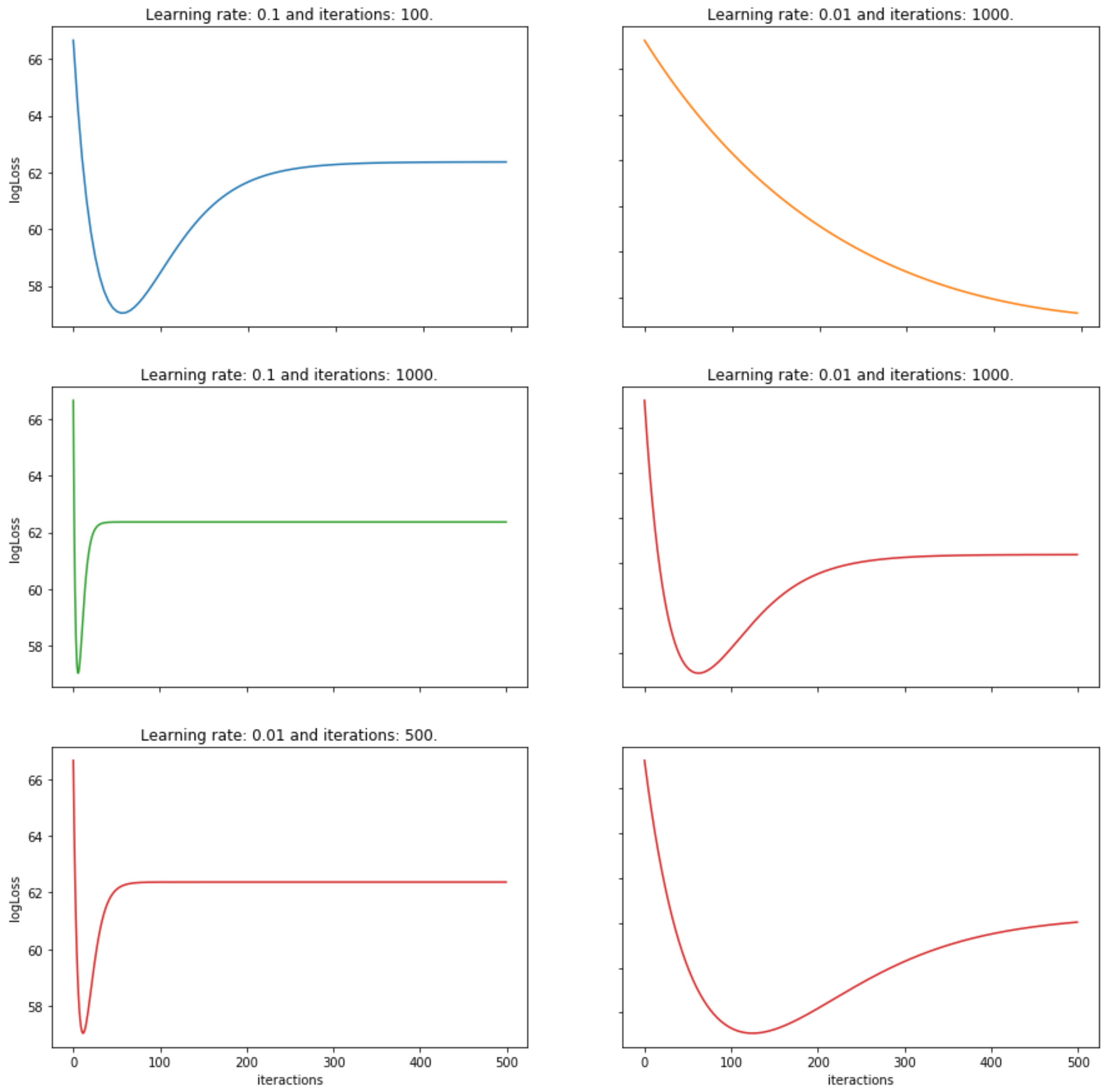
```
  n = X.shape[1]
  beta1 = Beta
  #beta1 = np.reshape(beta, (len(beta),1))
  loss = np.sum(y@(np.log(sigma(X, beta1))).T + (1-y)@(np.log(1-sigma(X, beta1))).T)
  betas, loss_total = minimize_Newton(X, y, u, beta1, num_iters)
  return betas, loss_total

def minimize_Newton(X, y, u, beta1, num_iters):
  logLoss_total = []
  for i in range(num_iters):
    y_hat1 = 1 / (1 + np.exp(-(beta1.T@X.T)))
    y_hat_vector1 = np.reshape(y_hat1.T, (len(y_hat1.T)))
    g = X.T@(y - y_hat1.T)
    W = np.diag(y_hat_vector1*(1-y_hat_vector1))
    XW = X.T@W
    H = np.matmul(XW, X)
    inv = np.linalg.inv(H)
    beta_hat1 = beta1 + u*(((np.linalg.inv(H))@g))
    logLoss = LogLoss_function(y, X, beta_hat1, beta1)
    logLoss_total.append(logLoss)
    beta1 = beta_hat1
  return beta_hat1, logLoss_total
```

```
X = tic_tac_test.drop(['Class'], axis=1).values
column_one = np.ones((X.shape[0],1))
X = np.concatenate((column_one, X), axis = 1)
y = tic_tac_test['Class'].values
y = np.reshape(y, (len(y),1))
```

```
Betas, logLoss = learn_logreg_Newton(X, y, 0.001, 100)
print('Betas', Betas, '\n', 'LogLoss', logLoss)
```

```
Betas [[ 0.41980782]
 [-0.45579958]
 [ 0.06901973]
 [ 0.04979284]
 [ 0.46201639]
 [-0.97074362]
 [ 0.06324369]
 [-0.26375076]
 [ 0.066664  ]
 [-0.32358181]]
 LogLoss [69.5210569589389, 69.49021120430541, 69.45943323002662, 69.42872287012368, 69.39807995918734, 69.36750433237526, 69.3
```

```
a, b = learn_logreg_Newton(X, y, 0.1, 100)
d, e = learn_logreg_Newton(X, y, 0.01, 100)
h, i = learn_logreg_Newton(X, y, 0.1, 1000)
k, l = learn_logreg_Newton(X, y, 0.01, 1000)
m, n = learn_logreg_Newton(X, y, 0.1, 500)
