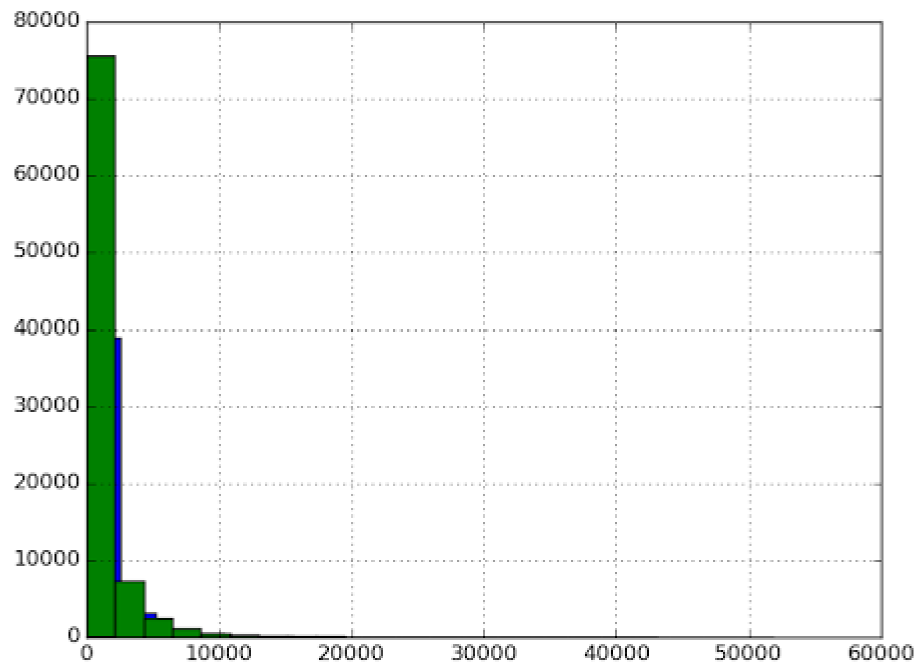
Problem Set3: Analyzing Subway Data

1 - Exploratory Data Analysis:

```
import numpy as np
import pandas
import matplotlib.pyplot as plt

def entries_histogram(turnstile_weather):
'''
   Before we perform any analysis, it might be useful to take a
   look at the data we're hoping to analyze. More specifically, let's
   examine the hourly entries in our NYC subway data and determine what
   distribution the data follows. This data is stored in a dataframe
   called turnstile_weather under the ['ENTRIESn_hourly'] column.

   Let's plot two histograms on the same axes to show hourly
   entries when raining vs. when not raining. Here's an example on how
   to plot histograms with pandas and matplotlib:
   turnstile_weather['column_to_graph'].hist()

   Your histograph may look similar to bar graph in the instructor notes below.

   You can read a bit about using matplotlib and pandas to plot histograms here:
   http://pandas.pydata.org/pandas-docs/stable/visualization.html#histograms

   You can see the information contained within the turnstile weather data here:
   https://www.dropbox.com/s/meyki2wl9xfa7yk/turnstile_data_master_with_weather.csv
   '''
x=turnstile_weather["ENTRIESn_hourly"][turnstile_weather["rain"] == 1] # your code here to plot a historgram for hourly entries when it is raining
   y=turnstile_weather["ENTRIESn_hourly"][turnstile_weather["rain"] == 0] # your code here to plot a historgram for hourly entries when it is not raining
   plt.figure()
   x.hist(bins=20)
   y.hist(bins=20)
   return plt
'''
Does the data seem normally distributed? - NO
Do you think we would be able to use Welch's t-test on this data? - No - Distribution is not normal
'''
```

```
The image produced by your code is shown below:
```

```
Does the data seem normally distributed?

Do you think we would be able to use Welch's t-test on this data?
```



2 - Welch's t-Test?

