1.

a.

- MLE for û ; argmax θ $P_\theta(D)$ = argmax θ log ($P_\theta(D)$)
  - $\theta = \{\mu\}$
  - D={$y_1, \dots, y_n$}
  - $P_\theta(D) = \prod_{i=0}^{n} \theta^{\sum_{i=1}^{n_1} y_i} (1 - \theta^{n-\sum_{i=1}^{n_1} y_i}) \prod_{i=0}^{n} \theta^{\sum_{i=1}^{n_1} y_i} (1 - \theta^{n-\sum_{i=1}^{n_1} y_i})$

b.

- $P_\theta(D) = \log\left(\theta^{\sum_{i=1}^{n_1} y_i} \left(1 - \theta^{n-\sum_{i=1}^{n_1} y_i}\right)\right)$
- $= \log(\theta^{\sum_{i=1}^{n_1} y_i} \left(1 - \theta^{n-\sum_{i=1}^{n_1} y_i}\right))$
- $= \log(\theta) * \sum_{i=1}^{n_1} y_i \ * \log(1-\theta) * (n - \sum_{i=1}^{n_1} y_i)$

c.

- $\log(\theta) * \sum_{i=1}^{n_1} y_i \ * \log(1-\theta) * (n - \sum_{i=1}^{n_1} y_i)$ dy
- = $\frac{\sum_{i=1}^{n_1} y_i}{\theta} - \frac{n-\sum_{i=1}^{n_1} y_i}{1-\theta} = \frac{(1-\theta)\sum_{i=1}^{n_1} y_i - \theta n + \theta \sum_{i=1}^{n_1} y_i}{\theta(1-\theta)} = \frac{\sum_{i=1}^{n_1} y_i - \theta n}{\theta(1-\theta)}$
- $\theta = \frac{\sum_{i=1}^{n_1} y_i}{n}$

MLE $\theta = \{ \mu \}$ = { $\mu = \frac{\sum_{i=1}^{n_1} y_i}{n}$ }

2.

a.

argmax Y p$(y|x)$ = argmax Y $\frac{\mathrm{p}(x|y)p(y)}{p(x)}$ by bays rule

= argmax Y $\frac{\mathrm{p}(y|x)p(y)}{p(x)}$ = $\frac{\mathrm{p}(y=1|x)p(y=1)}{\mathrm{p}(y=1|x)p(y=1)+\mathrm{p}(y=0|x)p(y=0)}$

$$\mathrm{P}(y=1|x)=\frac{\exp\left(-\frac{1}{2}(\mathrm{x}-\mu_1)^{\mathrm{T}}\,\Gamma^{-1}(\mathrm{x}-\mu_1)\right)*\varphi}{\exp\left(-\frac{1}{2}(\mathrm{x}-\mu_1)^{\mathrm{T}}\,\Gamma^{-1}(\mathrm{x}-\mu_1)\right)*\varphi+\exp\left(-\frac{1}{2}(\mathrm{x}-\mu_0)^{\mathrm{T}}\,\Gamma^{-1}(\mathrm{x}-\mu_0)\right)*(1-\varphi)}$$

$$=\frac{1}{1+\frac{1-\varphi}{\varphi}\exp\left(\frac{1}{2}(\mathrm{x}-\mu_1)^{\mathrm{T}}\,\Gamma^{-1}(\mathrm{x}-\mu_1)-\frac{1}{2}(\mathrm{x}-\mu_0)^{\mathrm{T}}\,\Gamma^{-1}(\mathrm{x}-\mu_0)\right)}$$

$$=\frac{1}{1+\exp\left(\frac{1}{2}(\mathrm{x}-\mu_1)^{\mathrm{T}}\,\Gamma^{-1}(\mathrm{x}-\mu_1)-\frac{1}{2}(\mathrm{x}-\mu_0)^{\mathrm{T}}\,\Gamma^{-1}(\mathrm{x}-\mu_0)+\log\left(\frac{1-\varphi}{\varphi}\right)\right)}$$

$$=\frac{1}{1+\exp\left(\frac{1}{2}(x^T\Gamma^{-1}x-\mu_1\Gamma^{-1}x-x^T\Gamma^{-1}\mu_1-\mu_1\Gamma^{-1}\mu_1-x^T\Gamma^{-1}x+\mu_0\Gamma^{-1}x-x^T\Gamma^{-1}\mu_0+\mu_0\Gamma^{-1}\mu_0\,)+\log\left(\frac{1-\varphi}{\varphi}\right)\right)}=$$

