Table of Contents

Section 2. Data Science – NYC Subway data

2.1 Learning outcomes

The following topics were studied and practiced using some practice exercises and applied to the current project

- *Pandas* and *Numpy*, dataframes
- The Mann-Whitney U Test using *scipy*
- Data wrangling
- Parsing XML, JSON formats of data
- Database schemas and web APIs for data wrangling
- Impute missing values in a dataset using linear regression
- Welch T Test in Python
- Non Parametric test
- Shapiro-Wilk Normality Test
- Supervised vs. unsupervised learning
- Prediction with regression
  - Ordinary Linear Regression
  - Linear regression with Gradient Descent
    - Cost function
    - Coefficients of determination
    - Learning rate
- Types of visual encoding
  - Position, Length, Angle, Direction, Shape, Area/Volume
  - Hue and saturation
- ggplot in python

2.2 Project 2: Analyze the New York Subway data set

In this project, we will look at the NYC subway data set and figure out if more people ride the ride the subway when it is raining or when it is not raining. We will also try and apply some MapReduce concepts to the same dataset.

This project can be divided into two parts.

- First part *will concentrate on converting the data into neat, analyzable, workable format using some data wrangling techniques. Then, we will perform Mann-Whitney U-test and Linear regression on the subway data to draw conclusions about ridership.*
- In Second part, *we will get a feel of how MapReduce will be applied for bigdata*

2.2.1 Part 1: Draw conclusions about ridership of subway vs. Rain

Note: Conclude using results from a statistical test (Mann-Whitney U test) and Linear regression

Objective:

We have a NYC subway ridership data with hourly entries into each station recorded along with several other weather parameters. Our goal is to analyze this data, conduct some statistical tests and draw some conclusions about the ridership.

Data used:

"nyc_subway_weather.csv"

Tools used:

ipython-qtconsole

Language:

python

Analysis:

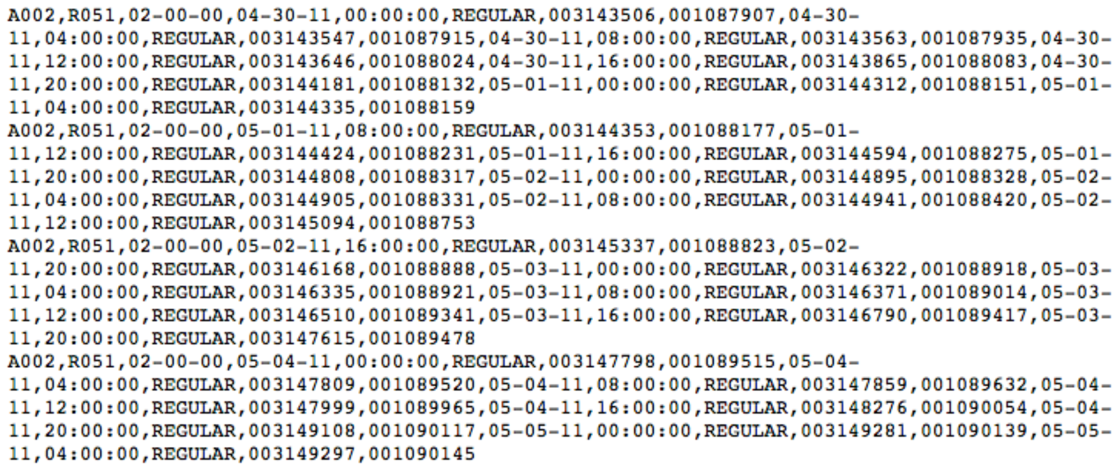
STEP 1: DATA WRANGLING

We need to transform the given data into something that has more workable, compatible data. We will define a set of functions to change and manipulate certain columns of the data. The order of manipulation, transformation and formatting can be as follows:

(A) Raw data to formatted data

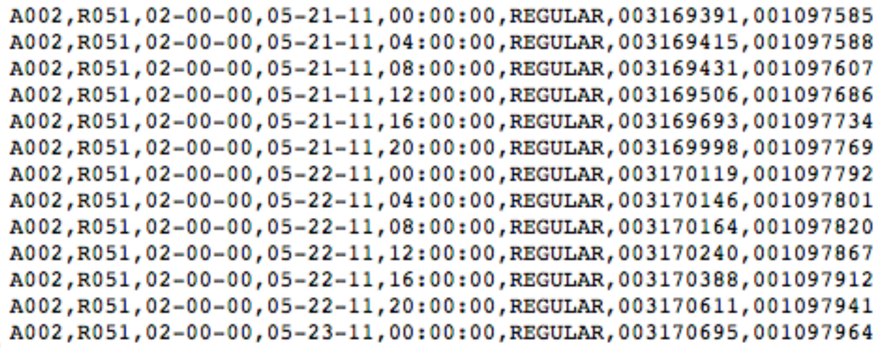
The raw subway data is available at this link:
http://web.mta.info/developers/data/nyct/turnstile/turnstile_110507.txt

A glimpse of how this data looks:

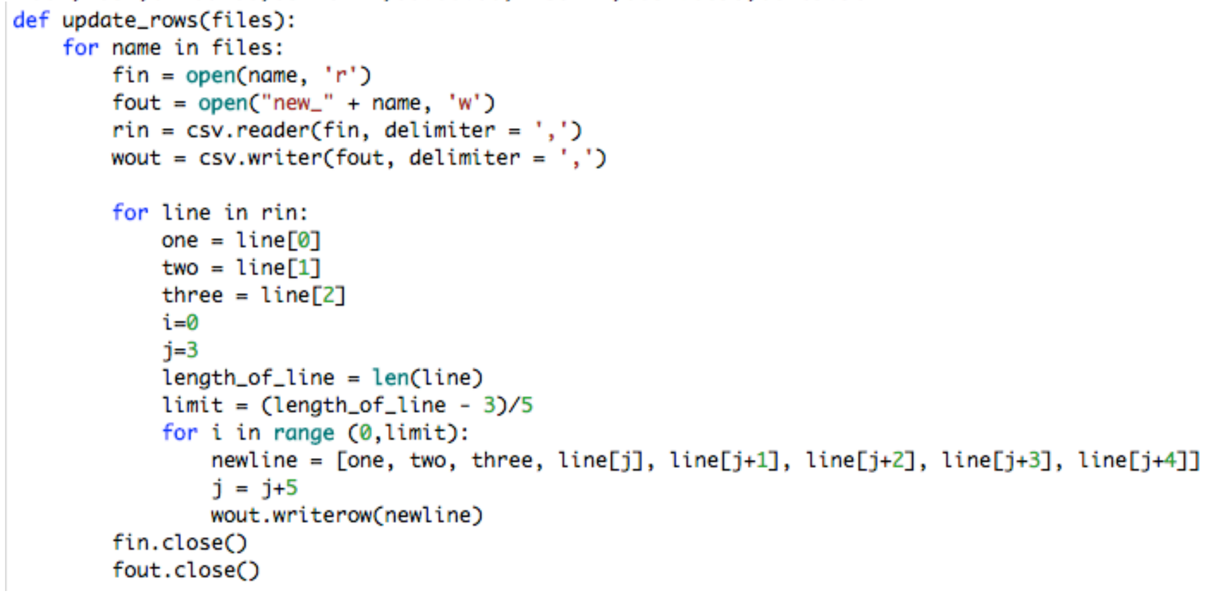


```
A002,R051,02-00-00,04-30-11,00:00:00,REGULAR,003143506,001087907,04-30-
11,04:00:00,REGULAR,003143547,001087915,04-30-11,08:00:00,REGULAR,003143563,001087935,04-30-
11,12:00:00,REGULAR,003143646,001088024,04-30-11,16:00:00,REGULAR,003143865,001088083,04-30-
11,20:00:00,REGULAR,003144181,001088132,05-01-11,00:00:00,REGULAR,003144312,001088151,05-01-
11,04:00:00,REGULAR,003144335,001088159
A002,R051,02-00-00,05-01-11,08:00:00,REGULAR,003144353,001088177,05-01-
11,12:00:00,REGULAR,003144424,001088231,05-01-11,16:00:00,REGULAR,003144594,001088275,05-01-
11,20:00:00,REGULAR,003144808,001088317,05-02-11,00:00:00,REGULAR,003144895,001088328,05-02-
11,04:00:00,REGULAR,003144905,001088331,05-02-11,08:00:00,REGULAR,003144941,001088420,05-02-
11,12:00:00,REGULAR,003145094,001088753
A002,R051,02-00-00,05-02-11,16:00:00,REGULAR,003145337,001088823,05-02-
11,20:00:00,REGULAR,003146168,001088888,05-03-11,00:00:00,REGULAR,003146322,001088918,05-03-
11,04:00:00,REGULAR,003146335,001088921,05-03-11,08:00:00,REGULAR,003146371,001089014,05-03-
11,12:00:00,REGULAR,003146510,001089341,05-03-11,16:00:00,REGULAR,003146790,001089417,05-03-
11,20:00:00,REGULAR,003147615,001089478
A002,R051,02-00-00,05-04-11,00:00:00,REGULAR,003147798,001089515,05-04-
11,04:00:00,REGULAR,003147809,001089520,05-04-11,08:00:00,REGULAR,003147859,001089632,05-04-
11,12:00:00,REGULAR,003147999,001089965,05-04-11,16:00:00,REGULAR,003148276,001090054,05-04-
11,20:00:00,REGULAR,003149108,001090117,05-05-11,00:00:00,REGULAR,003149281,001090139,05-05-
11,04:00:00,REGULAR,003149297,001090145
```