1) Does entries data from the previous exercise seem normally distributed?
   Ans: No
2) Can we run Welch's T test on entries data? Why or why not?
   Ans: Not at all, this test distribution is not normal so we can't use Welch's t test on entries data.

3 - Mann-Whitney U-Test:

```
import numpy as np
import scipy
import scipy.stats
import pandas
def mann_whitney_plus_means(turnstile_weather):
'''
    This function will consume the turnstile_weather dataframe containing
    our final turnstile weather data.
    You will want to take the means and run the Mann Whitney U-test on the
    ENTRIESn_hourly column in the turnstile_weather dataframe.
   This function should return:
       1) the mean of entries with rain
       2) the mean of entries without rain
       3) the Mann-Whitney U-statistic and p-value comparing the number of entries
          with rain and the number of entries without rain
    You should feel free to use scipy's Mann-Whitney implementation, and you
    might also find it useful to use numpy's mean function.
    Here are the functions' documentation:
    http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html
    http://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html

    You can look at the final turnstile weather data at the link below:
    https://www.dropbox.com/s/meyki2wl9xfa7yk/turnstile_data_master_with_weather.csv
    '''
### YOUR CODE HERE ###
rainy = turnstile_weather[turnstile_weather['rain'] == 1]
with_rain_mean = np.mean(rainy['ENTRIESn_hourly'])
not_rainy = turnstile_weather[turnstile_weather['rain'] == 0]
without_rain_mean = np.mean(not_rainy['ENTRIESn_hourly'])
 U , p = scipy.stats.mannwhitneyu(rainy['ENTRIESn_hourly'], not_rainy['ENTRIESn_hourly'])

 return with_rain_mean, without_rain_mean, U, p # leave this line for the grader
```

```
Good job! Your calculations are correct.

Here's your output:

(1105.4463767458733, 1090.278780151855, 1924409167.0, 0.024999912793489721)

Here's the correct output:

(1105.4463767458733, 1090.278780151855, 1924409167.0, 0.024999912793489721)
```

4 - Ridership on Rainy vs. Nonrainy Days:

1) Is the distribution of the number of entries statistically different between rainy & non rainy days?
   Ans: ***Yes***
2) Describe your results and the methods used
   Ans: ***np.mean method used for getting entries for Rainy and non-rainy day than assign to variables and used this values to run Mann Whitney U-test and gets U-statistic and p-value to compare rainy and non-rainy days entries.***

5 - Linear Regression:

```
import numpy as np
import pandas
from ggplot import *

"""
In this question, you need to:
1) implement the compute_cost() and gradient_descent() procedures
2) Select features (in the predictions procedure) and make predictions.

"""
def normalize_features(df):
    """
    Normalize the features in the data set.
    """
        mu = df.mean()
        sigma = df.std()

         if (sigma == 0).any():
        raise Exception("One or more features had the same value for all samples, and thus could " + \
                        "not be normalized. Please do not include features with only a single value " + \
                         "in your model.")
         df_normalized = (df - df.mean()) / df.std()

        return df_normalized, mu, sigma

def compute_cost(features, values, theta):
    """
    Compute the cost function given a set of features / values,
    and the values for our thetas.

    This can be the same code as the compute_cost function in the lesson #3 exercises,
```

```
    but feel free to implement your own.
    """

    # your code here
        m = len(values)
        sum_of_square_errors = np.square(np.dot(features, theta) - values).sum()
        cost = sum_of_square_errors / (2*m)
        return cost

def gradient_descent(features, values, theta, alpha, num_iterations):
    """
    Perform gradient descent given a data set with an arbitrary number of features.

    This can be the same gradient descent code as in the lesson #3 exercises,
    but feel free to implement your own.
    """
        m = len(values)
        cost_history = []

        for i in range(num_iterations):
         # your code here
                predicted_values = np.dot(features, theta)
                theta -= (alpha / m) * np.dot((predicted_values - values), features)
                 cost_history.append(compute_cost(features, values, theta))
        return theta, pandas.Series(cost_history)

def predictions(dataframe):
    '''
    The NYC turnstile data is stored in a pandas dataframe called weather_turnstile.
    Using the information stored in the dataframe, let's predict the ridership of
    the NYC subway using linear regression with gradient descent.

    You can download the complete turnstile weather dataframe here:
    https://www.dropbox.com/s/meyki2wl9xfa7yk/turnstile_data_master_with_weather.csv

    Your prediction should have a R^2 value of 0.20 or better.
    You need to experiment using various input features contained in the dataframe.
    We recommend that you don't use the EXITSn_hourly feature as an input to the
    linear model because we cannot use it as a predictor: we cannot use exits
    counts as a way to predict entry counts.

    Note: Due to the memory and CPU limitation of our Amazon EC2 instance, we will
    give you a random subet (~15%) of the data contained in
```