$$\frac{1}{1+\exp\left(\frac{1}{2}(\mu_0\Gamma^{-1}x-(x^T\Gamma^{-1}\mu_0\,)^T+\mu_0\Gamma^{-1}\mu_0+\mu_1\Gamma^{-1}x+(x^T\Gamma^{-1}\mu_1\,)^T-\mu_1\Gamma^{-1}\mu_1+\log\left(\frac{1-\varphi}{\varphi}\right)\right)}=$$

$$\frac{1}{1+\exp\left(\frac{1}{2}(-2\mu_0^T\Gamma^{-1}x+\mu_0\Gamma^{-1}\mu_0+2\,\mu_1\Gamma^{-1}x+-\mu_1\Gamma^{-1}\mu_1)+\log\left(\frac{1-\varphi}{\varphi}\right)\right)}=$$

$$\frac{1}{1+\exp\left(\frac{1}{2}x(-2\mu_0^T\Gamma^{-1}+2\,\mu_1\Gamma^{-1})+\frac{1}{2}(-\mu_1\Gamma^{-1}\mu_1+\mu_0\Gamma^{-1}\mu_0)+\log\left(\frac{1-\varphi}{\varphi}\right)\right)}=\frac{1}{1+\exp\left(\theta^T x+\theta_0\right)}$$

$\theta^T=(-2\mu_0^T\Gamma^{-1}+2\,\mu_1\Gamma^{-1})$

θ_0= $\frac{1}{2}(-\mu_1\Gamma^{-1}\mu_1+\mu_0\Gamma^{-1}\mu_0)\;+\log\left(\frac{1-\varphi}{\varphi}\right)$

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

$$\log(\,\mathrm{p}(y_i|x_i)p(y_i))\;=\log\prod_{i=0}^{n}\frac{1}{(2\pi)^{\frac{\mathrm{d}}{2}}|\Gamma|^{\frac{1}{2}}}\exp\left(-\frac{1}{2}(x_i-\mu_i)^{\mathrm{T}}\,\Gamma^{-1}(x_i-\mu_i)\right)*\;\varphi^{y_i}\,(1-\varphi)^{1-y_i}$$

$$=\;\log\prod_{i=0}^{n}\frac{1}{(2\pi)^{\frac{\mathrm{d}}{2}}|\Gamma|^{\frac{1}{2}}}\log\prod_{i=0}^{n}\exp\left(-\frac{1}{2}(x_i-\mu_i)^{\mathrm{T}}\,\Gamma^{-1}(x_i-\mu_i)\right)*\;\log\prod_{i=0}^{n}\varphi^{y_i}\,(1-\varphi)^{1-y_i}$$

log PΘ(D) dφ

= $\log\prod_{i=0}^{n}\varphi^{y_i}\,(1-\varphi)^{1-y_i}$ = $\sum_{i=1}^{n}\log(\varphi^{y_i}\,(1-\varphi)^{1-y_i})\mathrm{d}\varphi=$

$=\;\sum_{i=1}^{n}\,y_i(\log\varphi)+(1-y_i)(\log(1-\varphi)))\mathrm{d}\varphi=\;\sum_{i=1}^{n}\frac{y_i}{\varphi}+\frac{1-y_i}{(\varphi-1)}$ = $\sum_{i=1}^{n}\frac{(1-y_i\,)\varphi+y*(\varphi-1)}{(\varphi-1)\varphi}$

$=(1-\varphi)\varphi\,\sum_{i=1}^{n}(1-y_i\,)\varphi+y*(\varphi-1)\;=\sum_{i=1}^{n}(\varphi-y_i\,\varphi)+\varphi y_i\;-y_i$

$=\sum_{i=1}^{n}\varphi\;-y_i\;=\;\sum_{i=1}^{n}\varphi\;-\sum_{i=1}^{n}y_i\quad=n\varphi-\sum_{i=1}^{n}y_i$

$=n\varphi-\;\sum_{i=1}^{n}\;1\{y_i\,=\,1\}$

log PΘ(D) respect to μ_i

$= \log \prod_{i=0}^{n} \exp\left(-\frac{1}{2}\left(x_i - \mu_{yi}\right)^{\mathrm{T}} \Gamma^{-1}\left(x_i - \mu_{yi}\right)\right) \mathrm{d}\,\mu_i$

