

Machine Learning HW2

charles.kewu

① throw a coin n times, $D = \{y_1, \dots, y_n\}$ where $y_i \in \{0, 1\}$ and 0-tails, 1-heads. interested in $\theta \in [0, 1]$ given D.

$$(a.) P_\theta(D) = \prod_{i=1}^n P(y_i)$$

$$= \prod_{i=1}^n (\theta_{y=1})^{y_i} (1 - \theta_{y=1})^{1-y_i}$$

$$= (\theta_{y=1}^{n_1} (1 - \theta_{y=1})^{n-n_1})$$

$$(b.) \ln(P_\theta(D)) = \ln \left[(\theta_{y=1})^{n_1} (1 - \theta_{y=1})^{n-n_1} \right]$$

$$= n_1 \ln(\theta_{y=1}) + (n-n_1) \ln(1 - \theta_{y=1})$$

(c.) looking for M.L.E. i.e. set derivative equal to zero
 $\frac{\partial}{\partial \theta} \ln(P_\theta(D)) = \frac{\partial}{\partial \theta_{y=1}} \left[n_1 \ln(\theta_{y=1}) + (n-n_1) \ln(1 - \theta_{y=1}) \right]$

$$\Rightarrow \theta \stackrel{?}{=} \frac{n_1}{\theta_{y=1}} = \frac{n - n_1}{1 - \theta_{y=1}}$$

$$\Rightarrow n_1 (1 - \theta_{y=1}) = (n - n_1) \theta_{y=1}$$

$$\Rightarrow n_1 - n_1 \theta_{y=1} = n \theta_{y=1} - n \theta_{y=1}$$

$$\Rightarrow \frac{n_1}{\theta_{y=1}} = \frac{n \theta_{y=1}}{n - n_1}$$

$$\text{sol } \Rightarrow$$

$$\begin{aligned}
 \text{(a.) } p(y=1|x) &= \frac{p(x|y=1)p(y=1)}{p(x)} = \frac{\cancel{p(x|y=1)}p(y=1)}{\cancel{p(x|y=0)}p(y=0)} \\
 &= \frac{1}{1 + \frac{p(x|y=0)p(y=0)}{p(x|y=1)p(y=1)}} \\
 &= \frac{1}{1 + \frac{1-\theta}{\theta} \exp \left[-\frac{1}{2} (\bar{x} - \mu_0)^T \bar{\Sigma}^{-1} (\bar{x} - \mu_0) + \frac{1}{2} (\bar{x} - \mu_1)^T \bar{\Sigma}^{-1} (\bar{x} - \mu_1) \right]}
 \end{aligned}$$

$$\begin{aligned}
 \ln(P_{\text{obs}}) &= \ln \prod_{i=1}^n [P(x_i | y_i) P(\bar{y}_i)] \\
 &= \ln \left[\frac{1}{\pi_{i=1}^n} \left(\frac{1}{12\pi\sigma^2} e^{-\frac{(x_i - \mu_0)^2}{2\sigma^2}} \right)^{\delta_{y_i=0}} \left(\frac{1}{2\pi\sigma^2} e^{-\frac{(x_i - \mu_1)^2}{2\sigma^2}} \right)^{\delta_{y_i=1}} \phi^{1-\delta_{y_i=0}} (1-\phi)^{\delta_{y_i=1}} \right] \\
 &= \sum_{i=1}^n \ln \left(\dots \right) + \dots \\
 &= \sum_{i=1}^n -\frac{1}{2} \ln \{ \delta_{y_i=0} \} \ln(2\pi) - \sum_{i=1}^n \delta_{y_i=0} \{ \ln(\sigma) - \frac{1}{2} \ln \{ \delta_{y_i=1} \} \ln(2\pi) - \sum_{j=1}^n \delta_{y_j=1} \ln(\sigma) + \sum_{j=1}^n \delta_{y_j=1} \ln(\phi) + \sum_{j=1}^n \delta_{y_j=0} \ln(1-\phi) \} \\
 &\quad \left[= \phi \right] 0 = \frac{\partial}{\partial \phi} (\dots) = \frac{\partial}{\partial \phi} \left(\sum_{i=1}^n \delta_{y_i=0} \{ \ln(\phi) + \sum_{j=1}^n \delta_{y_j=1} \} \right) \quad \text{all other terms are constant drop} \\
 &= \frac{\sum_{i=1}^n \delta_{y_i=0}}{\phi} - \frac{\sum_{i=1}^n \delta_{y_i=0}}{1-\phi} \\
 &\Rightarrow \frac{\sum_{i=1}^n \delta_{y_i=0}}{\phi} = \frac{\sum_{i=1}^n \delta_{y_i=0}}{1-\phi} \\
 &\Rightarrow \sum_{i=1}^n \delta_{y_i=0} \cdot (1-\phi) = (n - \sum_{i=1}^n \delta_{y_i=0}) \phi \\
 &\Rightarrow \sum_{i=1}^n \delta_{y_i=0} = n\phi - \sum_{i=1}^n \delta_{y_i=0} \phi + \sum_{i=1}^n \delta_{y_i=0} \\
 &\Rightarrow \sum_{i=1}^n \delta_{y_i=0} = n\phi \\
 &\Rightarrow \phi = \frac{\sum_{i=1}^n \delta_{y_i=0}}{n} \\
 &\text{QED}
 \end{aligned}$$

