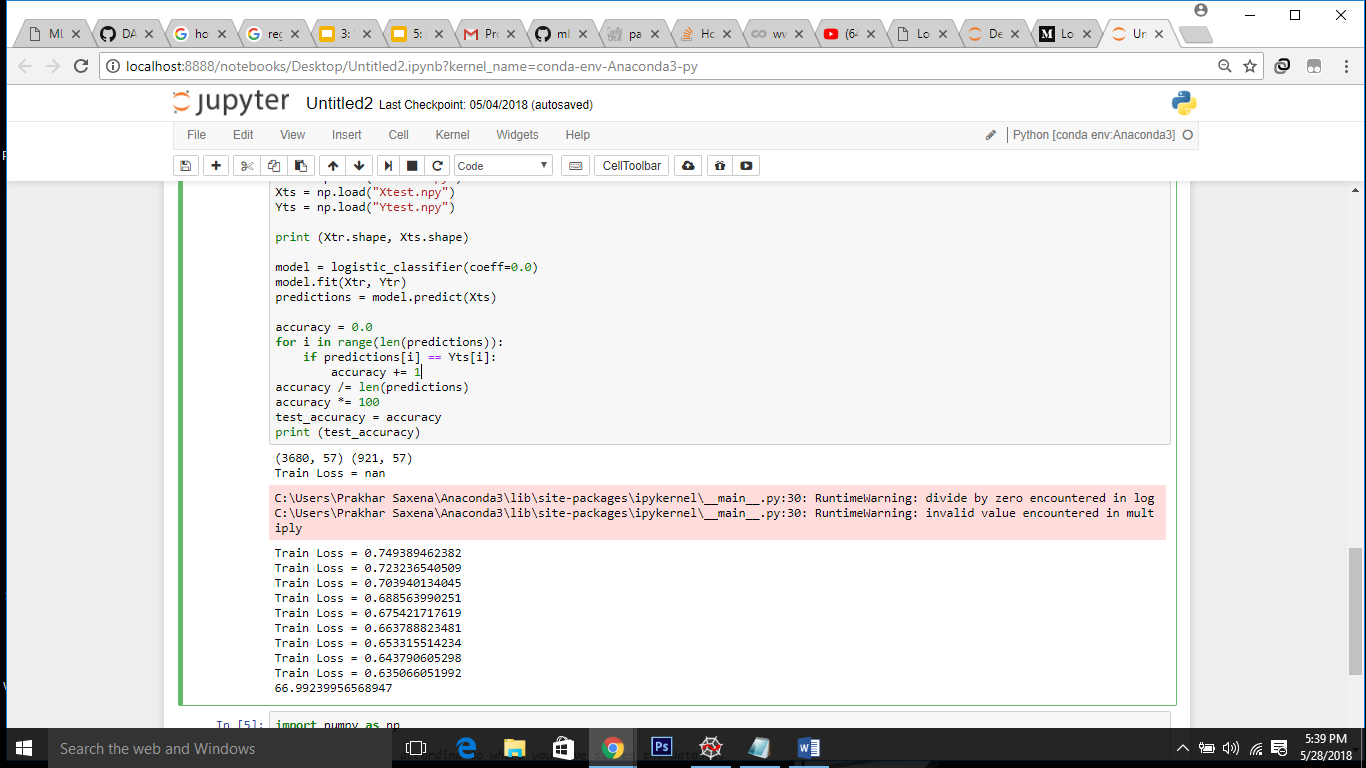
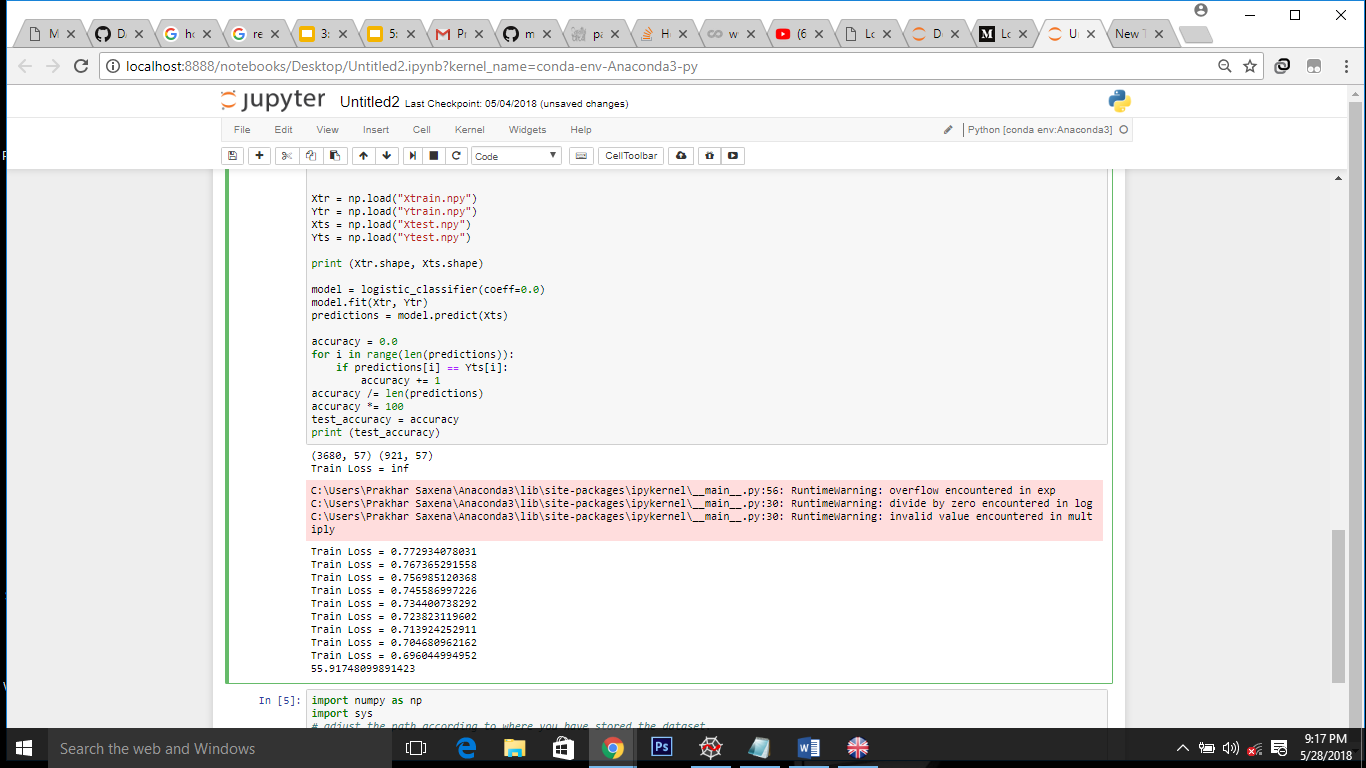
ML assignment (PRAKHAR SAXENA [prakharsaxena303@gmail.com](mailto:prakharsaxena303@gmail.com)