As we can see, each row has numerous data points, which should ideally be separated as several rows. So we need to write a function that will one subway text file at a time and update each row of the file such that there is only one entry per row and write the updates into a new file. Something like this:



```
A002,R051,02-00-00,05-21-11,00:00:00,REGULAR,003169391,001097585
A002,R051,02-00-00,05-21-11,04:00:00,REGULAR,003169415,001097588
A002,R051,02-00-00,05-21-11,08:00:00,REGULAR,003169431,001097607
A002,R051,02-00-00,05-21-11,12:00:00,REGULAR,003169506,001097686
A002,R051,02-00-00,05-21-11,16:00:00,REGULAR,003169693,001097734
A002,R051,02-00-00,05-21-11,20:00:00,REGULAR,003169998,001097769
A002,R051,02-00-00,05-22-11,00:00:00,REGULAR,003170119,001097792
A002,R051,02-00-00,05-22-11,04:00:00,REGULAR,003170146,001097801
A002,R051,02-00-00,05-22-11,08:00:00,REGULAR,003170164,001097820
A002,R051,02-00-00,05-22-11,12:00:00,REGULAR,003170240,001097867
A002,R051,02-00-00,05-22-11,16:00:00,REGULAR,003170388,001097912
A002,R051,02-00-00,05-22-11,20:00:00,REGULAR,003170611,001097941
A002,R051,02-00-00,05-23-11,00:00:00,REGULAR,003170695,001097964
```

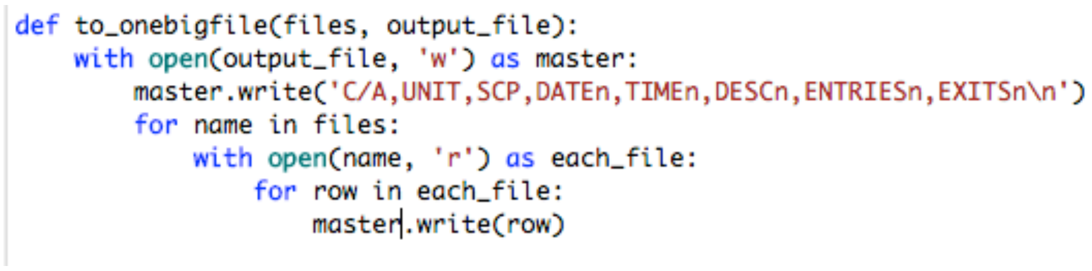


```python
def update_rows(files):
    for name in files:
        fin = open(name, 'r')
        fout = open("new_" + name, 'w')
        rin = csv.reader(fin, delimiter = ',')
        wout = csv.writer(fout, delimiter = ',')

        for line in rin:
            one = line[0]
            two = line[1]
            three = line[2]
            i=0
            j=3
            length_of_line = len(line)
            limit = (length_of_line - 3)/5
            for i in range (0,limit):
                newline = [one, two, three, line[j], line[j+1], line[j+2], line[j+3], line[j+4]]
                j = j+5
                wout.writerow(newline)
        fin.close()
        fout.close()
```

(B) Combine into one big file and add header

The headers for the columns in all data files is known from the MTA NYC subway turnstile website as : 'C/A, UNIT, SCP, DATEn, TIMEn, DESCn, ENTRIESn, EXITSn'

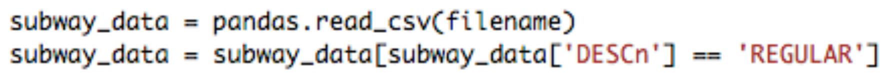
We will now combine all the files into one big file using a function and then add this header at the top, using the following function.



```python
def to_onebigfile(files, output_file):
    with open(output_file, 'w') as master:
        master.write('C/A,UNIT,SCP,DATEn,TIMEn,DESCn,ENTRIESn,EXITSn\n')
        for name in files:
            with open(name, 'r') as each_file:
                for row in each_file:
                    master.write(row)
```

(C) Filter for Regular entry

Now looking at the DESCn column, we have a few undesirable entry points, like 'Door' and 'Open' which correspond to staff. So lets filter our data for only "Regular" entries.



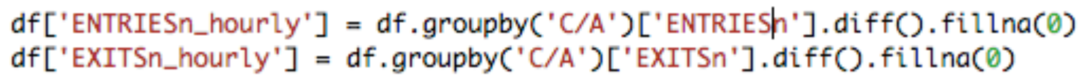
```python
subway_data = pandas.read_csv(filename)
subway_data = subway_data[subway_data['DESCn'] == 'REGULAR']
```

(D) Create new column for hourly entries and exits through subway

The MTA subway data has cumulative number of entries and exits per row. i.e., the entries in current row includes the entries from the previous row. Therefore, we will create a new column to count the entries since last row.

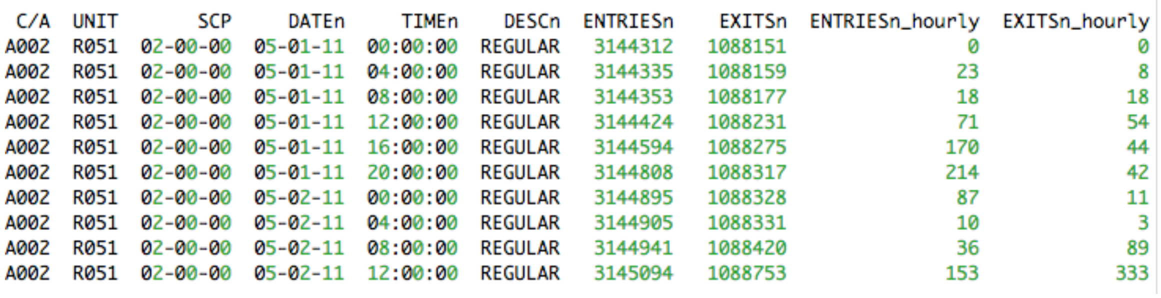
We will use the previously created "subway_data" dataframe to do the changes. We will also make use of pandas library for this operation.

We are grouping the Entries column by the column 'C/A' which indicates the station ID. The following line of code will calculate the entries since last reading.



```python
df['ENTRIESn_hourly'] = df.groupby('C/A')['ENTRIESn'].diff().fillna(0)
df['EXITSn_hourly'] = df.groupby('C/A')['EXITSn'].diff().fillna(0)
```

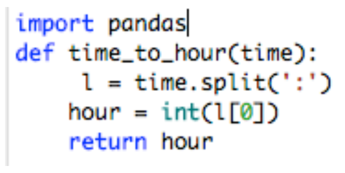
After which the data frame should like this:



```
 C/A  UNIT       SCP    DATEn     TIMEn    DESCn  ENTRIESn   EXITSn  ENTRIESn_hourly  EXITSn_hourly
A002  R051  02-00-00  05-01-11  00:00:00  REGULAR   3144312  1088151                0              0
A002  R051  02-00-00  05-01-11  04:00:00  REGULAR   3144335  1088159               23              8
A002  R051  02-00-00  05-01-11  08:00:00  REGULAR   3144353  1088177               18             18
A002  R051  02-00-00  05-01-11  12:00:00  REGULAR   3144424  1088231               71             54
A002  R051  02-00-00  05-01-11  16:00:00  REGULAR   3144594  1088275              170             44
A002  R051  02-00-00  05-01-11  20:00:00  REGULAR   3144808  1088317              214             42
A002  R051  02-00-00  05-02-11  00:00:00  REGULAR   3144895  1088328               87             11
A002  R051  02-00-00  05-02-11  04:00:00  REGULAR   3144905  1088331               10              3
A002  R051  02-00-00  05-02-11  08:00:00  REGULAR   3144941  1088420               36             89
A002  R051  02-00-00  05-02-11  12:00:00  REGULAR   3145094  1088753              153            333
```

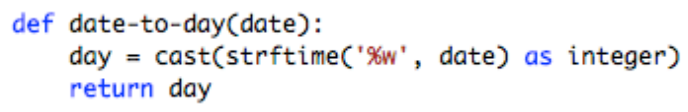
(E) Changing time into Hour of the day and Date into day of the week

We will update a new column called 'Hour' which converts the column 'TIMEn' of format hour:minutes:seconds to just hour.



```
import pandas
def time_to_hour(time):
     l = time.split(':')
    hour = int(l[0])
    return hour
```

We will then look at the date, which is of format 'mm-dd-yy' in our dataframe and convert it to week of the day



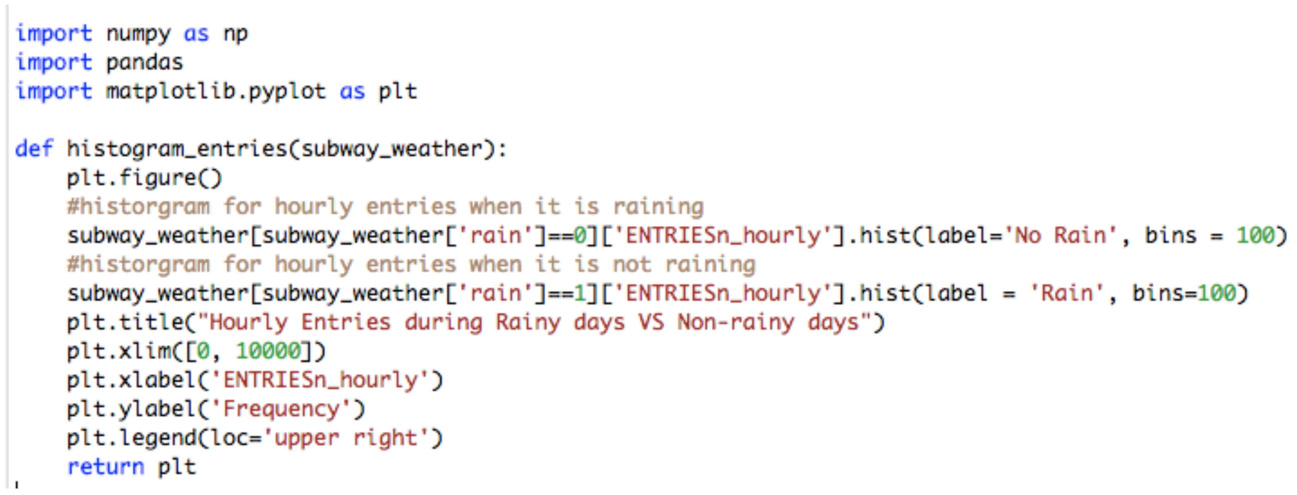
```
def date-to-day(date):
    day = cast(strftime('%w', date) as integer)
    return day
```

Although, we can do a lot more manipulations, we will stop at that as we have enough good format of columns to work with.