$$\begin{aligned}
 \ln(P_0(\sigma)) &= \ln \prod_{i=1}^n [P(x_i | y_i) P(\bar{y}_i)] \\
 &= \ln \left[\prod_{i=1}^n \left(\frac{1}{2\pi\sigma^2} e^{-\frac{(x_i - \mu_i)^2}{2\sigma^2}} \right) \right] \quad \text{from } P(x_i | y_i) = \frac{e^{-(x_i - \mu_i)^2/(2\sigma^2)}}{\sqrt{2\pi\sigma^2}} \\
 &\quad \downarrow \\
 &= \sum_{i=1}^n \ln \left(\frac{1}{2\pi\sigma^2} e^{-\frac{(x_i - \mu_i)^2}{2\sigma^2}} \right) + \ln(\bar{y}_1) + \ln(\bar{y}_2) + \dots \\
 &= \sum_{i=1}^n -\frac{1}{2} \ln(2\pi\sigma^2) - \frac{1}{2} \ln((x_i - \mu_i)^2) + \frac{1}{2\sigma^2} \ln(e^{-\frac{(x_i - \mu_i)^2}{2\sigma^2}}) \\
 &\quad \downarrow \\
 &= \sum_{i=1}^n \left[-\frac{1}{2} \ln(\sigma^2) - \frac{1}{2} \ln((x_i - \mu_i)^2) + \frac{(x_i - \mu_i)^2}{2\sigma^2} \right] \\
 &= \sum_{i=1}^n \left[\sigma^2 \left(-\frac{1}{2} \ln(y_i) + \frac{(x_i - \mu_i)^2}{2\sigma^2} \right) + \left(-y_i \cdot \sigma^2 \right) \right]
 \end{aligned}$$

whether $y_i = 0$ or $y_i = 1$, we get "sum $\sum_{i=1}^n (-\frac{1}{2} \ln(\sigma^2) + \frac{(x_i - \mu_i)^2}{2\sigma^2})$ " term and "term $\sum_{i=1}^n (x_i - \mu_i)(x_i - \mu_i)$ " term

$$\begin{aligned}
 0 &= \frac{\partial}{\partial \sigma^2} \sum_{i=1}^n \left(-\frac{1}{2} \ln(\sigma^2) - \frac{(x_i - \mu_i)^2}{2\sigma^2} \right) \\
 0 &= \frac{-n}{2\sigma^2} + \frac{\sum_{i=1}^n (x_i - \mu_i)(x_i - \mu_i)}{2\sigma^2} \\
 0 &= \frac{n}{2\sigma^2} - \frac{\sum_{i=1}^n (x_i - \mu_i)(x_i - \mu_i)}{2\sigma^2} \\
 \sigma^2 &= \frac{\sum_{i=1}^n (x_i - \mu_i)(x_i - \mu_i)}{n}
 \end{aligned}$$