turnstile_data_master_with_weather.csv. You are encouraged to experiment with this computer on your own computer, locally.

If you'd like to view a plot of your cost history, uncomment the call to plot_cost_history below. The slowdown from plotting is significant, so if you are timing out, the first thing to do is to comment out the plot command again.

If you receive a "server has encountered an error" message, that means you are hitting the 30-second limit that's placed on running your program. Try using a smaller number for num_iterations if that's the case.

If you are using your own algorithm/models, see if you can optimize your code so that it runs faster.

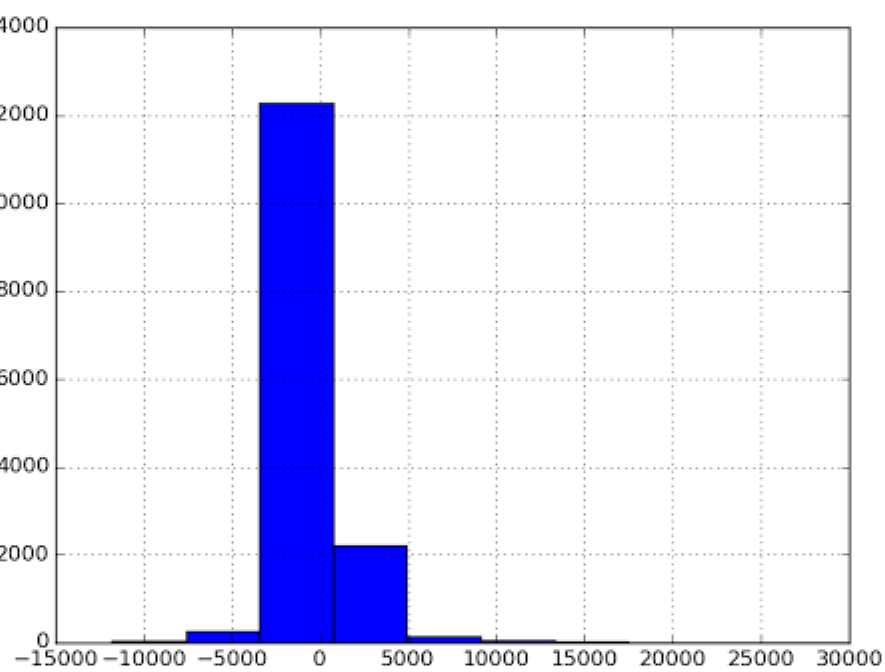
```
'''
# Select Features (try different features!)
features = dataframe[['rain', 'precipi', 'Hour', 'meantempi']]

# Add UNIT to features using dummy variables
dummy_units = pandas.get_dummies(dataframe['UNIT'], prefix='unit')
features = features.join(dummy_units)

# Values
values = dataframe['ENTRIESn_hourly']
m = len(values)

features, mu, sigma = normalize_features(features)
features['ones'] = np.ones(m) # Add a column of 1s (y intercept)

# Convert features and values to numpy arrays
features_array = np.array(features)
values_array = np.array(values)

# Set values for alpha, number of iterations.
alpha = 0.1 # please feel free to change this value
num_iterations = 75 # please feel free to change this value

# Initialize theta, perform gradient descent
theta_gradient_descent = np.zeros(len(features.columns))
theta_gradient_descent, cost_history = gradient_descent(features_array,
                                                        values_array,
                                                        theta_gradient_descent,
                                                        alpha,
```

```
                                        num_iterations)

    plot = None
    # --------------------------------------------------
    # Uncomment the next line to see your cost history
    # --------------------------------------------------
    # plot = plot_cost_history(alpha, cost_history)
    #
    # Please note, there is a possibility that plotting
    # this in addition to your calculation will exceed
    # the 30 second limit on the compute servers.

    predictions = np.dot(features_array, theta_gradient_descent)
    return predictions, plot


def plot_cost_history(alpha, cost_history):
   """This function is for viewing the plot of your cost history.
   You can run it by uncommenting this

      plot_cost_history(alpha, cost_history)

   call in predictions.

   If you want to run this locally, you should print the return value
   from this function.
   """
   cost_df = pandas.DataFrame({
      'Cost_History': cost_history,
      'Iteration': range(len(cost_history))
   })
   return ggplot(cost_df, aes('Iteration', 'Cost_History')) + \
      geom_point() + ggtitle('Cost History for alpha = %.3f' % alpha )
```

```
Your r^2 value is 0.463968815042
```