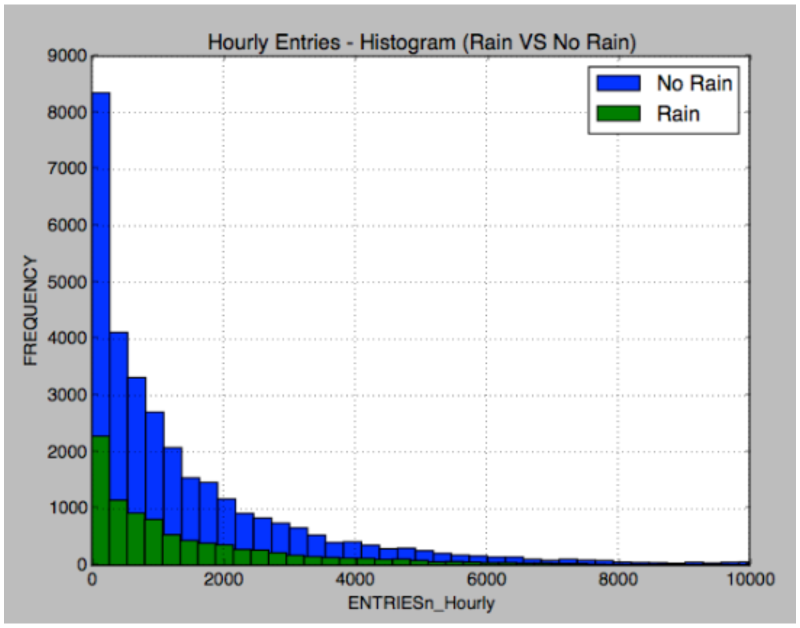
STEP 2: ANALYSIS, STATISTICAL TESTS AND PLOTTING

(A) Pre-Analysis

Before we perform any actual statistical tests, it will be useful to look at the data that we want to analyze. We can plot the histograms of when it is raining and when it is not raining to show the ENTRIESn_hourly data.



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

def histogram_entries(subway_weather):
    plt.figure()
    #historgram for hourly entries when it is raining
    subway_weather[subway_weather['rain']==0]['ENTRIESn_hourly'].hist(label='No Rain', bins = 100)
    #historgram for hourly entries when it is not raining
    subway_weather[subway_weather['rain']==1]['ENTRIESn_hourly'].hist(label = 'Rain', bins=100)
    plt.title("Hourly Entries during Rainy days VS Non-rainy days")
    plt.xlim([0, 10000])
    plt.xlabel('ENTRIESn_hourly')
    plt.ylabel('Frequency')
    plt.legend(loc='upper right')
    return plt
```



This histogram shows that both the distributions are highly skewed and not normally distributed. Also that, data collected for non-rainy days is far more than rainy days

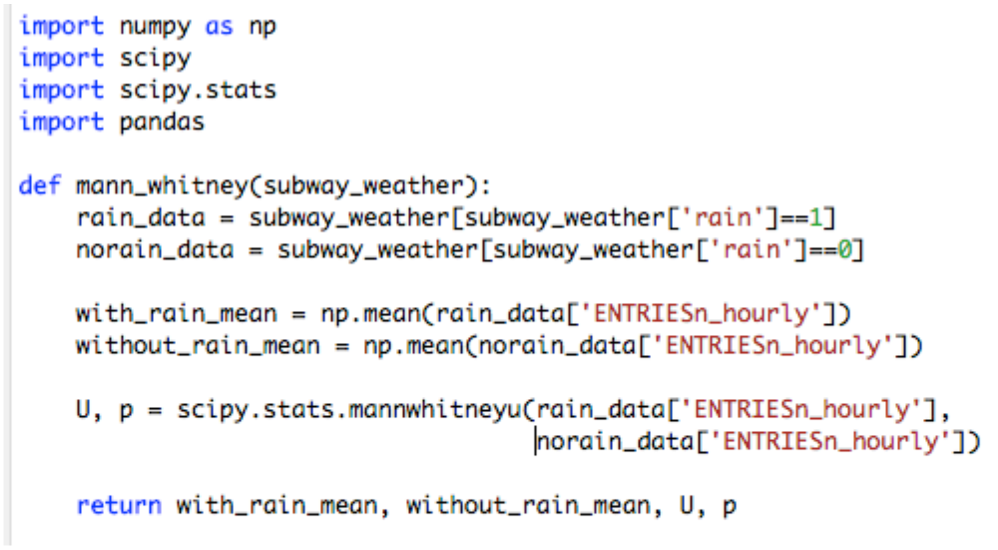
(B) Choosing the statistical test

Well, we can perform Welch's T-test like in the previous project, but this data isn't normally distributed. Welch's t-test assume that its samples come from a normal distribution. Hence, we will choose Mann-Whitney U test, which is a non-parametric test that does not make any assumptions about the probability distribution of its populations.

Lets pass our transformed dataframe to a function that returns the results from Mann-whitney U test. Within this function, we can use python's scipy library's mann-whitney implementation and numpy's mean function.
This function will 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

scipy.stats.mannwhitneyu function returns the U-statistic value and the p-value which denote the statistical significance.


```python
import numpy as np
import scipy
import scipy.stats
import pandas

def mann_whitney(subway_weather):
    rain_data = subway_weather[subway_weather['rain']==1]
    norain_data = subway_weather[subway_weather['rain']==0]

    with_rain_mean = np.mean(rain_data['ENTRIESn_hourly'])
    without_rain_mean = np.mean(norain_data['ENTRIESn_hourly'])

    U, p = scipy.stats.mannwhitneyu(rain_data['ENTRIESn_hourly'],
                                    norain_data['ENTRIESn_hourly'])

    return with_rain_mean, without_rain_mean, U, p
```

The mean values of both samples, the U-statistic and the p-value from the statistical test are as follows:

```
with_rain_mean, without_rain_mean, U, p = (1105.4463767458733, 1090.278780151855, 1
924409167.0, 0.023999912793489721)
```

(C) Establish Hypothesis and critical values

We have to again formalize the null and the alternate hypothesis about the relation between the hourly-entries into subway and the rain, in order to go towards a focused direction of proving whether there is a relation between these two or not.

Normally, the formulation of the hypothesis when using the Mann-Whitney U test is two-tailed, but we obtain a one-sided p-value from the area captured below U i.e. the p-vale returned by the Mann-Whitney U-Test is one-tailed. So, under the standard, two-sided formulation of the null hypothesis, we need to double this probability.
We considered two samples, one with Hourly entries on rainy days and the other with hourly entries on non-rainy days.

The null hypothesis can be defined as follows: *On any given day, if we look at the hourly entries data, it is likely that the day being rainy or non-rainy will not have any kind of effect on the entries. i.e, the ridership on a rainy day is same as the ridership on any non-rainy day. One is not more likely than the other*
H_0 : P(non-rainy > rainy) = 0.5
H_A: P(non-rainy > rainy) $\neq$0.5 (i.e one is more likely than the other)

With 95% confidence interval, our *p critical is 0.05* for a two tailed Mann-Whitney U-Test according to the formulation of the hypothesis.

(D) Significance and interpretation of results from Mann-Whitney U-test

From Part (B) above, we can see that the U-statistic and p-value returned from scipy's mann-whitney U-test are: U = 1 924409167.0, p = 0.023999912793489721

P value here is a one tailed value, so we need to double it to be able to reject or accept our two-tailed null hypothesis. i.e p(two-tailed) = 0.046

Our p critical is 0.05, thus p < p critical since 0.046 < 0.05., although by a very slight margin

Therefore we can be inclined to reject the null hypothesis and say that the ridership on a rainy day will be potentially different from that on a non-rainy day. But which day has a higher ridership???? To know this, we should perform linear regression on the hourly-entries data.

STEP 3: LINEAR REGRESSION

In this step, we will perform linear regression on the data and find the coefficients and the r^2 values, we will then use this to support our previous Mann-whitney results to draw a solid conclusion

We will perform linear regression in two ways
(A) By Ordinary least squares method (OLS)
(B) By Gradient descent method (GD)

We will build a linear model of the form Y =theta0 + theta1*X1 + theta2*X2.... , find the *coefficients of the model* and the *predictions for ENTRIESn_hourly.*

We will compare the results of the above two methods of linear regression with each other and then use the r^2 values and coefficient values of one of these along with previous statistical results to draw a more transparent conclusion.

(A) Linear regression by OLS:

We will use python's "statsmodels.api" library to perform linear regression by ordinary least squares.

(i) First we will select arbitrary set of features (Here I select only 'UNIT', 'rain', 'precipi', 'Hour', 'meantempi')
(ii) We will create dummy units for some categorical variables
(iii) perform linear regression on features and values selected using the statsmodels' OLS function.
(iv) predict on data using the model
(v) compute r^2