QED

Programming Assignment 2: Linear regression

As we know, the COVID-19 pandemic has been disrupting people's life around the world. To get it under control, a crucial aspect is to be able to accurate forecast the spread of the disease, which can be helpful as a planning tool for policymakers, clinicians, and public health officers to deal with this crisis. In this notebook, we will try to do some forecasting on the covid-19 epidemic progression using machine learning. We will use a dataset based on the COVID-19 Data Repository at John Hopkins university.

```
In [1]: import pandas as pd
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import numpy as np
from math import sqrt
import sys
import operator
from progressbar import ProgressBar
```

```
In [2]: print (sys.version)
```

```
3.8.3 (default, Jul 2 2020, 17:30:36) [MSC v.1916 64 bit (AMD64)]
```

First, let us load the data.

```
In [3]: data_orig = pd.read_csv("us_covid_data.csv")
data = data_orig.copy()
print (data_orig.columns)
data_orig
```

```
Index(['Unnamed: 0', 'date', 'Country', 'hospitalized_with_symptom',
       'Intensive_care', 'Total_hospitalized', 'Home_Isolation',
       'Total_positive', 'Daily_change_in_positive_cases',
       'New_positive_cases', 'Recovered', 'Deaths', 'Total_cases',
       'People_tested'],
      dtype='object')
```

Out[3]:

	Unnamed: 0	date	Country	hospitalized_with_symptom	Intensive_care	Total_hospitaliz
0	0	2020-02-24T18:00:00	US	101	26	1
1	1	2020-02-25T18:00:00	US	114	35	1
2	2	2020-02-26T18:00:00	US	128	36	1
3	3	2020-02-27T18:00:00	US	248	56	2
4	4	2020-02-28T18:00:00	US	345	64	2
...
201	201	2020-09-12T17:00:00	US	1951	182	21
202	202	2020-09-13T17:00:00	US	2042	187	22
203	203	2020-09-14T17:00:00	US	2122	197	23
204	204	2020-09-15T17:00:00	US	2222	201	24
205	205	2020-09-16T17:00:00	US	2285	207	24

206 rows × 14 columns

```
In [4]: # change the date format
dates = data['date']
date_format = [pd.to_datetime(d) for d in dates]
```

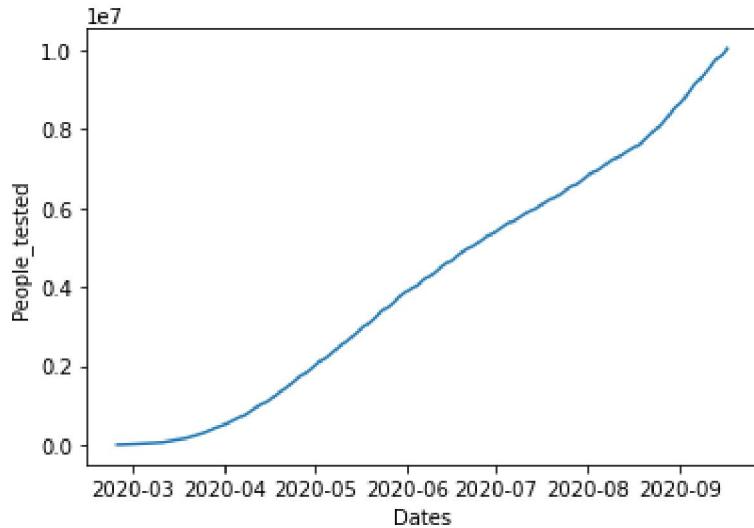
Data Visualization

Task P1: complete the following **three** visualization graphs that show the trend of the epidemic progression. Copy them to the solution file.

Graph 1: plot the total number of people tested for the entire period of the dataset. Your X axis will be the dates ("Dates") and Y-axis will be the total number of cases ("People_tested") over the period of time.

```
In [5]: # 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 ###  
plt.plot(date_format, data['People_tested'])  
plt.xlabel('Dates')  
plt.ylabel('People_tested')  
plt.show  
  
### End of code #####
```