o, p = learn_logreg_Newton(X, y, 0.01, 500)
fig, axs = plt.subplots(3, 2,figsize=(15,15))
axs[0, 0].plot(range(100), b)
axs[0, 0].set_title('Learning rate: 0.1 and iterations: 100.')
axs[0, 1].plot(range(100), e, 'tab:orange')
axs[0, 1].set_title('Learning rate: 0.01 and iterations: 1000.')
axs[1, 0].plot(range(1000), i, 'tab:green')
axs[1, 0].set_title('Learning rate: 0.1 and iterations: 1000.')
axs[1, 1].plot(range(1000), l, 'tab:red')
axs[1, 1].set_title('Learning rate: 0.01 and iterations: 1000.')
axs[2, 0].plot(range(500), n, 'tab:red')
axs[2, 0].set_title('Learning rate: 0.1 and iterations: 500.')
axs[2, 1].plot(range(500), p, 'tab:red')
axs[2, 0].set_title('Learning rate: 0.01 and iterations: 500.')

for ax in axs.flat:
    ax.set(xlabel='iteractions', ylabel='logLoss')

for ax in axs.flat:
    ax.label_outer()
```



Observations:

> The model is not working correctly while making the generalization in the test set.

> The bias in the dataset influences the output to the level for which variances in the iterations, and learning rate will not have an impact.

Final Thoughts

As a summary of the previpus observations:

> Gradient Ascent requires less iterations to reach convergence. Since it is a model with a line search leraning rate it is modeled to find the optimum and increase/decrease steps accordingly.

> Newton model on the other hand requires a high learning rate to get to the minimum quicker. Moreover, the less number of iterations the model has, the more steps it will require since the learning rate does not change.

> Finally, in the scenario presented for this work, Gradient Ascent performs better and generalize accurately in comparison to Newton's model.