* MLE for ; argmax = argmax
  + ={}
* dy
* =

MLE = { }



argmax = argmax by bays rule

= argmax =

=

=

=

)

=

* 1. log PΘ(D) = log Πn i=1p(xi , yi) = log Πn i=1p(xi |yi)p(yi).

log PΘ(D)

= =

=

=

=

=

log PΘ(D)

=

=

= =

log PΘ(D)

=



# Add code to plot the trend of the total number of people being tested as days progressed.

# X axis -> dates('Dates')

# Y axis -> number of people tested.('People\_tested')

### STUDENT: Start of Code ###

x = dates.to\_numpy()

y= data['People\_tested'].to\_numpy()

plt.plot(x,y)

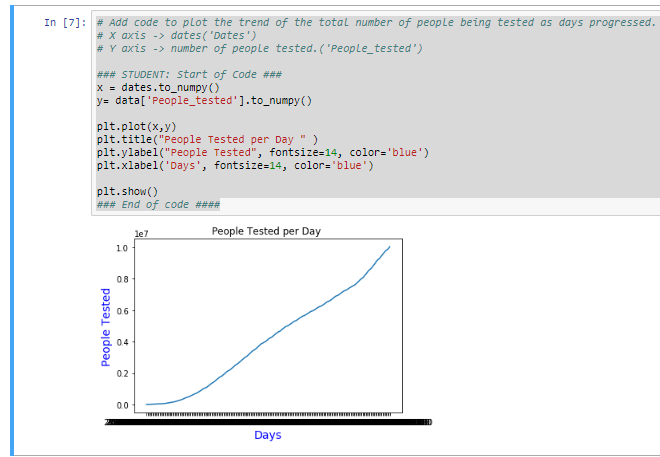
plt.title("People Tested per Day " )

plt.ylabel("People Tested", fontsize=14, color='blue')

plt.xlabel('Days', fontsize=14, color='blue')

plt.show()

### End of code ####



# Add code to plot the trend of total deaths as days progressed.

# X axis -> dates ('Dates')

# Y axis -> number of deaths ('Deaths')

### STUDENT: Start of Code ###

x = dates.to\_numpy()

y= data['Deaths'].to\_numpy()

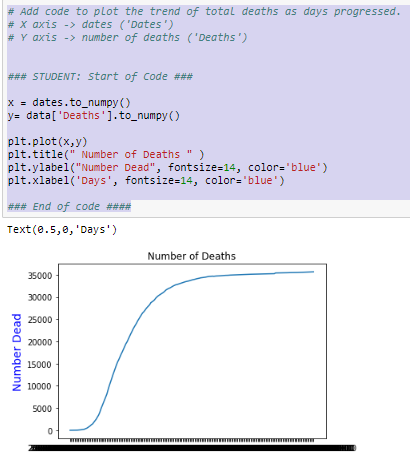
plt.plot(x,y)

plt.title(" Number of Deaths " )

plt.ylabel("Number Dead", fontsize=14, color='blue')

plt.xlabel('Days', fontsize=14, color='blue')

### End of code ####



### STUDENT: Start of Code ###

x = dates.to\_numpy()

y= data['New\_positive\_cases'].to\_numpy()

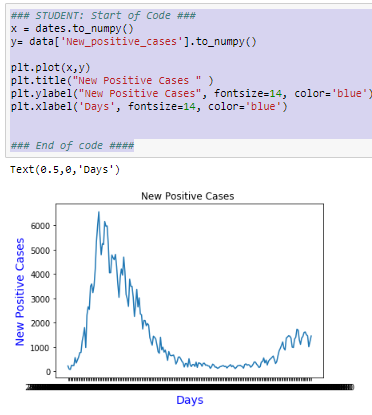
plt.plot(x,y)

plt.title("New Positive Cases " )

plt.ylabel("New Positive Cases", fontsize=14, color='blue')

plt.xlabel('Days', fontsize=14, color='blue')

### End of code ####





def predict\_output(feature\_matrix, weights):

# Inputs:

# feature\_matrix: a numpy matrix containing the features as columns (including the intercept),

# and each row corresponds to a data point

# weights: a numpy array for the corresponding regression weights (including the intercept)

# Output:

# a numpy array that contains the predicted outputs (according to the provided weights)

# for all the data points in the feature\_matrix

# STUDENT: Start of code ####

predictions=[]

sum = 0

rowSum = 0;

max\_value = 0;