$= \sum_{i=1}^{n} \log\left(\exp\left(-\frac{1}{2}\left(x_i - \mu_{yi}\right)^{\mathrm{T}} \Gamma^{-1}\left(x_i - \mu_{yi}\right)\right)\right) \mathrm{d}\mu_i$

$= \sum_{i=1}^{n} \left(-\frac{1}{2}(x_i - \mu_i)^{\mathrm{T}} \Gamma^{-1}(x_i - \mu_i)\right) d\mu_i = -\frac{1}{2} \sum_{i=1}^{n}\left(2\,(x_i - \mu_i)\right) = nu_i - \sum_{i=1}^{n} x_i$

$$nu_1 - \frac{\sum_{i=1}^{n} 1\{y_i = 1\}\, x}{\sum_{i=1}^{n} 1\{y_i = 1\}}\ , nu_0 - \frac{\sum_{i=1}^{n} 1\{y_i = 0\}\, x}{\sum_{i=1}^{n} 1\{y_i = 0\}}$$

log PΘ(D) respect to Γ

$= \log \prod_{i=0}^{n} \frac{1}{(2\pi)^{\frac{d}{2}}|\Gamma|^{\frac{1}{2}}} \log \prod_{i=0}^{n} \exp\left(-\frac{1}{2}(x_i - \mu_i)^{\mathrm{T}} \Gamma^{-1}(x_i - \mu_i)\right) d\,\Gamma =$

$\sum_{i=1}^{n} \left(\log\left(\frac{1}{(2\pi)^{\frac{d}{2}}|\Gamma|^{\frac{1}{2}}}\right)\right) + \sum_{i=1}^{n} \log\left(\exp\left(-\frac{1}{2}(x_i - \mu_i)^{\mathrm{T}} \Gamma^{-1}(x_i - \mu_i)\right)\right)$

$$= \sum_{i=1}^{n} (\log(2\,\pi)\mathrm{d} + \log(|\mathrm{T}|))/2 \quad + \sum_{i=1}^{n} \left(-\frac{1}{2}(x_i - \mu_i)^{\mathrm{T}} \Gamma^{-1}(x_i - \mu_i)\right) d\Gamma$$

$$= \sum_{i=1}^{n} \frac{1}{2\Gamma} - n(x_i - \mu_i)^{\mathrm{T}}(x_i - \mu_i) \sum_{i=1}^{n} \left(\frac{1}{2\Gamma^2}\right) = \frac{n}{2\Gamma} - \sum_{i=1}^{n} (x_i - \mu_i)^{\mathrm{T}}(x_i - \mu_i) * \frac{n}{2\Gamma^2}$$

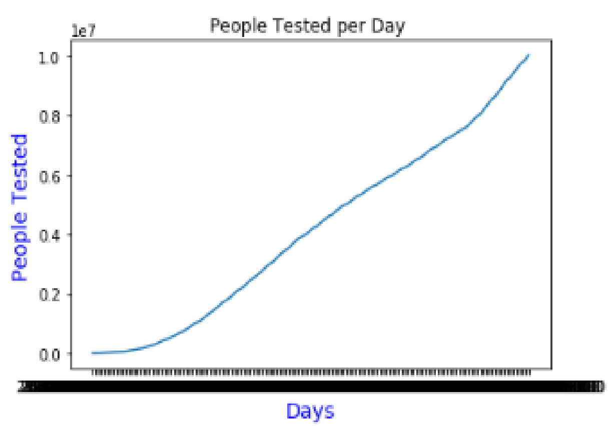
$$= \Gamma - \sum_{i=1}^{n} (x_i - \mu_i)^{\mathrm{T}}(x_i - \mu_i)$$

3.

a.

```
# 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 ####
```

```
In [7]: # 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 ####
```