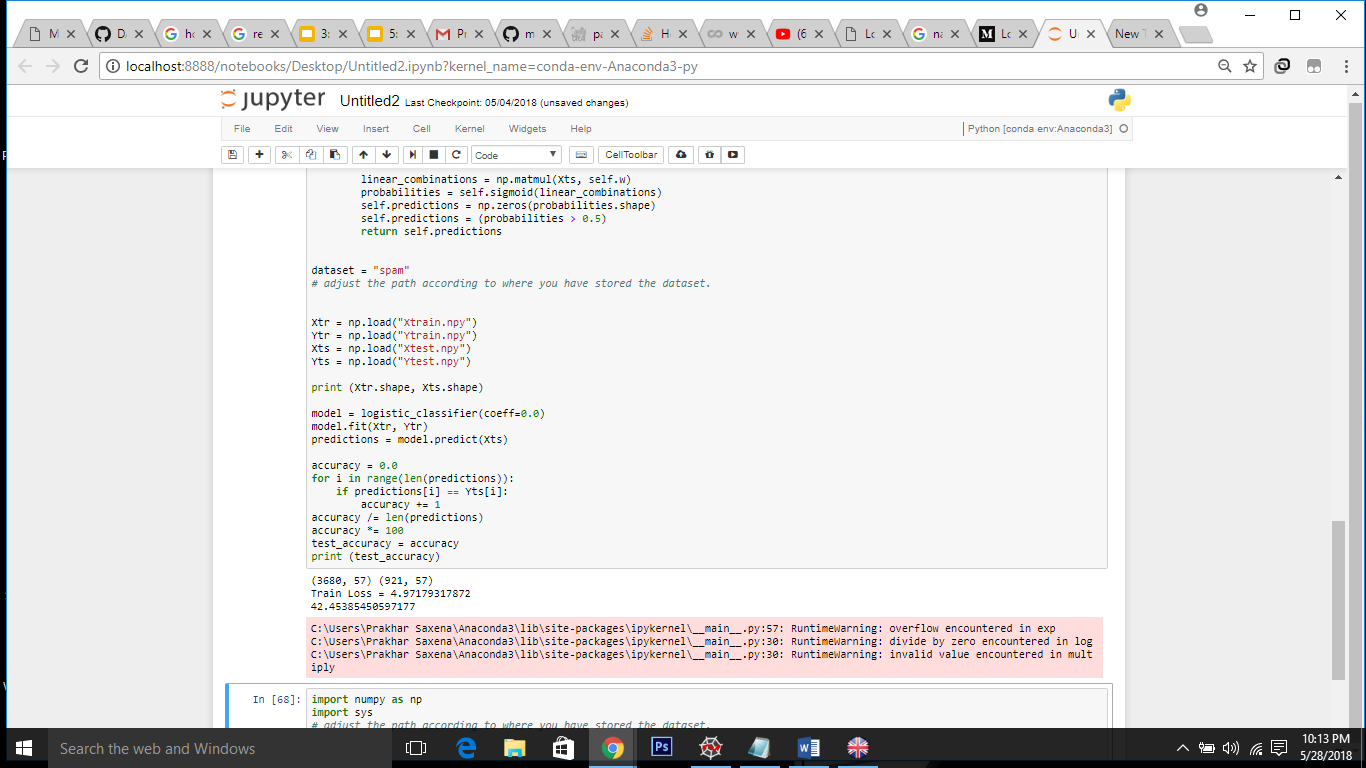
7292024671)