#get dot product for each row

for i in range(len(feature\_matrix)):

rowSum = 0

#dot

for j in range(len(feature\_matrix[i])):

rowSum =rowSum + (feature\_matrix[i][j]\*weights[j])

#for normalization

if(rowSum>max\_value):

max\_value = rowSum

#list of predictions from dot product

predictions.append(rowSum)

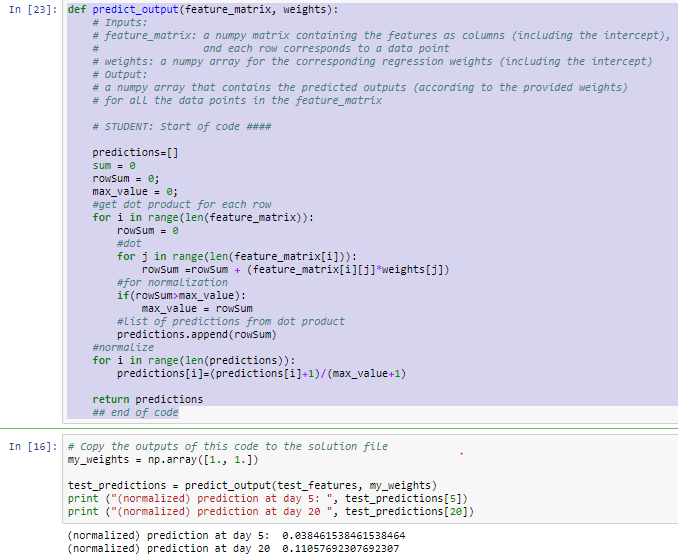
#normalize

for i in range(len(predictions)):

predictions[i]=(predictions[i]+1)/(max\_value+1)

return predictions

## end of code





def weight\_derivative(weights, feature\_matrix, labels):

# Input:

# weights: weight vector w, a numpy vector of dimension d

# feature\_matrix: numpy array of size n by d, where n is the number of data points, and d is the feature dimension

# labels: true labels y, a numpy vector of dimension d

# Output:

# Derivative of the regression cost function with respect to the weight w, a numpy array of dimension d

## STUDENT: Start of code ###

sum = 0

#get array of normalized dot produts

dot\_product=predict\_output(feature\_matrix,weights)

#sum the diffrence between dot and true value

for i in range(len(feature\_matrix)):

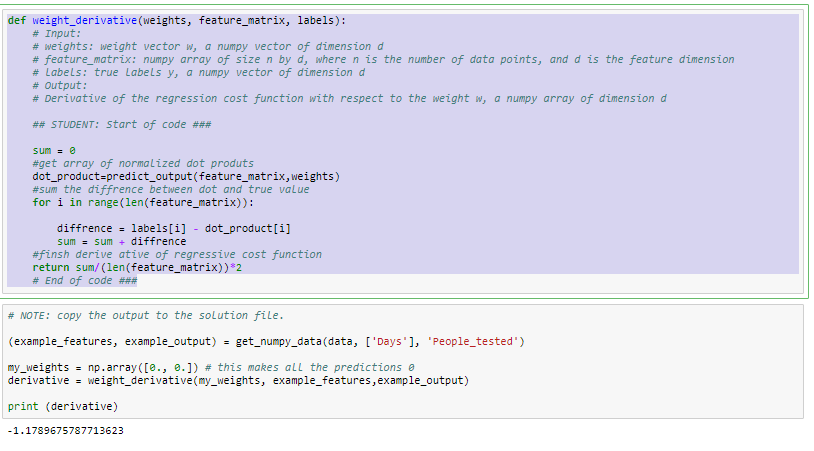
diffrence = labels[i] - dot\_product[i]

sum = sum + diffrence

#finsh derive ative of regressive cost function

return sum/(len(feature\_matrix))\*2

# End of code ###





def regression\_gradient\_descent(feature\_matrix, labels, initial\_weights, step\_size, tolerance):

# Gradient descent algorithm for linear regression problem

# Input:

# feature\_matrix: numpy array of size n by d, where n is the number of data points, and d is the feature dimension

# labels: true labels y, a numpy vector of dimension d

# initial\_weights: initial weight vector to start with, a numpy vector of dimension d

# step\_size: step size of update

# tolerance: tolerace epsilon for stopping condition

# Output:

# Weights obtained after convergence

converged = False

weights = np.array(initial\_weights) # current iterate

i = 0

while not converged:

i += 1

# STUDENT: Start of code: your impelementation of what the gradient descent algorithm does in every iteration

# Refer back to the update rule listed above: update the weight

# Compute the gradient magnitude:

weight\_deriv = weight\_derivative(weights[i-1], feature\_matrix,labels)

temp\_gradient = [0]\*len(weights[i-1])

for a in range(len(weights[i-1])):

weight\_deriv= weight\_deriv\*step\_size

temp\_gradient[a] = weights[i-1][a] - weight\_deriv

gradient\_magnitude = temp\_gradient

#size of gradient magnitude

gradient\_size = 0

for a in gradient\_magnitude:

gradient\_size += pow(a,2)

gradient\_size = pow(gradient\_size,1/2)

#reasign weights

for j in range(len(weights[i])):

weights[i][j]=gradient\_magnitude[j]

#weights[i][j]=temp\_gradient[j]

# Check the stopping condition to decide whether you want to stop the iterations

if (gradient\_size>=tolerance): # STUDENT: check the stopping condition here

converged = True

if (i>=len(weights)-1): # STUDENT: check the stopping condition here

converged = True

# End of code

print ("Iteration: ",i,"gradient\_magnitude: ", gradient\_magnitude) # for us to check about convergence

return(gradient\_magnitude)





simple\_features = ['Days']

my\_output = 'People\_tested'

# Use get\_numpy\_data method to calculate the feature matrix and output.

(simple\_feature\_matrix, output) = get\_numpy\_data(train\_data, simple\_features, my\_output)

#Initialize the weights, step size and tolerance

# Start of code

#STUDENT: Specify the initial\_weights, step\_size, and tolerance

temp\_weights = np.array([0.]\*2\*len(simple\_feature\_matrix))

initial\_weights = temp\_weights.reshape(len(simple\_feature\_matrix),2)

step\_size = 0.05

tolerance = 1

# end of code

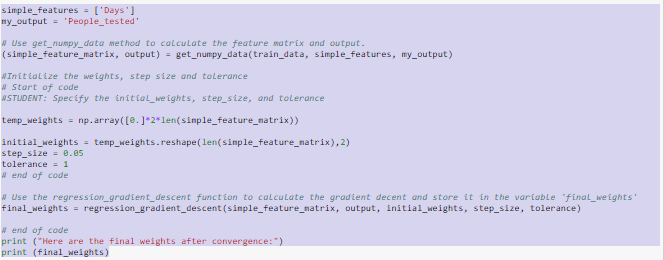
# Use the regression\_gradient\_descent function to calculate the gradient decent and store it in the variable 'final\_weights'

final\_weights = regression\_gradient\_descent(simple\_feature\_matrix, output, initial\_weights, step\_size, tolerance)

# end of code

print ("Here are the final weights after convergence:")

print (final\_weights)







# Calculate the test error

# STUDENT: Start of code

test\_error = 0;

size = len(test\_predictions)

#print (test\_output)

#print(test\_predictions)

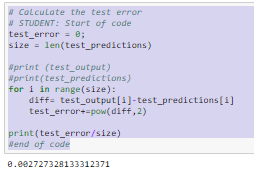
for i in range(size):

diff= test\_output[i]-test\_predictions[i]

test\_error+=pow(diff,2)

print(test\_error/size)

#end of code





model\_features = ['Intensive\_care','New\_positive\_cases','Days']

my\_output = 'People\_tested'

#call the get\_nupy\_data method to calculate the feature matrix and output. Store them in the variables "multi\_feature\_matrix" & "output"

(multi\_feature\_matrix, output) = get\_numpy\_data(data, model\_features, my\_output)

# Initialize the weights, step size and tolerance

# STUDENT: Start of code

# STUDENT: Specify the initial\_weights, step\_size, and tolerance

#print(len(multi\_feature\_matrix))

temp\_weights = np.array([0.]\*4\*len(multi\_feature\_matrix))

initial\_weights = temp\_weights.reshape(len(multi\_feature\_matrix),4)

#print(initial\_weights)

step\_size = 0.05

tolerance = 1

#print(multi\_feature\_matrix)

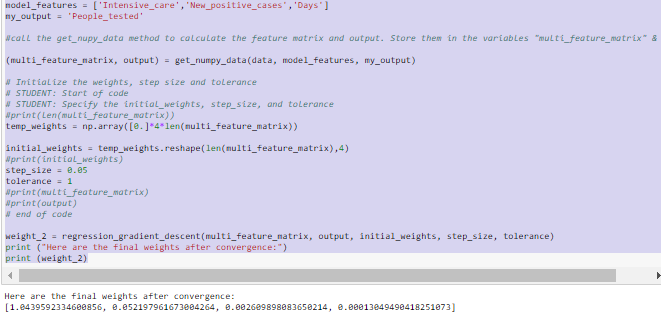
#print(output)

# end of code

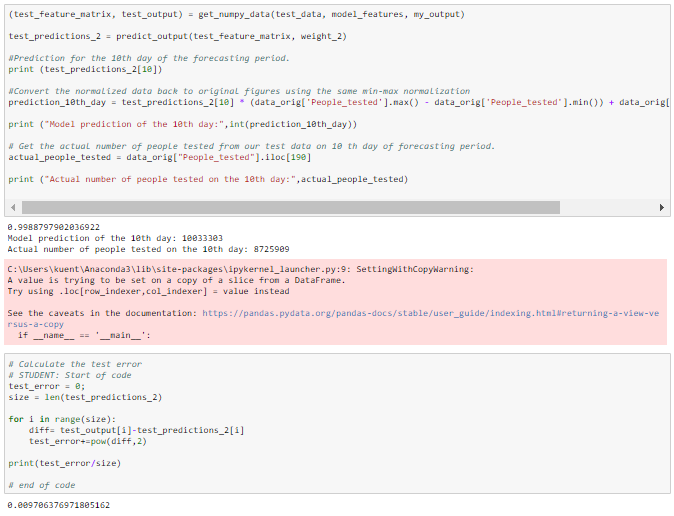
weight\_2 = regression\_gradient\_descent(multi\_feature\_matrix, output, initial\_weights, step\_size, tolerance)

print ("Here are the final weights after convergence:")

print (weight\_2)









# Explore an aspect of the model that interests you

### STUDENT: Start of code

model\_features = ['Intensive\_care','New\_positive\_cases','Days']

my\_output = 'People\_tested'

#call the get\_nupy\_data method to calculate the feature matrix and output. Store them in the variables "multi\_feature\_matrix" & "output"

(multi\_feature\_matrix, output) = get\_numpy\_data(data, model\_features, my\_output)

# Initialize the weights, step size and tolerance

# STUDENT: Start of code

# STUDENT: Specify the initial\_weights, step\_size, and tolerance

temp\_weights = np.array([0.]\*4\*len(multi\_feature\_matrix))

step\_size = 0.05

tolerance = 1

while tolerance < 20:

print ('tolerance: ', end = ' ')

print(tolerance)

initial\_weights = temp\_weights.reshape(len(multi\_feature\_matrix),4)

weight\_2 = regression\_gradient\_descent(multi\_feature\_matrix, output, initial\_weights, step\_size, tolerance)

(test\_feature\_matrix, test\_output) = get\_numpy\_data(test\_data, model\_features, my\_output)

test\_predictions\_2 = predict\_output(test\_feature\_matrix, weight\_2)

test\_error = 0;

size = len(test\_predictions\_2)

print('test\_error: ', end = ' ')

for i in range(size):

diff= test\_output[i]-test\_predictions\_2[i]

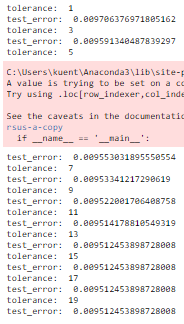
test\_error+=pow(diff,2)

print(test\_error/size)

tolerance +=2

### End of code





NOTE: to get this output I removed the only print statement from regression\_gradient\_descent. This function tests the tolerance error. As we can see, the lower the tolerance, the more incorrect the guess is. The error seems to converge to its lowest point when it has a tolerance of 13