6 - Plotting Residuals:

```
import numpy as np
import scipy
import matplotlib.pyplot as plt

def plot_residuals(turnstile_weather, predictions):
    '''
    Using the same methods that we used to plot a histogram of entries
    per hour for our data, why don't you make a histogram of the residuals
    (that is, the difference between the original hourly entry data and the predicted values).
    Try different binwidths for your histogram.

    Based on this residual histogram, do you have any insight into how our model
    performed?  Reading a bit on this webpage might be useful:

    http://www.itl.nist.gov/div898/handbook/pri/section2/pri24.htm
    '''
    plt.figure()
    (turnstile_weather['ENTRIESn_hourly'] - predictions).hist()
    return plt
```



7 - Compute R^2:

```
import numpy as np
import scipy
import matplotlib.pyplot as plt
import sys

def compute_r_squared(data, predictions):
'''
    In exercise 5, we calculated the R^2 value for you. But why don't you try and
    and calculate the R^2 value yourself.

    Given a list of original data points, and also a list of predicted data points,
    write a function that will compute and return the coefficient of determination (R^2)
    for this data.  numpy.mean() and numpy.sum() might both be useful here, but
    not necessary.

    Documentation about numpy.mean() and numpy.sum() below:
    http://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html
    http://docs.scipy.org/doc/numpy/reference/generated/numpy.sum.html
    '''

    # your code here
mean = np.mean(data)
StSr = np.sum(np.square(data - predictions))
StSt = np.sum(np.square(data - mean))
r_squared = 1.0 - (StSr / StSt)
return r_squared
```

```
You calculated R^2 value correctly!

Your calculated R^2 value is: 0.318137233709
```

8 - More Linear Regression (Optional):

```
# -*- coding: utf-8 -*-

import numpy as np
import pandas
import scipy
import statsmodels.api as sm

"""
In this optional exercise, you should complete the function called
predictions(turnstile_weather). This function takes in our pandas
turnstile weather dataframe, and returns a set of predicted ridership values,
based on the other information in the dataframe.

In exercise 3.5 we used Gradient Descent in order to compute the coefficients
theta used for the ridership prediction. Here you should attempt to implement
another way of computing the coeffcients theta. You may also try using a reference implementation such as:
http://statsmodels.sourceforge.net/devel/generated/statsmodels.regression.linear_model.OLS.html

One of the advantages of the statsmodels implementation is that it gives you
easy access to the values of the coefficients theta. This can help you infer relationships
between variables in the dataset.

You may also experiment with polynomial terms as part of the input variables.

The following links might be useful:
http://en.wikipedia.org/wiki/Ordinary_least_squares
http://en.wikipedia.org/w/index.php?title=Linear_least_squares_(mathematics)
http://en.wikipedia.org/wiki/Polynomial_regression

This is your playground. Go wild!

How does your choice of linear regression compare to linear regression
with gradient descent computed in Exercise 3.5?

You can look at the information contained in the turnstile_weather dataframe below:
https://www.dropbox.com/s/meyki2wl9xfa7yk/turnstile_data_master_with_weather.csv

Note: due to the memory and CPU limitation of our amazon EC2 instance, we will
give you a random subset (~10%) of the data contained in turnstile_data_master_with_weather.csv

If you receive a "server has encountered an error" message, that means you are hitting
```