OUTPUT(observations)





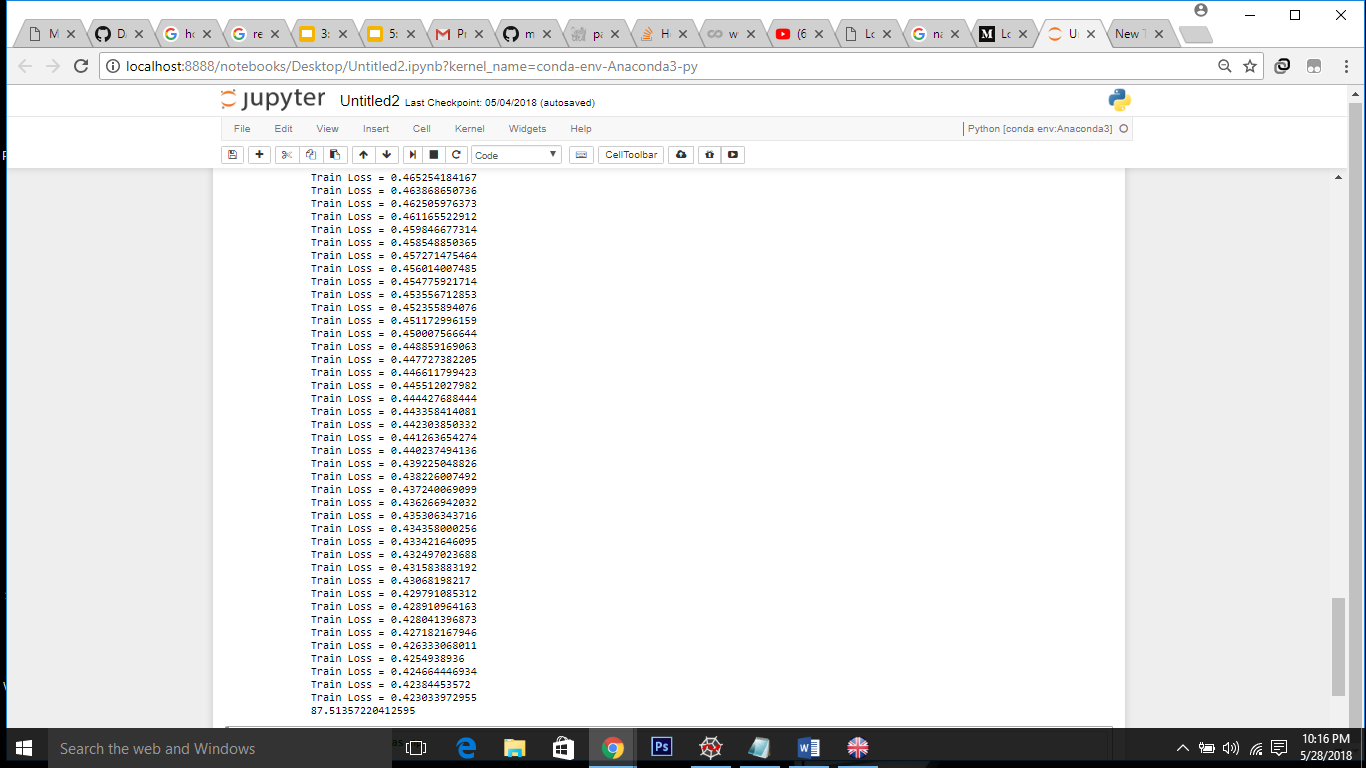
**Changing no of iterations to 100**



No of attempts to minimize loss reduced

Decreased accuracy

**Changing no of iterations to 100000**



A large number of attempts to reduce training loss

Time expensive **But** accuracy increased to 88 % !!!!

##### code view #######

* def \_\_init\_\_(self, coeff):

self.coeff = coeff

self.w = 0.0

Simple initialization with parameter “w”

* def compute\_probabilities(self, Xtr):

#print("sigmoid")