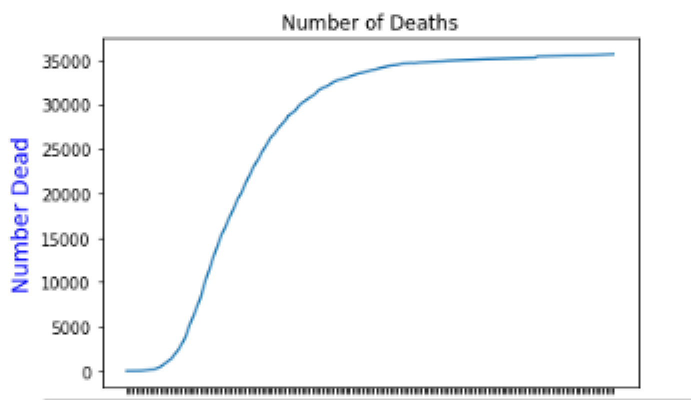



```
# Add code to plot the trend of total deaths as days progressed.
# X axis -> dates ('Dates')
# Y axis -> number of deaths ('Deaths')
```

```
### 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 ####
```

```
# 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 ####
```

```
Text(0.5,0,'Days')
```

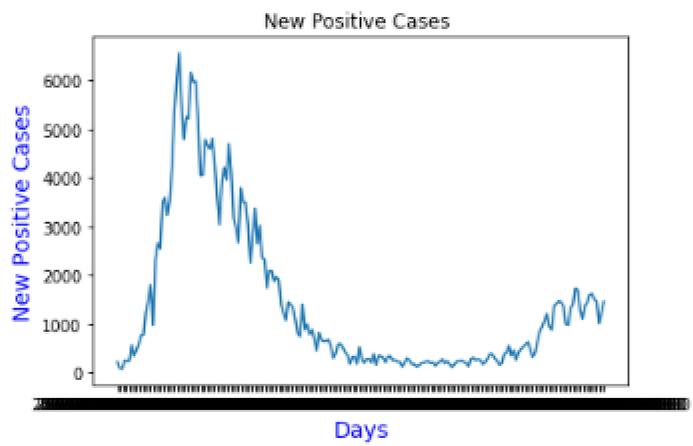



```
### STUDENT: Start of Code ###
x = dates.to_numpy()
y= data['New_positive_cases'].to_numpy()

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

```
### 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 ####
```

```
Text(0.5,0,'Days')
```



b.

```
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
```

```
In [23]: 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
```

```
In [16]: # Copy the outputs of this code to the solution file
         my_weights = np.array([1., 1.])

         test_predictions = predict_output(test_features, my_weights)
         print ("(normalized) prediction at day 5: ", test_predictions[5])
         print ("(normalized) prediction at day 20 ", test_predictions[20])

         (normalized) prediction at day 5:  0.038461538461538464
         (normalized) prediction at day 20  0.11057692307692307
```

c.

```
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 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 ###
```

```
# NOTE: copy the output to the solution file.

(example_features, example_output) = get_numpy_data(data, ['Days'], 'People_tested')

my_weights = np.array([0., 0.]) # this makes all the predictions 0
derivative = weight_derivative(my_weights, example_features,example_output)

print (derivative)
```

```
-1.1789675787713623
```

d.

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)
```

```
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)
```

e.

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)

```
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)
```

```
Iteration:  1 gradient_magnitude:  [0.06592743077975785, 0.003296371538987927]
Iteration:  2 gradient_magnitude:  [0.11407448303180767, 0.005703724151590385]
Iteration:  3 gradient_magnitude:  [0.15615585311634989, 0.007807792655817495]
Iteration:  4 gradient_magnitude:  [0.1948031169901319, 0.009740155849506596]
Iteration:  5 gradient_magnitude:  [0.23116017017581494, 0.011558008508790747]
Iteration:  6 gradient_magnitude:  [0.2658506994590075, 0.013292534972950376]
Iteration:  7 gradient_magnitude:  [0.29925989583773044, 0.014962994791886522]
Iteration:  8 gradient_magnitude:  [0.33164556477677176, 0.016582278238838587]
Iteration:  9 gradient_magnitude:  [0.36319029772564665, 0.018159514886282334]
Iteration:  10 gradient_magnitude:  [0.39402898441247575, 0.019701449220623786]
Iteration:  11 gradient_magnitude:  [0.42426458817639945, 0.021213229408081997]
Iteration:  12 gradient_magnitude:  [0.4539777852484522, 0.02269888926242261]
Iteration:  13 gradient_magnitude:  [0.48323315727271865, 0.024161657863635933]
Iteration:  14 gradient_magnitude:  [0.5120833342676021, 0.025604166713380103]
Iteration:  15 gradient_magnitude:  [0.5405718610198629, 0.027028593050993137]
Iteration:  16 gradient_magnitude:  [0.5687352371280266, 0.028436761856401323]
Iteration:  17 gradient_magnitude:  [0.5966044044150732, 0.02983022022075365]
Iteration:  18 gradient_magnitude:  [0.6242058542961136, 0.03121029271480567]
Iteration:  19 gradient_magnitude:  [0.6515624673808108, 0.03257812336904053]
Iteration:  20 gradient_magnitude:  [0.6786941603732565, 0.03393470801866281]
Iteration:  21 gradient_magnitude:  [0.705618391665625, 0.03528091958328124]
Iteration:  22 gradient_magnitude:  [0.732350561570377, 0.03661752807851884]
Iteration:  23 gradient_magnitude:  [0.7589043328088438, 0.03794521664044218]
Iteration:  24 gradient_magnitude:  [0.7852918898262533, 0.03926459449131266]
Iteration:  25 gradient_magnitude:  [0.8115241506021136, 0.04057620753010567]
Iteration:  26 gradient_magnitude:  [0.8376109411580704, 0.04188054705790351]
Iteration:  27 gradient_magnitude:  [0.8635611404749283, 0.043178057023746406]
Iteration:  28 gradient_magnitude:  [0.8893828017160705, 0.04446914008580352]
Iteration:  29 gradient_magnitude:  [0.9150832543153286, 0.04575416271576642]
Iteration:  30 gradient_magnitude:  [0.9406691904871081, 0.04703345952435539]
Iteration:  31 gradient_magnitude:  [0.9661467389612365, 0.048307336948061816]
Iteration:  32 gradient_magnitude:  [0.991521528168747, 0.04957607640843734]
Iteration:  33 gradient_magnitude:  [1.0167987406610173, 0.05083993703305086]
Here are the final weights after convergence:
[1.0167987406610173, 0.05083993703305086]
```

f.

```
# 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
```


```
# 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
```


```
0.002727328133312371
```

g.

```
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)
```


```
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" &

(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)
```


```
Here are the final weights after convergence:
[1.0439592334600856, 0.052197961673004264, 0.002609898083650214, 0.00013049490418251073]
```

h.

```python
(test_feature_matrix, test_output) = get_numpy_data(test_data, model_features, my_output)

test_predictions_2 = predict_output(test_feature_matrix, weight_2)

#Prediction for the 10th day of the forecasting period.
print (test_predictions_2[10])

#Convert the normalized data back to original figures using the same min-max normalization
prediction_10th_day = test_predictions_2[10] * (data_orig['People_tested'].max() - data_orig['People_tested'].min()) + data_orig[

print ("Model prediction of the 10th day:",int(prediction_10th_day))

# Get the actual number of people tested from our test data on 10 th day of forecasting period.
actual_people_tested = data_orig["People_tested"].iloc[190]

print ("Actual number of people tested on the 10th day:",actual_people_tested)
```

```
0.9988797902036922
Model prediction of the 10th day: 10033303
Actual number of people tested on the 10th day: 8725909
```

```
C:\Users\kuent\Anaconda3\lib\site-packages\ipykernel_launcher.py:9: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy
  if __name__ == '__main__':
```

```python
# Calculate the test error
# STUDENT: Start of code
test_error = 0;
size = len(test_predictions_2)

for i in range(size):
    diff= test_output[i]-test_predictions_2[i]
    test_error+=pow(diff,2)

print(test_error/size)

# end of code
```

```
0.009706376971805162
```

i.

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
```

```python
# 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" & "

(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
```

```
tolerance:  1
test_error:  0.009706376971805162
tolerance:  3
test_error:  0.009591340487839297
tolerance:  5
```

```
C:\Users\kuent\Anaconda3\lib\site-p
A value is trying to be set on a co
Try using .loc[row_indexer,col_inde

See the caveats in the documentatic
rsus-a-copy
  if __name__ == '__main__':
```

```
test_error:  0.009553031895550554
tolerance:  7
test_error:  0.00953341217290619
tolerance:  9
test_error:  0.009522001706408758
tolerance:  11
test_error:  0.009514178810549319
tolerance:  13
test_error:  0.009512453898728008
tolerance:  15
test_error:  0.009512453898728008
tolerance:  17
test_error:  0.009512453898728008
tolerance:  19
test_error:  0.009512453898728008
```

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