the 30 second limit that's placed on running your program. See if you can optimize your code so it runs faster.

```
"""
def normalize_features(array):
   """
   Normalize the features in the data set.
   """
        array_normalized = (array-array.mean())/array.std()
        mu = array.mean()
        sigma = array.std()

        return array_normalized, mu, sigma

def compute_cost(features, values, theta):
    """
    Compute the cost function given a set of features / values,
    and the values for our thetas.

    This can be the same code as the compute_cost function in the lesson #3 exercises,
    but feel free to implement your own.
    """

        m = len(values)
        sum_of_square_errors = np.square(np.dot(features, theta) - values).sum()
        cost = sum_of_square_errors / (2*m)
        return cost

def gradient_descent(features, values, theta, alpha, num_iterations):
    """
    Perform gradient descent given a data set with an arbitrary number of features.

    This can be the same gradient descent code as in the lesson #3 exercises,
    but feel free to implement your own.
    """
        m = len(values)
        cost_history = []


        for i in range(num_iterations):
            predicted_values = np.dot(features, theta)
             theta -= (alpha / m) * np.dot((predicted_values - values), features)
            cost_history.append(compute_cost(features, values, theta))
        return theta, pandas.Series(cost_history)
```

```
def predictions(dataframe):
   '''
   The NYC turnstile data is stored in a pandas dataframe called weather_turnstile.
   Using the information stored in the dataframe, let's predict the ridership of
   the NYC subway using linear regression with gradient descent.

   You can see the information contained in the turnstile weather dataframe here:
   https://www.dropbox.com/s/meyki2wl9xfa7yk/turnstile_data_master_with_weather.csv

   Your prediction should have a R^2 value of 0.40 or better.

   Note: Due to the memory and CPU limitation of our Amazon EC2 instance, we will
   give you a random subet (~15%) of the data contained in
   turnstile_data_master_with_weather.csv

   If you receive a "server has encountered an error" message, that means you are
   hitting the 30-second limit that's placed on running your program. Try using a
   smaller number for num_iterations if that's the case.

   If you are using your own algorithm/models, see if you can optimize your code so
   that it runs faster.
   '''

   dummy_units = pandas.get_dummies(dataframe['UNIT'], prefix='unit')
   features = dataframe[['rain', 'precipi', 'Hour', 'meantempi']].join(dummy_units)
   values = dataframe[['ENTRIESn_hourly']]
   m = len(values)

   features, mu, sigma = normalize_features(features)

   features['ones'] = np.ones(m)
   features_array = np.array(features)
   values_array = np.array(values).flatten()

   #Set values for alpha, number of iterations.
   alpha = 0.1 # please feel free to change this value
   num_iterations = 100 # please feel free to change this value

   #Initialize theta, perform gradient descent
   theta_gradient_descent = np.zeros(len(features.columns))
   theta_gradient_descent, cost_history = gradient_descent(features_array,
                                                           values_array,
```

```
                                    theta_gradient_descent,
                                    alpha,
                                    num_iterations)

    predictions = np.dot(features_array, theta_gradient_descent)
    return predictions
```

```
our R^2 value is: 0.483404930545

Can you beat the 0.4 R^2 value that we achieved with gradient descent?
```