```
import numpy as np
import pandas
import statsmodels.api as sm
def linear_regression(features, values):
    features = sm.add_constant(features)
    model = sm.OLS(values, features)
    results = model.fit()
    intercept = results.params[0]
    params = results.params[1:]
    return intercept, params

features = subway_data[['rain', 'precipi', 'Hour', 'meantempi']]
dummy_units = pandas.get_dummies(subway_data['UNIT'])
features = features.join(dummy_units)
values = subway_data['ENTRIESn_hourly']
# linear regression
intercept, params = linear_regression(features, values)

print params

predictions = intercept + np.dot(features, params)
```

```
rain            29.464529
precipi         28.726380
Hour            65.334565
meantempi      -10.531825
unit_R001     4078.938117
unit_R002     -928.711069
unit_R003    -1275.537542
unit_R004    -1085.047542
unit_R005     -436.619905
```

From above we have our predictions for y. But, just because we're able to come up with some model doesn't mean that it's a good one. The data could be distinctly non-linear. Or, maybe the attributes that we've trained our model on have little to no bearing on our output variable. We need some way to evaluate the effectiveness of our model. One way we can measure this is a quantity called the coefficient of determination, also referred to as R squared. Lets compute R^2, which is equal to
1 - {Sum of (predictions – actual y)^2/sum of (actual y – mean of y)^2}

```
numer = np.sum((values-predictions)**2)
denom = np.sum((values-np.mean(values))**2)
r_squared = 1 - (numer/denom)
print r_squared
```

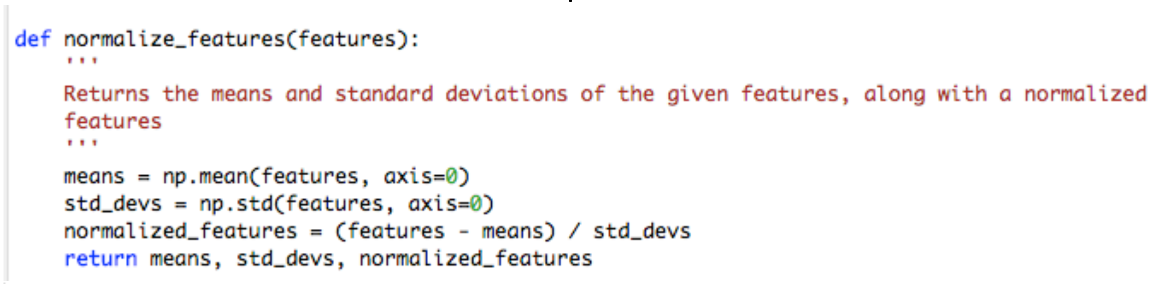
```
0.47924770782
```

<u>(B) Linear Regression by Gradient descent:</u>

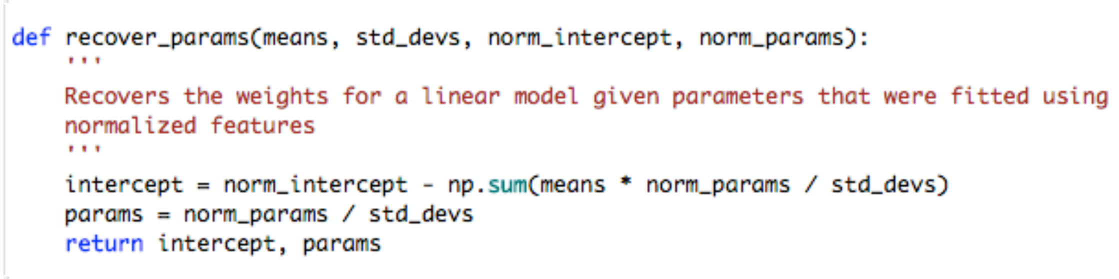
we will use sklearn.linear_model library in python which has a module called SGDRegressor (short for stochastic gradient descent)

For gradient descent procedure, we will have to
(i) first, normalize the feature space
(ii) define or pick features that give the best r^2 (experiment on a trial and error basis)
(iii) fit a linear regression model with gradient descent
(iv) denormalize the parameters obtained from linear regression
(iv) make predictions on data using the built model
(v) compute r^2.

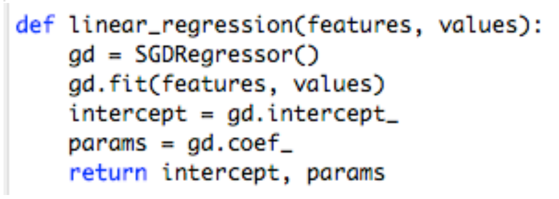
The following code will perform the above steps in order and obtain the coefficients of model and the r^2 value which we will compare with the OLS method.



```
def normalize_features(features):
    '''
    Returns the means and standard deviations of the given features, along with a normalized
    features
    '''
    means = np.mean(features, axis=0)
    std_devs = np.std(features, axis=0)
    normalized_features = (features - means) / std_devs
    return means, std_devs, normalized_features
```



```
def recover_params(means, std_devs, norm_intercept, norm_params):
    '''
    Recovers the weights for a linear model given parameters that were fitted using
    normalized features
    '''
    intercept = norm_intercept - np.sum(means * norm_params / std_devs)
    params = norm_params / std_devs
    return intercept, params
```



```
def linear_regression(features, values):
    gd = SGDRegressor()
    gd.fit(features, values)
    intercept = gd.intercept_
    params = gd.coef_
    return intercept, params
```

```
#Picking features set
features = subway_data[['precipi', 'meantempi', 'meanwindspdi',
                       'meandewpti', 'meanpressurei', 'maxtempi']]
#picking categorical variables
dummy_units = pandas.get_dummies(subway_data['UNIT'], prefix='unit')
features = features.join(dummy_units)
dummy_units = pandas.get_dummies(subway_data['rain'], prefix='rain')
features = features.join(dummy_units)
dummy_units = pandas.get_dummies(subway_data['fog'], prefix='fog')
features = features.join(dummy_units)
dummy_units = pandas.get_dummies(subway_data['Hour'], prefix='Hour')
features = features.join(dummy_units)
# Values - actual y values
values = subway_data['ENTRIESn_hourly']

# Put the features and values into an array
features_array = features.values
values_array = values.values

#calling the normalize_features function that we defined earlier
means, std_devs, normalized_features = normalize_features(features_array)

# Perform gradient descent
normalized_intercept, normalized_params = linear_regression(normalized_features_array,
                                                            values_array)

#we will de-normalize the intercept and the parameters obtained above using
#the function we wrote earlier
intercept, params = recover_params(means, std_devs, normalized_intercept, normalized_params)

print params
predictions = intercept + np.dot(features_array, params)
```

```
[ -5.23603365e+01  -7.72792572e+01   3.96751543e+01   1.59794404e+01
  -5.54307131e+02   4.90495865e+01   2.59370926e+03  -2.32190912e+02
  -1.72482995e+03  -1.38132210e+03  -1.00316587e+03  -5.61699532e+02
  -4.46749247e+02  -1.14554828e+03  -6.29537613e+02   3.21142397e+03
```

Again, from the above calculated predictions, lets compute the r^2 value

```
numer = np.sum((values-predictions)**2)
denom = np.sum((values-np.mean(data))**2)
r_squared = 1 - (numer/denom)
```