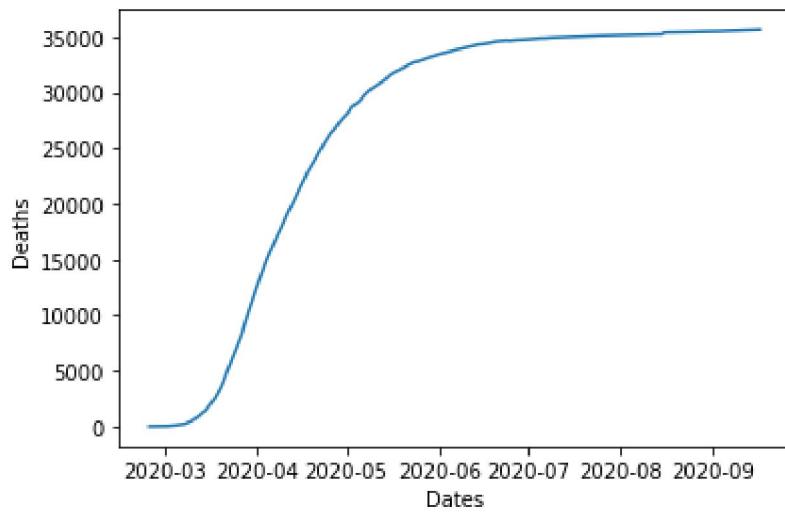
Out[5]: <function matplotlib.pyplot.show(*args, **kw)>



Graph 2: plot the total number of deaths for the entire period. Your X axis will be the dates ("Dates") and Y-axis will be the total number of death cases("Deaths") over the period of time.

```
In [6]: # 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 ###  
plt.plot(date_format, data['Deaths'])  
plt.xlabel('Dates')  
plt.ylabel('Deaths')  
plt.show  
  
### End of code #####
```

Out[6]: <function matplotlib.pyplot.show(*args, **kw)>



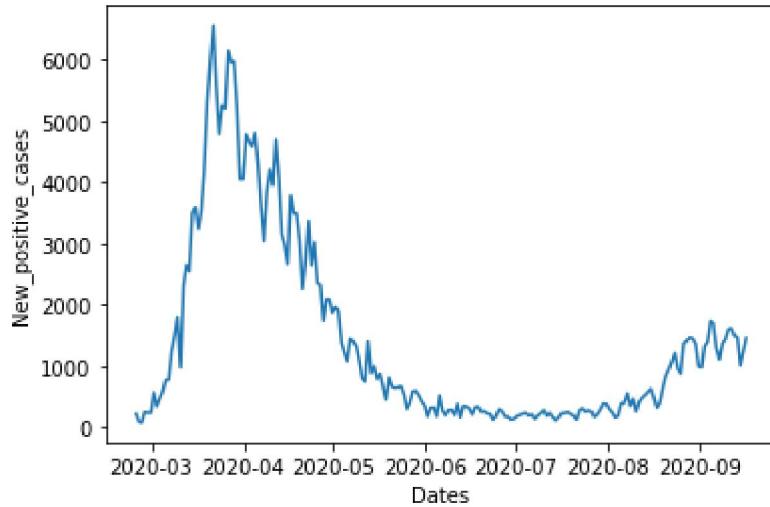
Graph 3: plot the total number of infected cases for the entire period. Your X axis will be the dates ("Dates") and Y-axis will be the total number of infected cases ('New_positive_cases') over the period of time.

In [7]: *### STUDENT: Start of Code ###*

```
plt.plot(date_format, data[ 'New_positive_cases' ])
plt.xlabel('Dates')
plt.ylabel('New_positive_cases')
plt.show
```

End of code

Out[7]: <function matplotlib.pyplot.show(*args, **kw)>



As we can see that our data has different ranges of values for every feature and this can cause problems in our model, so here we will normalize our data (ignoring the categorical variables) so that our data is scaled between 0 and 1. The downside, however, is that the numbers are no longer interpretable. To interpret it, you need to multiply back by the scaling factor.

IMPORTANT: From now on, we will work with the normalized features to build the regression model. However, in **Task P8**, you need to convert the number back to the actual units.

```
In [8]: data_list = data_orig.columns.values.tolist()

for i in data_list[-11:]:
    data[[i]]=(data_orig[i]-data_orig[i].min())/(data_orig[i].max()-data_orig[i].min())

data
```

Out[8]:

		Unnamed: 0	date	Country	hospitalized_with_symptom	Intensive_care	Total_hospitaliz
0	0	2020-02-24T18:00:00	US		0.000000	0.000000	0.0000
1	1	2020-02-25T18:00:00	US		0.000450	0.002227	0.0007
2	2	2020-02-26T18:00:00	US		0.000934	0.002474	0.0011
3	3	2020-02-27T18:00:00	US		0.005085	0.007422	0.0053
4	4	2020-02-28T18:00:00	US		0.008440	0.009401	0.0085
...
201	201	2020-09-12T17:00:00	US		0.063994	0.038595	0.0610
202	202	2020-09-13T17:00:00	US		0.067142	0.039832	0.0639
203	203	2020-09-14T17:00:00	US		0.069909	0.042306	0.0666
204	204	2020-09-15T17:00:00	US		0.073368	0.043295	0.0698
205	205	2020-09-16T17:00:00	US		0.075547	0.044780	0.0719