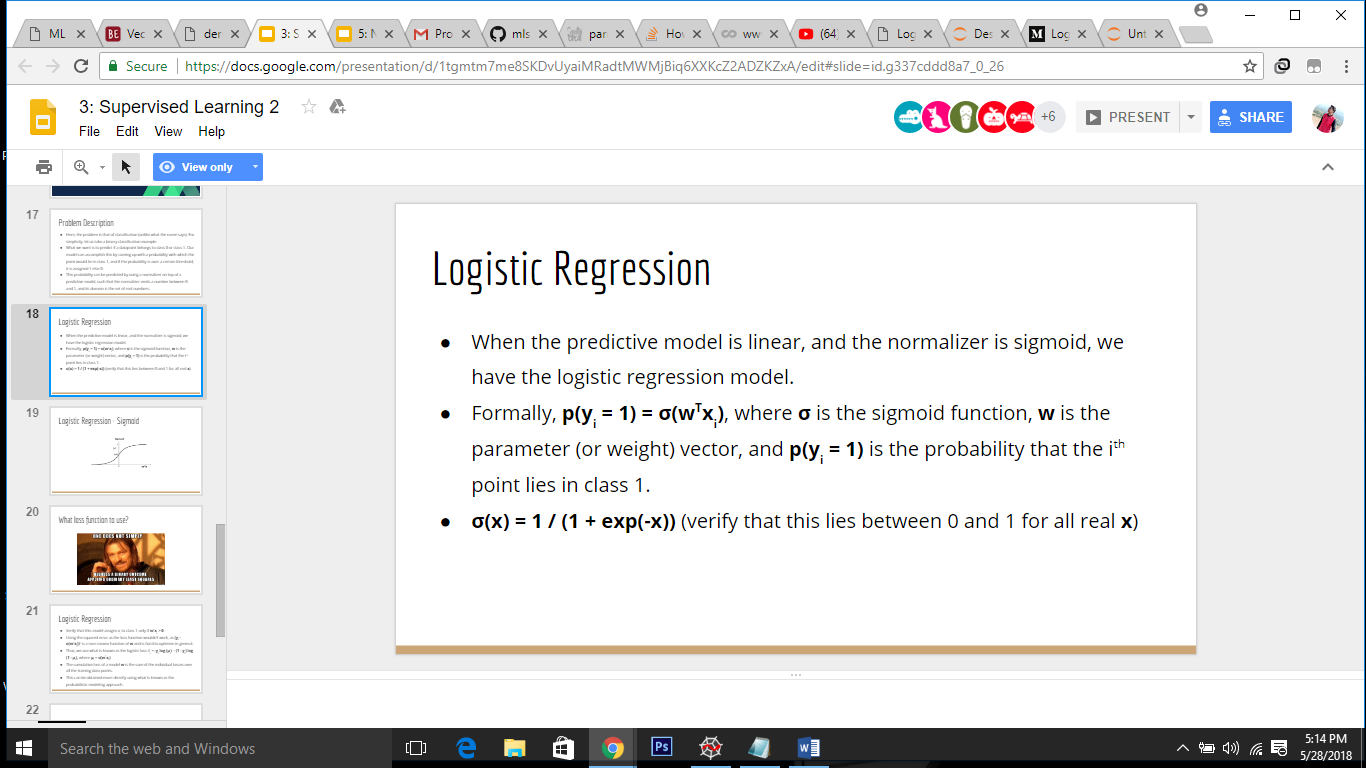
z = np.dot(Xtr, self.w)

h = self.sigmoid(z)

#print (h)

return h

Generate probabilities, logistic regression uses a function(sigmoid) that gives outputs between 0 and 1 for all values of X.

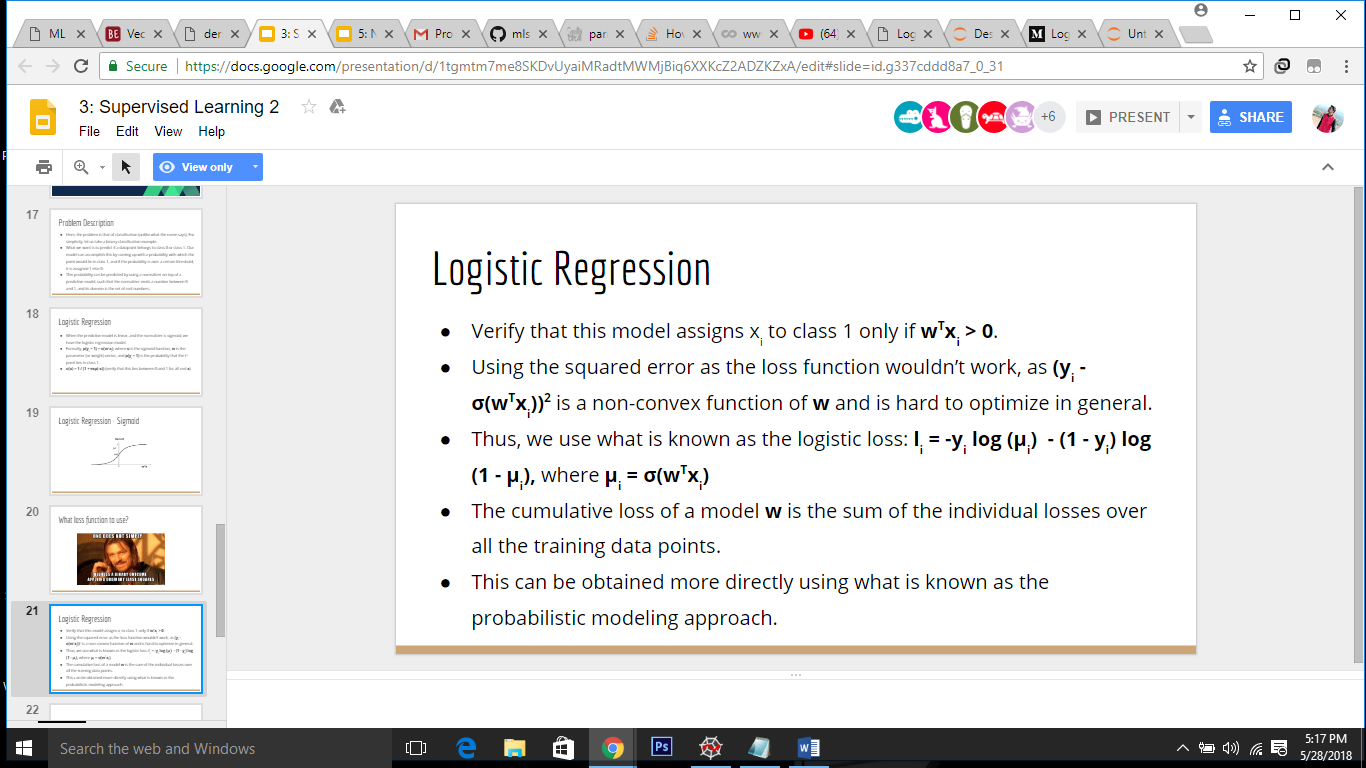


* def compute\_loss(self, probabilities, Ytr):

mm = (-Ytr \* np.log(probabilities) - (1 - Ytr) \* np.log(1 - probabilities)).mean()

#print ("loss =" + mm)

return mm



* def compute\_gradients(self, probabilities, Ytr, Xtr):

grad = np.dot(Xtr.T, (probabilities - Ytr))

#print ("grad = " , grad )

return grad

Goal is to minimize the loss function and the way we have to achive it is by increasing/decreasing the weights(updated later in a function).  the derivative of the loss function with respect to each weight. It tells us how loss would change if we modified the parameters.

Regularization component(**not used** bcuz of weird output) to remove outfit.

* def update\_weights(self, learning\_rate, grads):

# function to update weights. This needs to be filled in.

self.w -= learning\_rate \* grads

pass

* def sigmoid(self, inputs):

#(not working){ **IP=-inputs**

**exp=1/(1 + np.exp(IP))**

**exp[exp>0.99]=0.99**

**exp[exp>0.01]=0.01**

**return exp**

}

return 1 / (1 + np.exp(-inputs))

to compute sigmoid function .

* def fit(self, Xtr, Ytr):

learning\_rate = 0.00005

self.num\_iters = 10000

# random initialization for w.

self.w = np.random.normal(0.0, 0.1, Xtr.shape[1])

for iter in range(self.num\_iters):

probabilities = self.compute\_probabilities(Xtr)

train\_loss = self.compute\_loss(probabilities, Ytr)

grads = self.compute\_gradients(probabilities, Ytr, Xtr)

grads = grads / Xtr.shape[0]

self.update\_weights(learning\_rate, grads)

if iter % 1000 == 0:

print ("Train Loss = " + str(train\_loss))

* def predict(self, Xts):

linear\_combinations = np.matmul(Xts, self.w)

probabilities = self.sigmoid(linear\_combinations)

self.predictions = np.zeros(probabilities.shape)

self.predictions = (probabilities > 0.5)

return self.predictions