```
0.443118282864
```

<u>(C) Summary of OLS and GD : Comparison and reflection</u>

Using OLS approach:

For this model, I used the following input variables: ‘rain', 'precipi', 'Hour', 'meantempi' I also used ‘UNIT’ as a dummy variable to improve the R^2 value.

1 At first, when only ‘rain' and 'precipi' features are used, the R^2 value is as low as 0.0003455
2 Adding more meaningful params to the features list (‘rain', 'precipi', 'Hour') yields an r^2 value of 0.02986
3 Adding 'meantempi' to the feature list yields an r^2 of 0.03064
4 But finally adding ‘UNIT’ as a dummy variable (therefore making UNIT as a categorical variable) yields to a r^2 value of 0.47925

Using SGDRegressor (Gradient descent):

For this model, I used the following features:

- Features: *‘Hour’, 'precipi', 'meantempi', 'meanwindspdi', 'meandewpti', 'meanpressurei', 'maxtempi'*
- Dummy Variables: *UNIT, rain, fog*

Since GD can handle all the variables as features, I could have passed all the variables as features as shown below:
'rain', 'fog', 'Hour','precipi', 'meantempi', 'meanwindspdi', 'meandewpti', 'meanpressurei', 'maxtempi','maxdewpti', 'mintempi', 'mindewpti', 'minpressurei'
But the same was not very effective and gave an r^2 value of 0.391

So more playing around with variables gave different values of r^2. However, the actual increase happened when I changed the categorical variables into dummy variables. So by allowing 6 feature variables and 4 dummy variables as shown in the code, the r^2 value increased to 0.443

Goodness of fit (r^2):

The coefficient of determination denotes the proportion of the variance in the dependent variable that is predictable from the independent variable. Larger the value of R2 , the better the fit of our regression model
Here although both are Linear models, I feel OLS is better suited method than Gradient descent as OLS gave a better r^2 value.

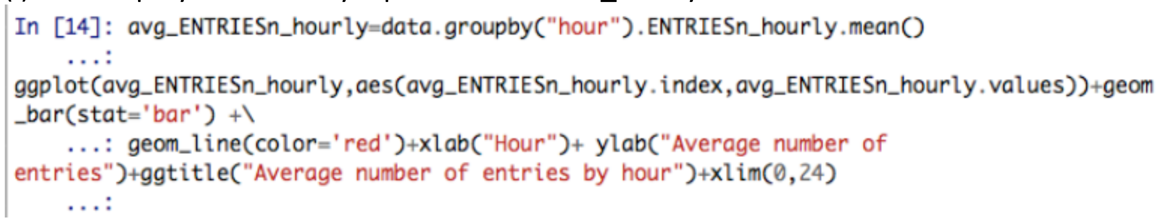
Reflection:

For our Linear model to predict the ridership, I feel, there is a lot of scope for improvement in both OLS and GD methods. Because, according to the R squared value, 48% of the variance in y is accounted for but we still do not know the reason behind remaining 52% of variance. Perhaps more features from the improved dataset with parameters like ‘weekday’, ‘weekend’, ‘day-time’, etc., will make a right impact

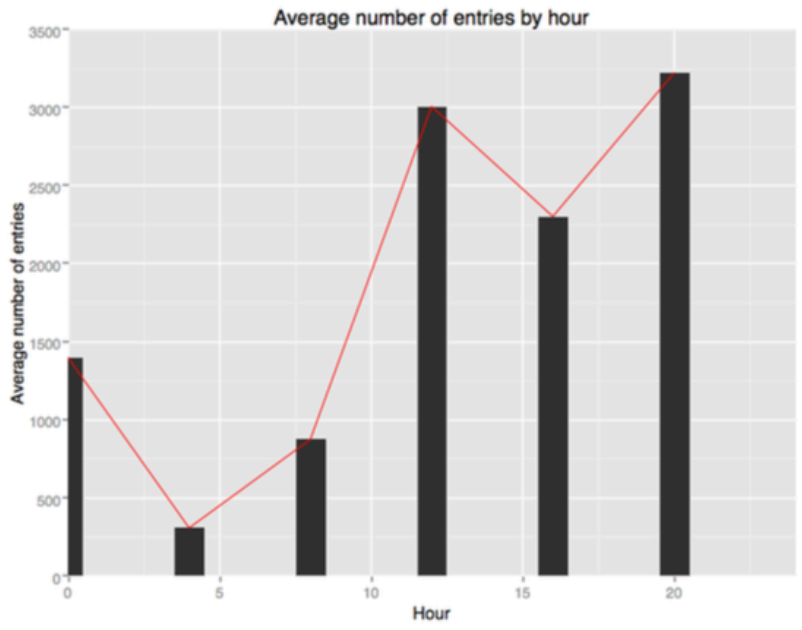
STEP 4: VISUALIZATION

Lets also look at some visualizations from the data before we move on to conclusions about ridership:

(i)Ridership by time of day – plot of ENTRIESn_hourly vs. Hour

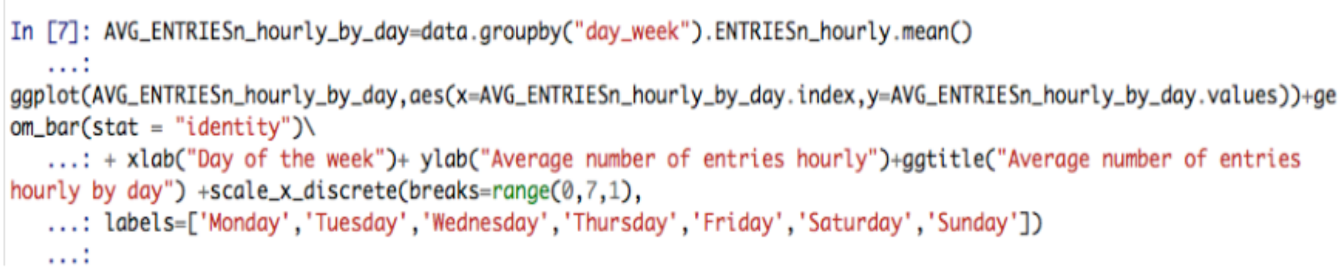


```
In [14]: avg_ENTRIESn_hourly=data.groupby("hour").ENTRIESn_hourly.mean()
    ...:
ggplot(avg_ENTRIESn_hourly,aes(avg_ENTRIESn_hourly.index,avg_ENTRIESn_hourly.values))+geom
_bar(stat='bar') +\
    ...: geom_line(color='red')+xlab("Hour")+ ylab("Average number of
entries")+ggtitle("Average number of entries by hour")+xlim(0,24)
    ...:
```

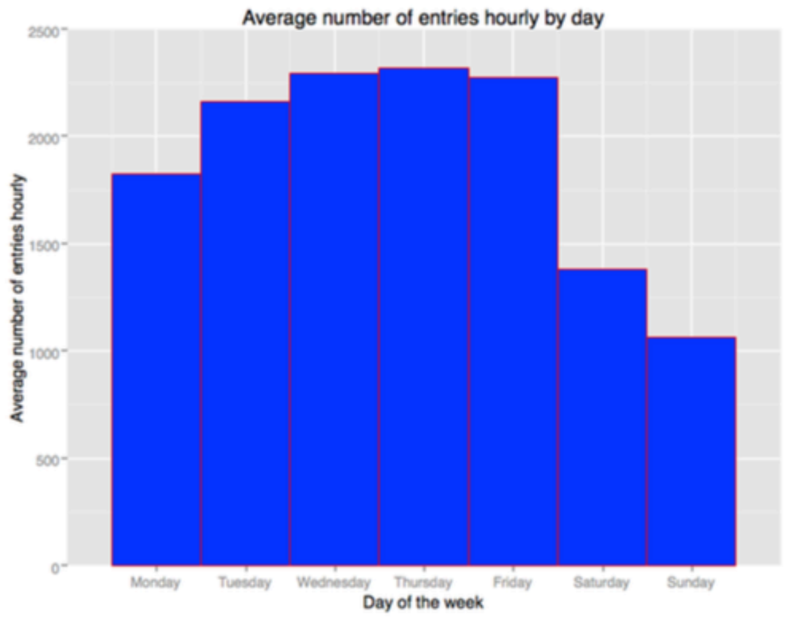


The main inference that can be drawn from above time-of-day plot is that the subway is busy during the hours – 11 to 12 in the morning and at 8 in the evening. I,e the average ridership is at its peak during these hours.

(ii)Ridership by day of the week – plot of ENTRIESn_hourly vs. Day



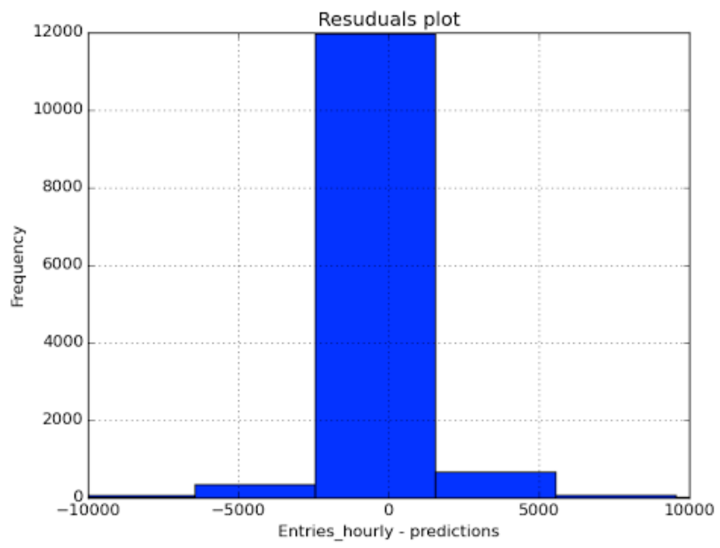
```
In [7]: AVG_ENTRIESn_hourly_by_day=data.groupby("day_week").ENTRIESn_hourly.mean()
   ...:
ggplot(AVG_ENTRIESn_hourly_by_day,aes(x=AVG_ENTRIESn_hourly_by_day.index,y=AVG_ENTRIESn_hourly_by_day.values))+ge
om_bar(stat = "identity")\
   ...: + xlab("Day of the week")+ ylab("Average number of entries hourly")+ggtitle("Average number of entries
hourly by day") +scale_x_discrete(breaks=range(0,7,1),
   ...: labels=['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday'])
   ...:
```



The main inference from the above plot is that the ridership is significantly less on weekends when compared to weekdays

(iii)Residuals plot – Actual ENTRIESn_hourly vs. its predictions from LG(OLS) model (i.e., observed minus predicted)

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

plt.figure()
(subway_data['ENTRIESn_hourly'] - predictions).hist()
plt.title("Resuduals plot")
plt.xlim([-10000,10000])
plt.xlabel('Entries_hourly - predictions')
plt.ylabel('Frequency')
plt.show()
```



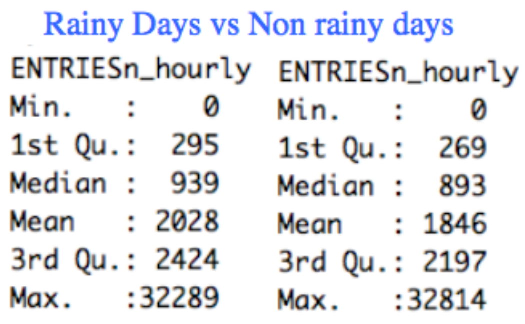
From this we can say that our OLS model performed okay. With most of the residuals lying within -1000 to 1000 range, and some worst cases beyond 1000 till 6000 or so in both directions

STEP 4: FINAL CONCLUSION ABOUT RIDERSHIP VS. RAIN RELATIONSHIP

From results of Mann-Whitney U test Plus Central tendencies of both samples

The results of Mann-Whitney U test only suggest that the ridership is going to be more likely on one type of day than the other type of day. We do not know if rainy days are busy are not. In order to corroborate the mann-whitney test results, we additionally need a median or interquartile range to understand, which day the ridership is more likely. So lets look at the summary of rainy and non-rainy days. need

This summary can be obtained from summary() command on the data in R:


Rainy Days vs Non rainy days

```
ENTRIESn_hourly  ENTRIESn_hourly
Min.   :    0    Min.   :    0
1st Qu.:  295    1st Qu.:  269
Median :  939    Median :  893
Mean   : 2028    Mean   : 1846
3rd Qu.: 2424    3rd Qu.: 2197
Max.   :32289    Max.   :32814
```

The interquartile range for rainy days = 2424 – 295 = 2129
The interquartile range for non-rainy days = 2197 – 269 = 1928

Mean, median and IQR of rainy days – all three > those of non-rainy days
This, along with the results of the statistical test *suggest that the ridership on rainy day is probably more than a non-rainy day.*

Now, From results of Linear regression

Looking at our Linear regression model, our coefficient for rain is positive 29.5, which suggests that there is a positive correlation between rain and ridership. But again, the coefficient of ‘Hour’ (65.33) seems stronger than the ‘rain’ and may be ‘day-week’ also has a stronger influence than ‘rain’. Also, ‘precipi’ has equally strong coefficient indicating positive relation with ridership.

Conclusion from combining both results:

“More people ride the NYC subway when it is raining”

2.2.2 Part 2: Apply MapReduce concepts to the NYC subway data

Concept of MapReduce:

MapReduce is a parallel programming model for processing large data sets across a cluster of computers. When we have to analyze large data, which is too large to sit on one disk (say we want to index all the books in the world), it will be appropriate to use the MapReduce model. MapReduce can employ many computers simultaneously who do not have knowledge of each other’s actions. It breaks large jobs into several smaller chunks, fits each chunk to one machine and perform computations simultaneously.

MapReduce’s computation on a high level is done by two functions – Mapper and Reducer. Mapper and reducer are individually applied to each chunk of document. Our mapper sends all the values of one key to the same reducer. In the end, each reducer will produce one final [key, value] pair

Objective:

The objective of this project is to write mapper and reducer functions for the NYC subway data and draw some interesting facts like how many people passed through subway in the month of may 2011etc.,. Even though subway data is not large enough to apply parallel computing and MapReduce, we can do a simulation of how Mapper and Reducer algorithms are used to perform computations on data.

This will be done in three parts to draw three important inferences from the data.

Data used:

“nyc_subway_weather.csv”

Tools used:

ipython-qtconsole

Language:

python

Application 1: Subway ridership by weather conditions – fog and rain

Here we want to compute the average entries per hour for the two different weather conditions we are provided with in this data set – rain and fog

We will write two functions – one mapper, which will print the weather type as the key and the number in the ENTRIESn_hourly column as the value separated by a tab
 For example: 'fog-norain\t897' and one reducer, which will count the keys of same type and take an average of the corresponding values that belong to same key

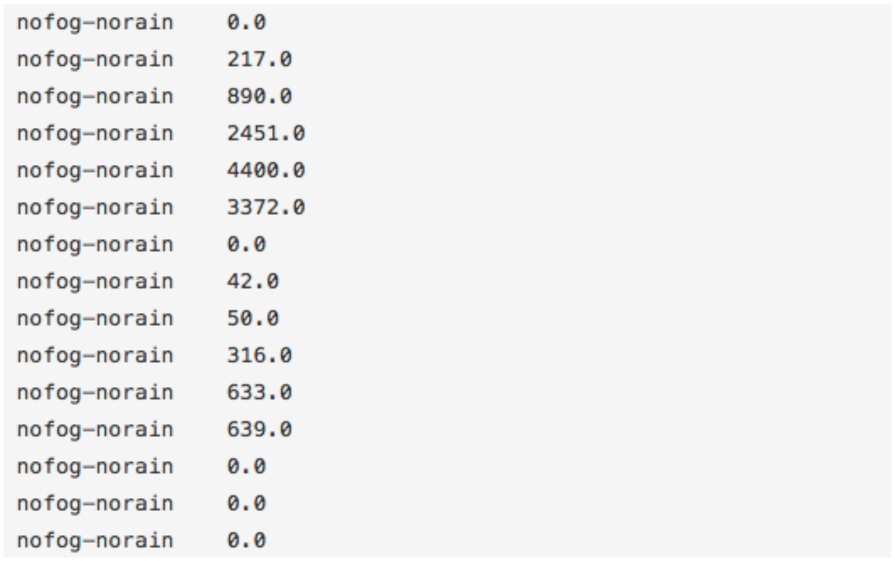
The output of the mapper function can be logged into a file and sent to reducer by writing the following code above each execution of mapper.

```
from util import reducer_logfile
logging.basicConfig(filename=reducer_logfile, format='%(message)s',
                    level=logging.INFO, filemode='w')
```

MAPPER:

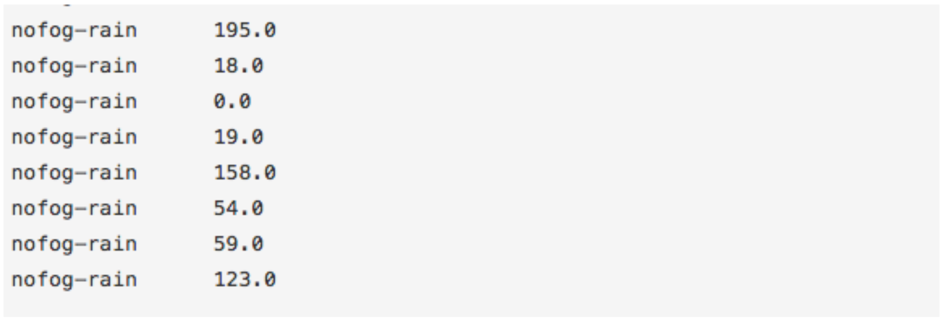
```
import sys
import string
import logging

def mapper():

    # Takes in variables indicating whether it is foggy and/or rainy and
    # returns a formatted key that you should output.
    def format_key(fog, rain):
        return '{}fog-{}rain'.format(
            '' if fog else 'no',
            '' if rain else 'no'
        )

    for line in sys.stdin:
        data = line.strip().split(',');

        if len(data) !=22 or data[6] == "ENTRIESn_hourly":
            continue
        else:
            print "{0}\t{1}".format(format_key(float(data[14]),float(data[15])), data[6])

mapper()
```

This will produce all the 42000+ rows as key-value pairs



```
nofog-norain    0.0
nofog-norain    217.0
nofog-norain    890.0
nofog-norain    2451.0
nofog-norain    4400.0
nofog-norain    3372.0
nofog-norain    0.0
nofog-norain    42.0
nofog-norain    50.0
nofog-norain    316.0
nofog-norain    633.0
nofog-norain    639.0
nofog-norain    0.0
nofog-norain    0.0
nofog-norain    0.0
```

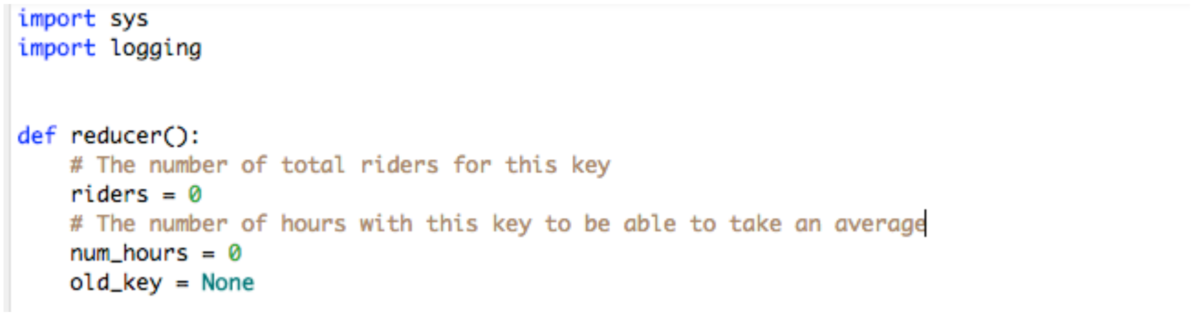
.....
.....
.....



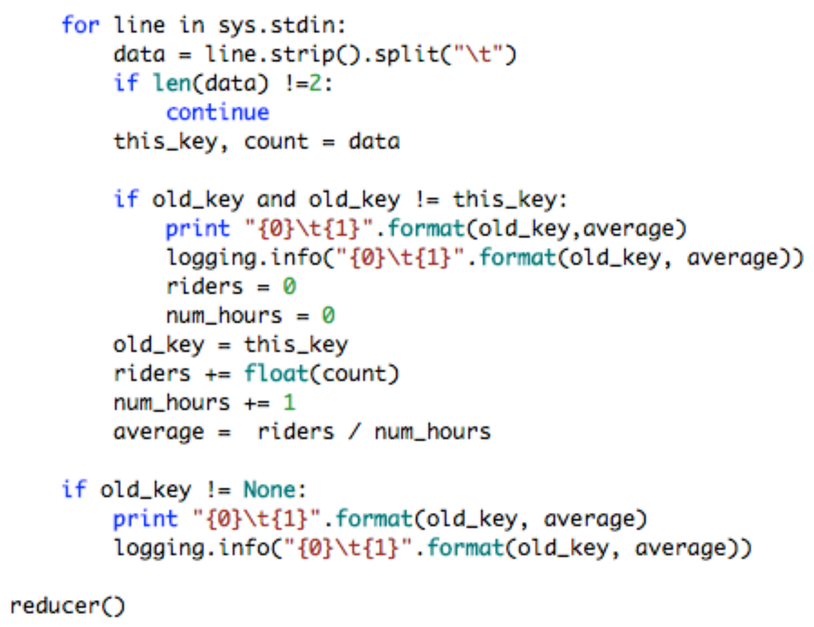
```
nofog-rain      195.0
nofog-rain      18.0
nofog-rain      0.0
nofog-rain      19.0
nofog-rain      158.0
nofog-rain      54.0
nofog-rain      59.0
nofog-rain      123.0
```

Now, the reducer function will take in the output of the mapper as an input. Note: The input to the reducer will be sorted by weather type so that the entries corresponding to a single weather (key value) are together. In reality one such group will go to the same reducer and different groups may go to different reducers.

REDUCER:

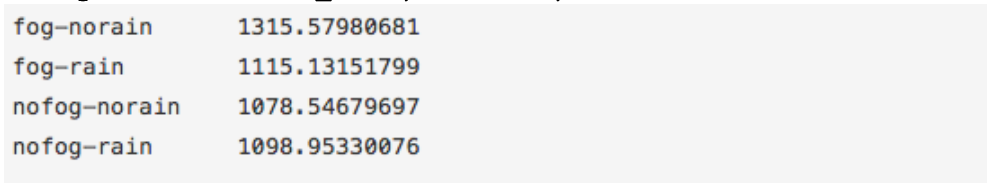


```python
import sys
import logging


def reducer():
    # The number of total riders for this key
    riders = 0
    # The number of hours with this key to be able to take an average
    num_hours = 0
    old_key = None
```



```
    for line in sys.stdin:
        data = line.strip().split("\t")
        if len(data) !=2:
            continue
        this_key, count = data

        if old_key and old_key != this_key:
            print "{0}\t{1}".format(old_key,average)
            logging.info("{0}\t{1}".format(old_key, average))
            riders = 0
            num_hours = 0
        old_key = this_key
        riders += float(count)
        num_hours += 1
        average =  riders / num_hours

    if old_key != None:
        print "{0}\t{1}".format(old_key, average)
        logging.info("{0}\t{1}".format(old_key, average))

reducer()
```

This will produce a key value pair where key is the consolidated key and value is the average value of entries_hourly for this key

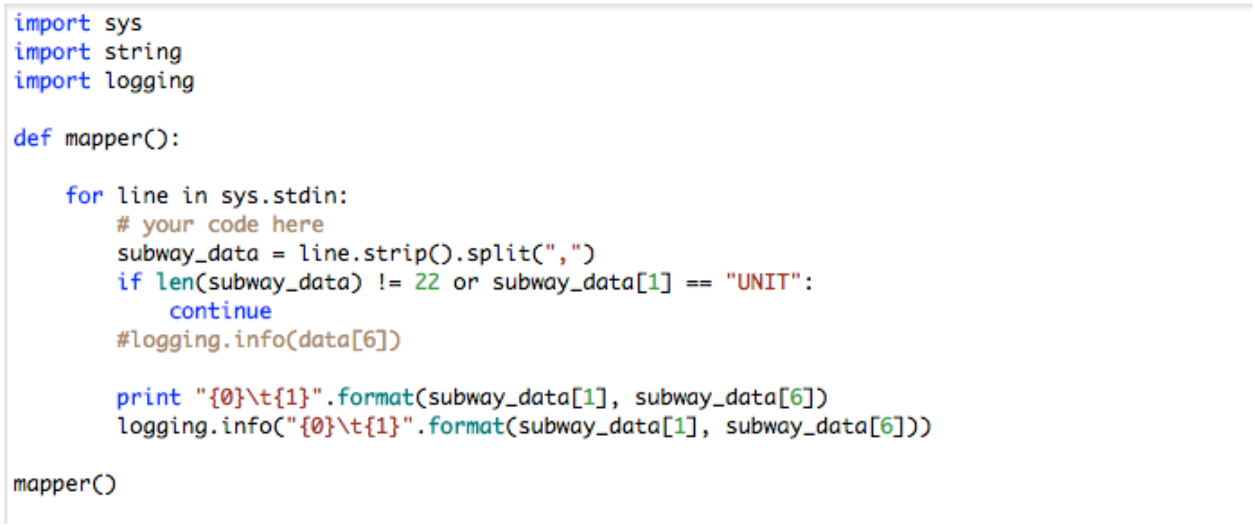


```
fog-norain      1315.57980681
fog-rain        1115.13151799
nofog-norain    1078.54679697
nofog-rain      1098.95330076
```

Inference 2: Subway riders per station

For each line of input, the mapper will print the column “UNIT” which indicates the station code as Key and the number of entries_hourly from the “ENTRIESn_hourly” column as the value, the key-value pair separated by tab

MAPPER:



```
import sys
import string
import logging

def mapper():

    for line in sys.stdin:
        # your code here
        subway_data = line.strip().split(",")
        if len(subway_data) != 22 or subway_data[1] == "UNIT":
            continue
        #logging.info(data[6])

        print "{0}\t{1}".format(subway_data[1], subway_data[6])
        logging.info("{0}\t{1}".format(subway_data[1], subway_data[6]))

mapper()
```

```
R001    0.0
R001    217.0
R001    890.0
R001    2451.0
R001    4400.0
R001    3372.0
R002    0.0
R002    42.0
R002    50.0
R002    316.0
R002    633.0
R002    639.0
R003    0.0
R003    0.0
R003    0.0
```

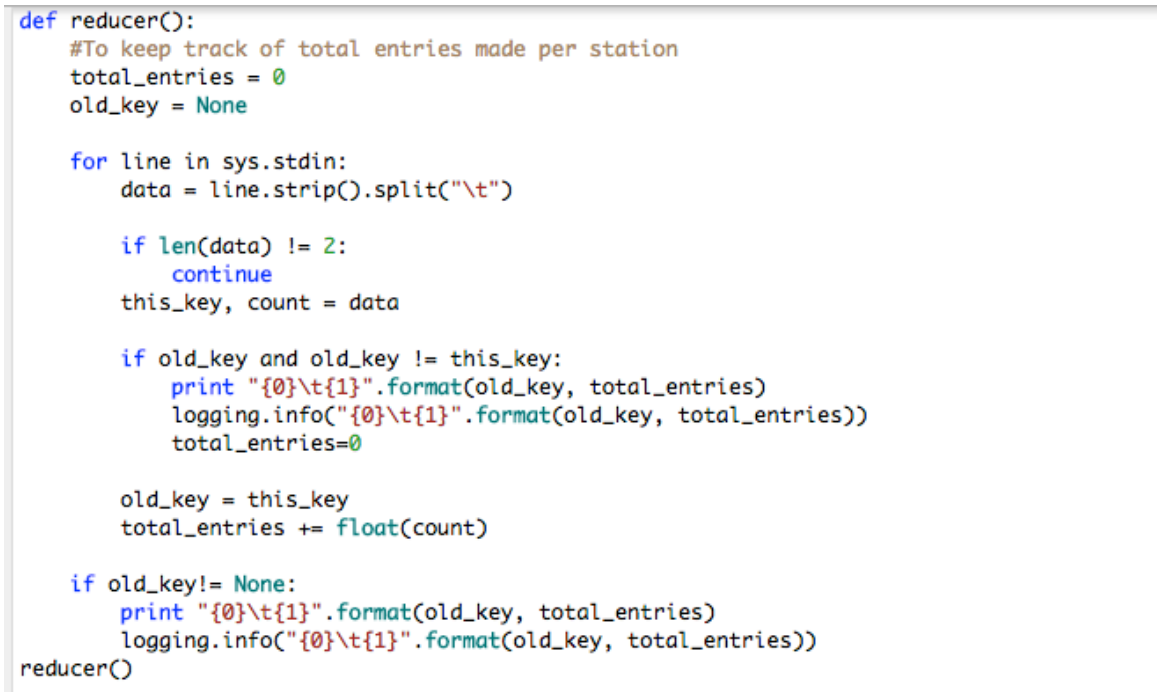
….

….

…all the 40000 rows from the dataset

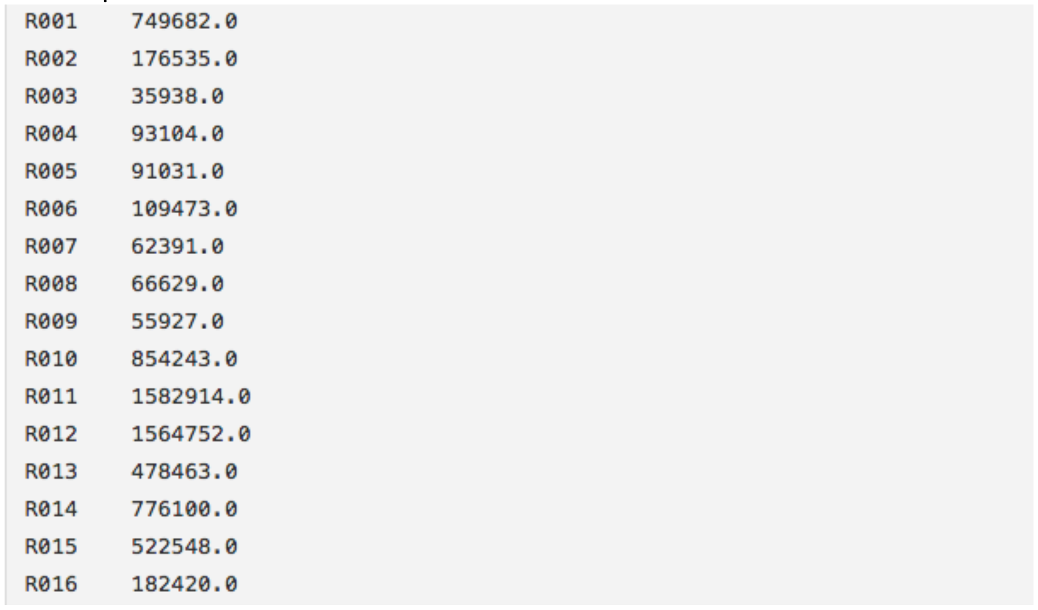
```
R552    68.0
R552    7.0
R552    80.0
R552    195.0
R552    18.0
R552    0.0
R552    19.0
R552    158.0
R552    54.0
R552    59.0
R552    123.0
```

REDUCER:



```
def reducer():
    #To keep track of total entries made per station
    total_entries = 0
    old_key = None

    for line in sys.stdin:
        data = line.strip().split("\t")

        if len(data) != 2:
            continue
        this_key, count = data

        if old_key and old_key != this_key:
            print "{0}\t{1}".format(old_key, total_entries)
            logging.info("{0}\t{1}".format(old_key, total_entries))
            total_entries=0

        old_key = this_key
        total_entries += float(count)

    if old_key!= None:
        print "{0}\t{1}".format(old_key, total_entries)
        logging.info("{0}\t{1}".format(old_key, total_entries))
reducer()
```

The reducer reduces the output of mapper into exact unique key-value pairs.
The output of the reducer is as follows:



```
R001    749682.0
R002    176535.0
R003    35938.0
R004    93104.0
R005    91031.0
R006    109473.0
R007    62391.0
R008    66629.0
R009    55927.0
R010    854243.0
R011    1582914.0
R012    1564752.0
R013    478463.0
R014    776100.0
R015    522548.0
R016    182420.0
```

….
Exactly 552 rows as there are 552 stations
….

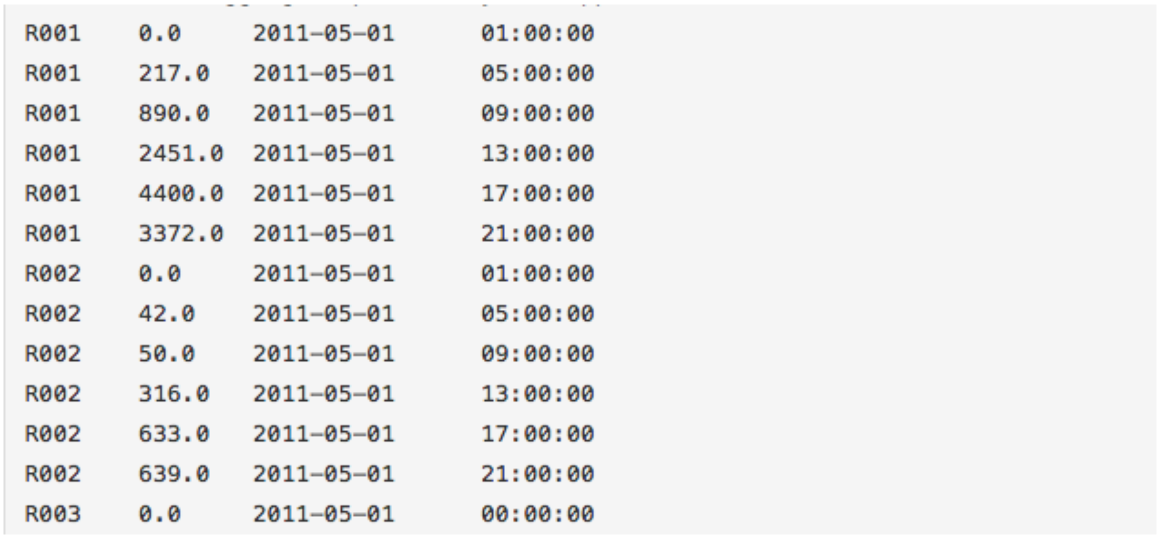
```
R541    887765.0
R542    187258.0
R543    394576.0
R544    164670.0
R545    232567.0
R546    460156.0
R547    127308.0
R548    109426.0
R549    721823.0
R550    668183.0
R551    389045.0
R552    683945.0
```

Inference 3: Busiest Hour of the subway

In this section, we should come up with the date and time at which the most people entered through the unit. For each line, the mapper should return UNIT, ENTRIESn_hourly, DATEn, and TIMEn columns, separated by tabs

MAPPER:

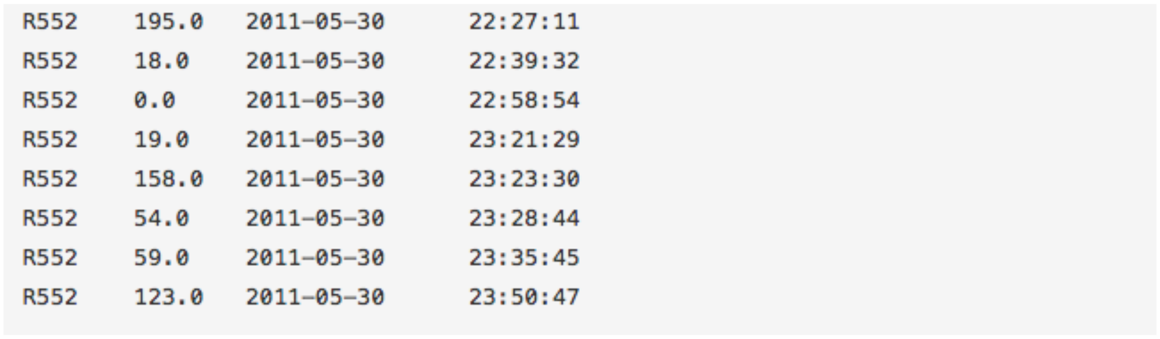
```python
import sys
import string
import logging

def mapper():
    for line in sys.stdin:
        data = line.strip().split(",")
        if len(data) == 22 and data[6] == 'ENTRIESn_hourly':
            continue

        print "{0}\t{1}\t{2}\t{3}".format(data[1],data[6],data[2],data[3])
        logging.info("{0}\t{1}\t{2}\t{3}".format(data[1],data[6],data[2],data[3]))

mapper()
```



```
R001    0.0     2011-05-01      01:00:00
R001    217.0   2011-05-01      05:00:00
R001    890.0   2011-05-01      09:00:00
R001    2451.0  2011-05-01      13:00:00
R001    4400.0  2011-05-01      17:00:00
R001    3372.0  2011-05-01      21:00:00
R002    0.0     2011-05-01      01:00:00
R002    42.0    2011-05-01      05:00:00
R002    50.0    2011-05-01      09:00:00
R002    316.0   2011-05-01      13:00:00
R002    633.0   2011-05-01      17:00:00
R002    639.0   2011-05-01      21:00:00
R003    0.0     2011-05-01      00:00:00
```

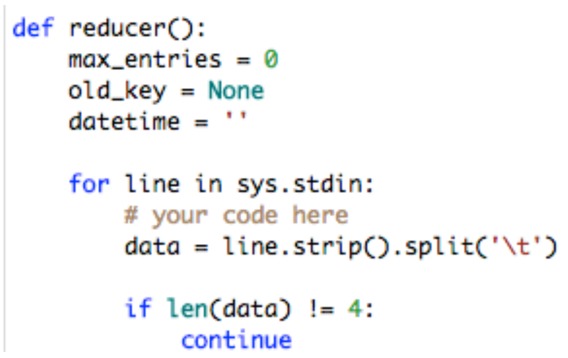
….

40000 + rows



```
R552    195.0   2011-05-30      22:27:11
R552    18.0    2011-05-30      22:39:32
R552    0.0     2011-05-30      22:58:54
R552    19.0    2011-05-30      23:21:29
R552    158.0   2011-05-30      23:23:30
R552    54.0    2011-05-30      23:28:44
R552    59.0    2011-05-30      23:35:45
R552    123.0   2011-05-30      23:50:47
```

REDUCER:



```python
def reducer():
    max_entries = 0
    old_key = None
    datetime = ''

    for line in sys.stdin:
        # your code here
        data = line.strip().split('\t')

        if len(data) != 4:
            continue
        this_key, count, date, time  = data
        count = float(count)

        if old_key and old_key != this_key:
            print "{0}\t{1}\t{2}".format(old_key, datetime, max_entries)
            logging.info("{0}\t{1}\t{2}".format(old_key, datetime, max_entries))
            max_entries = 0
            datetime = ''

        old_key = this_key
        # this will automatically take care of ties and break
        #them in favor of entries coming later in the data (later in may)
        #since the data is already sorted by date and time
        if count >= max_entries:
            max_entries = count
            datetime = str(date) + ' ' + str(time)

    if old_key != None:
        print "{0}\t{1}\t{2}".format(old_key, datetime, max_entries)
        logging.info("{0}\t{1}\t{2}".format(old_key, datetime, max_entries))

reducer()
```

The Reducer produces exactly 552 lines – one for each station
And the date time shown against each station is the busiest hour for that station

```
R001    2011-05-11 17:00:00     31213.0
R002    2011-05-12 21:00:00     4295.0
R003    2011-05-05 12:00:00     995.0
R004    2011-05-12 12:00:00     2318.0
R005    2011-05-10 12:00:00     2705.0
R006    2011-05-25 12:00:00     2784.0
R007    2011-05-10 12:00:00     1763.0
R008    2011-05-12 12:00:00     1724.0
R009    2011-05-05 12:00:00     1230.0
```

….

…..

```
R547    2011-05-25 23:23:07     1247.0
R548    2011-05-06 20:19:16     1664.0
R549    2011-05-03 09:29:37     1237.0
R550    2011-05-11 13:47:15     1637.0
R551    2011-05-16 12:37:58     1280.0
R552    2011-05-20 21:54:55     2917.0
```