206 rows × 14 columns

Calculate the feature matrix

The following is a function that accepts a list of feature names (e.g. ['Total_Hospitalized', 'People_tested']) and an target feature e.g. ('Deaths') and returns two things:

1. A numpy matrix whose columns are the desired features plus a column with a constant value 1, which is also known as the 'intercept'.
2. A numpy array that contains the values of the target output.

```
In [9]: def get_numpy_data(data_frame, features, output):
    # Steps
    # select the columns of data_Frame given by the features list into the variable features_sframe which will include the constant
    # Convert the features_frame into a numpy matrix
    # assign the column of data_frame associated with the output to the array output_array
    # convert the array into a numpy array by first converting it to a list
    # return feature_matrix,output_array

    data_frame['constant'] = 1 # here we are adding a constant column
    # add the column 'constant' to the front of the features list.
    features = ['constant'] + features

    # select the columns of data_Frame given by the features list into the variable features_sframe which will include the constant)
    features_frame = data_frame[features]

    # Convert the features_frame into a numpy matrix
    feature_matrix = features_frame.to_numpy()
    # print ("feature_matrix:", feature_matrix)

    # assign the column of data_frame associated with the output to the array output_array
    output_array = data_frame[output]

    # convert the array into a numpy array by first converting it to a list
    output_array = output_array.to_numpy()

    return(feature_matrix, output_array)
```

For dates, we need to convert them into a sequence of numbers. We now add a new column to our dataframe corresponding to the number of days since the start of the dataset.

```
In [10]: X = date_format
day_numbers = []
for i in range(1, len(X) + 1):
    day_numbers.append([i])

data['Days'] = pd.DataFrame(day_numbers,columns = ['Days'])
data["Days"] = data["Days"].astype(int)
```

Test the above function for a particular input and output feature.

```
In [11]: (example_features, example_output) = get_numpy_data(data, ['Days'], 'New_positive_cases')
print (example_features[0,:])
print (example_output[0])

[1 1]
0.022071307300509338
```

Predict the outputs with given regression weights

Suppose we had the weights $[1, 1]$ corresponding to the features $[1, 100]$, to compute the predicted output, we can simply take the dot product between them, so the output is $1 * 1 + 1 * 100 = 101$. Now, let's create the data with

```
In [12]: (test_features, output) = get_numpy_data(data, ['Days'], 'People_tested')
```

Task P2: Complete the following function 'predict_output'. Copy the the outputs of the code to the solution file.

```
In [13]: 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 = []
    for feature_row in feature_matrix:
        predictions.append(np.dot(feature_row, weights))
    return np.array(predictions)
    ## end of code
```

```
In [14]: # Copy the outputs of this code to the solution file
my_weights = np.array([1., 1.])
test_predictions = predict_output(example_features, my_weights)
# print(example_features)
print ("(normalized) prediction at day 5: ", test_predictions[5])
print ("(normalized) prediction at day 20 ", test_predictions[20])

(normalized) prediction at day 5:  7.0
(normalized) prediction at day 20  22.0
```

Compute the derivative

We will now compute the derivative of the regression cost function:

$$L_D(w) = \frac{1}{n} \sum_{i=1}^n (y_i - w \cdot x_i)^2,$$

$$\frac{\partial}{\partial w} L_D(w) = \frac{2}{n} \sum_{i=1}^n -x_i \cdot (y_i - w \cdot x_i),$$

where $x_i \in \mathbf{R}^d$ is the input feature of dimension d , $y_i \in \mathbf{R}$ is the output response, and $w \in \mathbf{R}^d$ is the regression weights.

Task P3: Complete the function 'weight_derivative' to calculate the derivative of the cost function with respect to regression weights w , i.e., $\frac{\partial}{\partial w} L_D(w)$. Note that this should be a d dimensional vector. Also copy the output of the code for the test example to the solution file.

```
In [15]: 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 #####
    L_D = np.zeros(len(feature_matrix[0]))
    prediction = predict_output(feature_matrix, weights)
    for i in range(len(feature_matrix)):
        scalar = np.dot(weights, feature_matrix[i]) - labels[i]
        L_D += scalar * feature_matrix[i]
    return (2 / len(feature_matrix)) * L_D
    # End of code #####
```

```
In [16]: # 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)
```

[-0.82103242 -120.60087518]

Gradient descent algorithm

Here, we will write a function to perform gradient descent algorithm on the linear regression cost. Given an initial point, we will update the current weights by moving in the negative gradient direction to minimize the cost function. Thus, in each iteration we obtain the updated weight w_{t+1} from the current iterate w_t as follows:

$$w_{t+1} = w_t - h \frac{\partial}{\partial w} L_D(w_t),$$

where h is the 'step_size' that is the amount by which we move in the negative gradient direction.

We stop when we are sufficiently close to the optimum (where gradient is the zero vector) by checking the condition with respect to the magnitude (length) of the gradient vector:

$$\left\| \frac{\partial}{\partial w} L_D(w_t) \right\|_2 \leq \epsilon,$$

where ϵ is the 'tolerance' parameter.

Task P4: Complete the code section to perform the gradient decent in the function `regression_gradient_descent`. Copy the code to the solution file.

```
In [17]: 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: tolerance 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 implementation of what the gradient descent algorithm does in every iteration
        # Refer back to the update rule listed above: update the weight
        d_weight = weight_derivative(weights, feature_matrix, labels)
        correction = step_size * d_weight
        weights = np.subtract(weights, correction)

        # Compute the gradient magnitude:
        gradient_magnitude = np.linalg.norm(d_weight)

        # Check the stopping condition to decide whether you want to stop the iterations
        if gradient_magnitude <= tolerance: # 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(weights)
```

Use gradient descent for linear regression

Let's test the gradient descent algorithm for linear regression with a single feature ('Day'). Here we are using first 180 days' data as our training data.

```
In [18]: #train_data
train_data = data[:180]
```

Task P5: Specify the initial_weights, step_size, and tolerance for the function regression_gradient_descent . Copy the outputs of the code to the solution file.

```
In [19]: 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
initial_weights = [0, 0]
step_size = 1e-5
tolerance = 0.75
# 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)
```

Here are the final weights after convergence:
[2.30146386e-05 4.01343741e-03]

<ipython-input-9-ea0b874cb38f>: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-versus-a-copy
`data_frame['constant'] = 1 # here we are adding a constant column`

Task P6: Use the learned weights to predict 'People tested' in the last three weeks in the dataset. Copy the predictions to the solution file, and calculate the test error $\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$ where n is the number of test data, y_i is the true label, \hat{y}_i is the predicted label.

```
In [20]: # Create the test data
test_data = data.iloc[-21:]
# print(test_data["People_tested"])
(test_simple_feature_matrix, test_output) = get_numpy_data(test_data, simple_features, my_output)
test_predictions = predict_output(test_simple_feature_matrix, final_weights)
print (test_predictions)
```

```
[0.74652237 0.75053581 0.75454925 0.75856269 0.76257612 0.76658956
 0.770603 0.77461644 0.77862987 0.78264331 0.78665675 0.79067019
 0.79468362 0.79869706 0.8027105 0.80672393 0.81073737 0.81475081
 0.81876425 0.82277768 0.82679112]
```

```
<ipython-input-9-ea0b874cb38f>: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-versus-a-copy
`data_frame['constant'] = 1 # here we are adding a constant column`

```
In [21]: # Calculate the test error
# STUDENT: Start of code
error = 0
for i in range(len(test_predictions)):
    error += (test_data["People_tested"][185 + i] - test_predictions[i]) ** 2
error /= len(test_predictions)
print(error)
#end of code
```

```
0.017328326473639064
```

Linear regression using multiple features

Here, we will be considering multiple input features (`Intensive_care` , `New_positive_cases` , `Days`) to predict the `People_tested` in the future.

Task P7: Specify the `initial_weights`, `step_size`, and tolerance for the function `regression_gradient_descent`. Print the outputs of the code.

```
In [22]: model_features = ['Intensive_care', 'New_positive_cases', 'Days']
my_output = 'People_tested'

#call the get_numpy_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
initial_weights = [-0.15, -0.05, 0.5, 0.88]
step_size = 1e-5
tolerance = 1.5
# 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:
[-0.15637694 -0.05073187 0.49912599 0.00496001]

Task P8: Use the learned weights to predict 'People_tested' in the last three weeks in the dataset. Find the value of the model predictions on the 10th day of the forecasting period. Also print the actual number of people tested on that particular day. Copy the predictions to the solution file, and calculate the test error. Note: here we are asking you to report the number before normalization. So you need to convert the prediction back to the unit of people.

```
In [23]: (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['People_tested'].min()

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)
```

0.9083511780398816
Model prediction of the 10th day: 9124376
Actual number of people tested on the 10th day: 8725909

<ipython-input-9-ea0b874cb38f>: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/table/user_guide/indexing.html#returning-a-view-versus-a-copy
data_frame['constant'] = 1 # here we are adding a constant column

```
In [24]: # Calculate the test error
# STUDENT: Start of code
error_m = 0
norm = data_orig['People_tested'].max() - data_orig['People_tested'].min()
intercept = data_orig['People_tested'].min()
for i in range(len(test_predictions_2)):
    error_m += ((test_data["People_tested"][185 + i] - test_predictions_2[i])
    * norm + intercept) ** 2
error_m /= len(test_predictions)
print(error_m)

# end of code
```

71041222945.10501

Explore on your own

Now that you have tried two models for predictions, in this section, you can explore on your own an aspect of the problem that interests you. Here are some examples:

- What features or what combination of features are most predictive for 'People_tested'?
- How does tolerance for convergence affect prediction errors?
- How does step size affect prediction errors?
- How can we use validation to select the set of features to improve prediction?

Report your question of investigation, as well as your results/interpretation in the solution file.

Q: what is the optimal initial weights that provides the best gradient descent results Method:

- iterate through a range of values for the initial weights
- calculate the test error for each combination of weights
- refocus the range of initial weights

Results:

- first iteration all values (0, 1, 0.2) yielded [0.0, 0.0, 0.6, 0.8]
- second iteration with [0, 0, (0.5, 0.7), (0.7, 0.9)] yielded [0, 0, 0.57, 0.87]
- third iteration with [(-0.1, 0.1), (-0.1, 0.1), (0.5, 0.6), (0.8, 0.9)] yielded (108238227940.11084, [-0.1, -0.1, 0.5, 0.88])
- fourth iteration with [(-0.3, 0.), (-0.3, 0.), (0.5, 0.6), (0.8, 0.9)] yielded (71041222945.10503, [-0.15, -0.05, 0.5, 0.88])

```
In [25]: # Explore an aspect of the model that interests you
### STUDENT: Start of code
step_size = 1e-5
tolerance = 1.5
errors = []

initial_weight_matrix = [[i for i in np.arange(-0.3, 0., 0.05)], [j for j in np.arange(-0.3, 0., 0.05)], [k for k in np.arange(0.5, 0.6, 0.01)], [l for l in np.arange(0.8, 0.9, 0.01)]] 

for x in initial_weight_matrix[0]:
    for y in initial_weight_matrix[1]:
        for z in initial_weight_matrix[2]:
            for w in initial_weight_matrix[3]:
                initial_weights = [x, y, z, w]
                print(initial_weights, "
", end='\\r')
                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)
                error_n = 0
                for i in range(len(test_predictions_2)):
                    error_n += ((test_data["People_tested"])[185 + i] - test_predictions_2[i]) * norm + intercept) ** 2
                error_n /= len(test_predictions)
                errors.append( (error_n, initial_weights) )

### End of code
```

```
<ipython-input-9-ea0b874cb38f>: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/table/user_guide/indexing.html#returning-a-view-versus-a-copy
data_frame['constant'] = 1 # here we are adding a constant column
[-0.0500000000000044, -0.0500000000000044, 0.5900000000000001, 0.8900000000000001]
```

```
In [26]: errors.sort(key = operator.itemgetter(0))
print("optimized initial weights:")
print("minimized error:", errors[0][0])
print("initial weights:", errors[0][1])
```

```
optimized initial weights:
minimized error: 71041222945.10503
initial weights: [-0.1500000000000002, -0.0500000000000044, 0.5, 0.8800000000000001]
```

In []: