Metro ridership during rainy days versus non rainy days

Federico A. Todeschini

April 2014

1 References

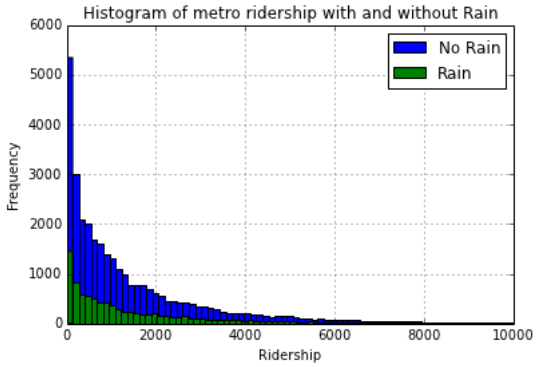
- http://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test
- http://statsmodels.sourceforge.net/stable/
- http://blog.yhathq.com/posts/aggregating-and-plotting-time-series-in-python.html
- https://python.g-node.org/python-summerschool-2009/python_code_in_latex

2 Statistical Test

1.1 In order to answer the question on whether more people used the subway on a rainy day, we performed a Mann-Whitney test. In this case the null hypothesis was that the both the number of entries per hour during rain and during no rain were from the same population and as such the probability of an observation from the rain population exceeding and observation from the non rain population equals the probability of an observation from the non rain population exceeding an observation from the rain population. We used a two sided test, since we had no a priori knowledge on the median of the two variables and as a consequence the difference of the two means could take any direction. Our choice for the critical p value was 0.05.

1.2 The Mann-Whitney test is a nonparametric test that allows two groups or conditions or treatments to be compared without making the assumption that values are normally distributed. In order to see whether the variables were normally distributed, we first draw and histogram of the number of entries per hour both for the cases where it was raining and when it was not raining, which we can see in graphic 1.

Figure 1: Metro ridership with and without rain



According to the aforementioned graphic, the distribution of the variables do not seem to come from a normal distribution. Because of that we run the Shapiro-Wilk test to know what was the likelihood that the number of entries per hour was obtained from a normal distribution. Since the student test is applicable

only certain assumptions (normal distribution) but is more efficient than the Mann-Whitney test, we wanted to make sure which one to use.

Due to the evidence at hand and the fact that we had enough observations, we performed the the non parametric Mann-Whitney test

1.3 In table 1 we have the mean value of the metro ridership during rainy days and non rainy days, and the p-value from the Mann-Whitney test.

Table 1: Mean Ridership during rainy and non rainy days

Mean Rain	Mean No Rain	Mann Whitney p value
2028.20	1845.54	2.7410695712437496e-06

1.4 From table 1 we see that the mean ridership per hour during rainy moments is larger than during non rainy moments. Now, this could be due just by chance, that is, an unfortunate sample. The result from the Mann Whitney test tell us that the probability of observing such extreme difference between the two distribution is very low. As a consequence, we can reject the null hypothesis that the two distributions are the same in favor of the alternative hypothesis.

3 Linear Regression

2.1 The approach used to compute the coefficients theta and produce prediction for ENTRIESn_hourly in my regression model was OLS using Statsmodels.

2.2 Among the input variables used to estimate the model were

- rain
- precipi
- fog
- tempi
- pressurei
- wspdi
- station
- day

- hour

Weekday is a dummy variable that indicates whether it is a weekday or not. The variable hour was transformed into dummy variables for each time of the day and the variable day was transformed into dummy variables indicating the day of the week. Additionally the station is also used as an input and it is transformed into dummy variables. We have also created another variables from precipi. One precipi2, which is precipi squared and another, rain alt, which is a dummy variable being one when precipi is greater than zero.

As such, the estimated model was:

$$Ridership_{sht} = \alpha + rain_{sht} + precipi_{sht} + precipi^2_{st} + tempi_{sht} + pressurei_{sht} + wspdi_{sht} + \mu_s + \rho_h + \rho_t + \epsilon_{sht} \quad (1)$$

The results of this model is that rain **decreases** ridership in 72 persons per hour, a result that is statistically different from zero, which leads us to reject the null hyptohesis that ridership is not affected by rain. In table 2 we show the results for the different specification of the statistical analysis

Table 2: Results from the different OLS estimations

Variable	Baseline model	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
$rain$	182.65***	-398.33***	404.60***	241.96***	-95.4270**	-78.70*	-82.80***
$rain\ alt$		298.54***					
$precipi$			-78.19***	-70.38***	88.09***	78.2412***	22.35
$precipi^2$			1.32	0.88	-3.32***	-3.19***	-1.84**
fog			100.40	162.43	-685.29***	-670.90***	-208.46*
$tempi$			33.93***	32.07	-14.57***	-15.68***	-16.18***
$pressurei$			-123.61	-459.73	-657.96***	-529.39***	-420.48***
$wspdi$			37.74***	25.03	-45.02***	-43.74***	-0.87
$weekday$				917.74***	1058.09***		
Hour dummies	No	No	No	Yes	Yes	Yes	Yes
Day dummies	No	No	No	No	Yes	Yes	Yes
Station dummies	No	No	No	No	No	No	Yes
R^2	0.001	0.002	0.015	0.034	0.162		0.491

*,** and *** signifcant at 10%, 5% and 1% respectively

2.3 Weekday is included because it is a variable that is correlated with metro ridership, as more people uses the metro during labor days. As such, if in our sample rain mostly happens during sunday, shall we not include this variable we might conclude that ridership is not affected by rain. For the same reason we included the station and the hour of the day. The variable precipitation as its square was included because it is possible that the relationship between rain and ridership is not linear. The variables fog, tempi, pressurei and wspdi were included on an intuitive basis, in the sense that atmospheric conditions might affect ridership.

2.4 The coefficients for the non dummy variables in our the model 3 estimation, that includes precipi, precipi2, fog, tempi, pressurei and wspdi and weekday are shown in table 2 along with the R^2.

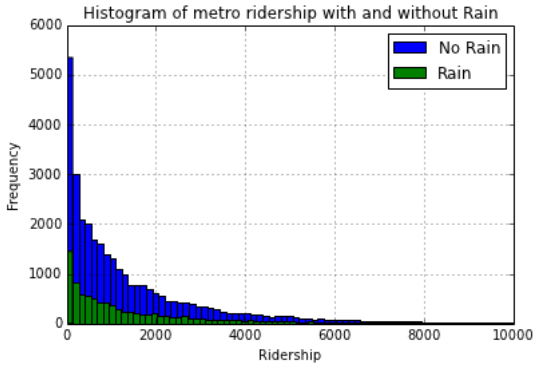
2.5 The estimated model's largest R^2 is 0.491. However, this is a model that includes dummy variables for each station, each day of the week and each hour of the day, so it is probably 'inflated' in the sense that fixed effects usually drive up the R^2 without necessarily improving the prediction ability of the model

2.6 The value obtained means that we are able to explain about 50% of the variance, of 29% of the standard deviation.

4 Visualization

3.1 Histrogram of ridership for rainy and non rainy days In grafic 2 we can see the histogram for the ridership for rain from 0 to 10,000 riders per hour. As we can see, on those days when it rain, ridership increases creating very extreme values with a very low frequency.

Figure 2: Metro ridership with and without rain



3.2 Boxplot of ridership by day of the week In figure 3 we can see how the variabiliy of metro ridership by day of the week using a boxplot graph. This chart, show us both the different quantiles of the distribution plus the outliers.

Figure 4 shows boxplots by weekday and whether it has rained or not. We can see that it is mostly the outliers from the working days that are driving the results in the mean test and in the OLS regression

Finally in figure 5 we can see the total ridership per day during May at the different hours of the day. In this graphic we can see the huge stationality that the data has.

5 Reflection

4.1 According to the analysis done we can conclude that rain affects metro ridership. That is, during the day that some rain is recorded, more people uses the metro.

Figure 3: Metro ridership by day of the week

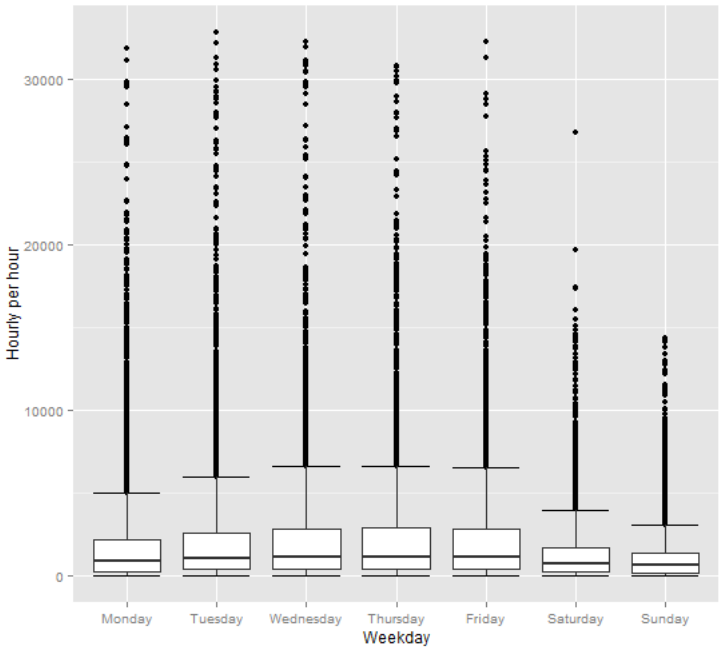


Figure 4: Metro ridership by weekday and rain

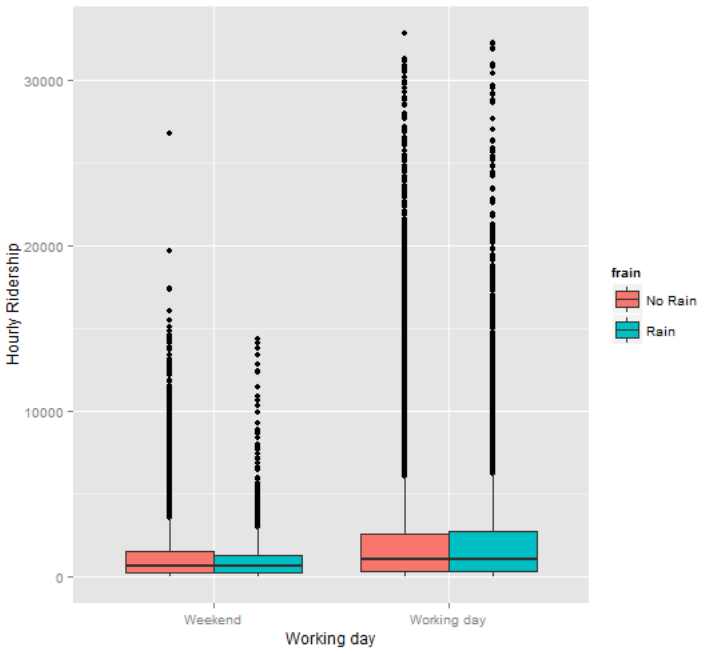
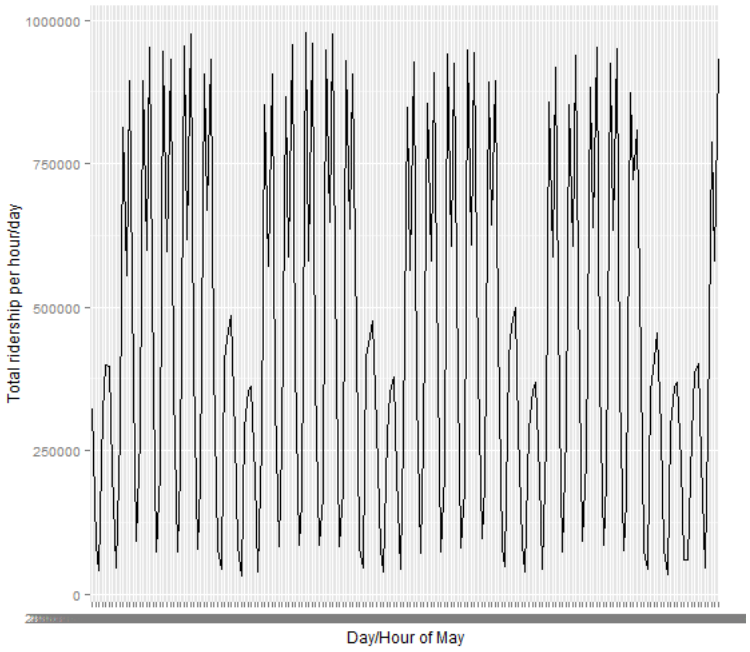


Figure 5: Total daily metro ridership



4.2 The regression analysis, the statistical test and the graphic all show the same: in the day that it rains, ridership is increased. In the graphic, for instance, we can see that rainy days have a much larger values although a very low frequency as the proportion of rainy days in the sample is quite low (22%). This result holds even as we include controls by hour and controls by day, as well as other controls for atmospheric conditions. However, when we include controls at the station the sign of rain changes. One possible explanation for this fact is that at the moment of the rain less people go out and as a consequence it reduces ridership. Also, the fact is that the extreme values are the ones driving the results on the more basic tests and the fixed effects by station and hour are probably capturing a large part of this effect.

6 Reflection

5.1 A very first problem that we have with our dataset is that the variable of interest, metro ridership by hour, is probably not stationary. That means, in a very plane manner, that the variable grows in time and that can affect the relationship between any variable and ridership. In that sense, it would be useful to put in context, for instance, something like the ratio of ridership to the total population. A second shortcoming of the dataset is that during the year there are rainy seasons and there are seasons without rain. That means that rain will behave very differently during different moments of the year and we should try to remove that. The same happens with ridership, although in a worst manner. If we would be taking our information on August and September and it happens to be the case that september is the rainy moment of the year while August is the

holiday month, everything else being equal we would wrongly conclude that rain negatively affects ridership. Another interesting thing would be to know whether on that day there was something important going on (some accident, a parade, etc) that would have driven more people into the metro irrespectively. Last but definitely not least, people usually try to forecast what will be the weather during the following day. In that sense, if let's say the forecast is rain then they will be equipped with an umbrella such that rain shouldn't be a problem. Therefore, what will probably affects ridership is unexpected rain. Following on that argument, we would like to incorporate to the dataset the weather forecast from the previous day.

5.1 Since the outliers in this dataset play a major role in driving the results, maybe using linear models it's not the best way to answer the question of whether people ride more the metro during rainy days. One potential way of dealing with this issue would be to model rainy days and non rainy days separately, as if they were two different process. In the case of the Titanic, we did something in that line of reasoning as we tried to separate the different people on the ship. Alternatively, we could try to get rid of the outliers using a more robust model, like quantile regression.

7 Code for the Problem Sets

7.1 PS2

```
import pandas
import pandasql


def num_rainy_days(filename):
    weather_data = pandas.read_csv(filename)

    q = """
    SELECT
    count(date)
    FROM
    weather_data
    WHERE
    cast(rain as integer) = 1
    """

    #Execute your SQL command against the pandas frame
    rainy_days = pandasql.sqldf(q.lower(), locals())
    return rainy_days
```

```
import pandas
import pandasql


def max_temp_aggregate_by_fog(filename):
    weather_data = pandas.read_csv(filename)

    q = """
    SELECT
    fog, max(cast (maxtempi as integer))
    FROM
    weather_data
    GROUP BY
    fog;
    """

    #Execute your SQL command against the pandas frame
    foggy_days = pandasql.sqldf(q.lower(), locals())
    return foggy_days



import pandas
import pandasql

def avg_weekend_temperature(filename):
    weather_data = pandas.read_csv(filename)

    q = """
    SELECT
    avg(meantempi)
    FROM
    weather_data
    WHERE
    cast(strftime('%w', date) as integer) = 0 OR cast(strftime('%w', date) as integer) = 6;
    """

    #Execute your SQL command against the pandas frame
    mean_temp_weekends = pandasql.sqldf(q.lower(), locals())
    return mean_temp\_weekends
```

```
import pandas
import pandasql

def avg_min_temperature(filename):
    weather_data = pandas.read_csv(filename)

    q = """
    SELECT
    avg(mintempi)
    FROM
    weather_data
    WHERE
    rain = 1 and cast(mintempi as integer)>55;
    """

    #Execute your SQL command against the pandas frame
    avg_min_temp_rainy = pandasql.sqldf(q.lower(), locals())
    return avg_min_temp_rainy


import csv

def fix_turnstile_data(filenames):
    for name in filenames:
        f_in = open(name, 'r')
        f_out = open('updated_' + name, 'w')

        reader_in = csv.reader(f_in, delimiter=',')
        writer_out = csv.writer(f_out, delimiter=',')

        #reader_in.next()
        for line in reader_in:
            primero = line[0]
            segundo = line[1]
            tercero = line[2]
            for row in range((len(line)-3)/5):
                linea = [primero, segundo, tercero, line[3+5*row], line[4+5*row], line[5+5*row],
                writer_out.writerow(linea)
        f_in.close()
        f_out.close()
```

```
import csv
def create_master_turnstile_file(filenames, output_file):
    with open(output_file, 'w') as master_file:
        master_file.write('C/A,UNIT,SCP,DATEn,TIMEn,DESCn,ENTRIESn,EXITSn\n')
        for filename in filenames:
            f_in = open(filename, 'r')
            reader_in = csv.reader(f_in, delimiter=',')
            for linea in reader_in:
                a = ','.join(linea)
                a = a+'\n'
                master_file.write(a)
            f_in.close()
```

```
import pandas
import pandasql
import csv

def filter_by_regular(filename):
    turnstile = pandas.read_csv(filename)
    turnstile_data = pandas.DataFrame(turnstile)
    turnstile_data = turnstile_data[turnstile_data.DESCn == 'REGULAR']

    return turnstile_data
```

```
import pandas
import pandasql
import csv

def filter_by_regular(filename):
    turnstile = pandas.read_csv(filename)
    turnstile_data = pandas.DataFrame(turnstile)
    turnstile_data = turnstile_data[turnstile_data.DESCn == 'REGULAR']

    return turnstile_data
```

Ex9 import pandas

```
def get_hourly_exits(df):
    df['EXITSn_hourly'] = df['EXITSn'] - df['EXITSn'].shift()
    df['EXITSn_hourly'].fillna(value = 0, inplace = True)
```

```
    return df


import pandas

def time_to_hour(time):
    hour = int(time[0:2])
    return hour


import pandas

def time_to_hour(time):
    hour = int(time[0:2])
    return hour


import datetime

def reformat_subway_dates(date):
    date_formatted = datetime.datetime.strptime(date, "%m-%d-%y")
    date_formatted = date_formatted.date()
    return date_formatted
```

7.2 PS 3

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

def entries_histogram(turnstile_weather):
    plt.figure()
    turnstile_weather['ENTRIESn_hourly'][turnstile_weather['rain']==0].hist(bins=240, facecolor=
    turnstile_weather['ENTRIESn_hourly'][turnstile_weather['rain']==1].hist(bins=240, facecolor=
    plt.xlabel("ENTRIESn_hourly")
    plt.ylabel("Frequency")
    plt.title("Histogram of ENTRIESn_hourly")
    plt.xlim(0, 6000)
    plt.legend()
    return plt
```

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

def mann_whitney_plus_means(turnstile_weather):
    rain = turnstile_weather['ENTRIESn_hourly'][turnstile_weather['rain']==1]
    norain = turnstile_weather['ENTRIESn_hourly'][turnstile_weather['rain']==0]

    with_rain_mean = np.mean(rain)
    without_rain_mean = np.mean(norain)
    mwt = scipy.stats.mannwhitneyu(rain, norain)
    U = mwt[0]
    p = mwt[1]

    return with_rain_mean, without_rain_mean, U, p


import numpy as np
import pandas
from ggplot import *

def normalize_features(df):
    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 v
                        "in your model.")
    df_normalized = (df - df.mean()) / df.std()

    return df_normalized, mu, sigma

def compute_cost(features, values, theta):
    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):
```

```
    m = len(values)

    cost_history = []
    updated_theta = theta
    i = 0

    ############################
    ### YOUR CODE GOES HERE ###
    ############################
    for i in range(num_iterations):
        predicted_values = np.dot(features, updated_theta)
        updated_theta += alpha/m * np.dot((values - predicted_values), features)
        cost_history.append(compute_cost(features, values, updated_theta))
        i += 1

    theta = list(updated_theta)
    return theta, pandas.Series(cost_history)


def predictions(dataframe):
    # Select Features (try different features!)
    features = dataframe[['rain', 'precipi', 'Hour', 'fog']]

    # 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):
   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 )



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

def plot_residuals(turnstile_weather, predictions):

    plt.figure()
    (turnstile_weather['ENTRIESn_hourly'] - predictions).hist()
    return plt
```

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

def compute_r_squared(data, predictions):
    SSReg = np.sum((data-predictions)**2)
    ## Alternatively
    ## SSReg = (data-predictions)**2).sum()
    SST = np.sum((data-np.mean(data))**2)
    r_squared = 1 - SSReg / SST


    return r_squared


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

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

def predictions(weather_turnstile):
    Y = weather_turnstile['ENTRIESn_hourly']
    X = weather_turnstile[['rain', 'precipi', 'Hour', 'fog', 'thunder']]
    X = sm.add_constant(X)
    dummy_units = pandas.get_dummies(weather_turnstile['UNIT'], prefix='unit')
    X = X.join(dummy_units)
    model = sm.OLS.from_formula('ENTRIESn_hourly ~ 1 + Hour + Hour*rain + fog*rain + rain + I(ra
fog + UNIT', weather_turnstile.reset_index())
    results = model.fit()
    print(model.fit().summary2())
    prediction = results.predict(X)



    return prediction
```

7.3 Problem Set 4

```
from pandas import *
from ggplot import *

def plot_weather_data(turnstile_weather):

    turnstile_df = DataFrame(turnstile_weather)
    turnstile_df['Date'] = to_datetime(turnstile_df['DATEn'])

    plot = ggplot(turnstile_weather, aes(x='Date', y='ENTRIESn_hourly')) + geom_boxplot() +
ggtitle('Ridership by day of the week') + xlab('Day') + ylab('Riders by hour')
    return plot
```

7.4 Problem Set 5

```
import sys
import string
import logging

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

def mapper():
    for line in sys.stdin:
        data = line.strip().split(",")
        if len(data) != 22 or data[1] == 'UNIT':
            continue
        print "{0}\t{1}".format(data[1], data[6])
        #logging.info("{0}\t{1}".format(data[1], data[6]))
        #logging.info(str(data[1])+"\t"+str(data[6]))


mapper()
```

```
import sys
import logging
```

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

def reducer():
    '''
    Given the output of the mapper for this exercise, the reducer should PRINT
    (not return) one line per UNIT along with the total number of ENTRIESn_hourly
    over the course of May (which is the duration of our data), separated by a tab.
    An example output row from the reducer might look like this: 'R001\t500625.0'

    You can assume that the input to the reducer is sorted such that all rows
    corresponding to a particular UNIT are grouped together.

    Since you are printing the output of your program, printing a debug
    statement will interfere with the operation of the grader. Instead,
    use the logging module, which we've configured to log to a file printed
    when you click "Test Run". For example:
    logging.info("My debugging message")
    Note that, unlike print, logging.info will take only a single argument.
    So logging.info("my message") will work, but logging.info("my","message") will not.
    '''

    reg_count = 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, reg_count)
            #logging.info("{0}\t{1}".format(old_key, reg_count))
            #logging.info(str(old_key)+"\t"+str(reg_count))
            reg_count = 0

        old_key = this_key
        reg_count += float(count)

    if old_key != None:
```

```
        print "{0}\t{1}".format(old_key, reg_count)
        #logging.info("{0}\t{1}".format(old_key, reg_count))
        #logging.info(str(old_key)+"\t"+str(reg_count))

reducer()



import sys
import string
import logging

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

def mapper():
    # Takes in variables indicating whether it is foggy and/or rainy and
    # returns a formatted key that you should output.  The variables passed in
    # can be booleans, ints (0 for false and 1 for true) or floats (0.0 for
    # false and 1.0 for true), but the strings '0.0' and '1.0' will not work,
    # so make sure you convert these values to an appropriate type before
    # calling the function.
    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[1] == 'UNIT':
            continue
        #logging.info(data[14])
        #logging.info(data[15])
        #logging.info(format_key(data[14],data[15]))
        print "{0}\t{1}".format(format_key(float(data[14]), float(data[15])), data[6])
        #logging.info("{0}\t{1}".format(format_key(float(data[14]), float(data[15])), data[6]))
mapper()

import sys
import logging
```

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

def reducer():
    riders = 0      # The number of total riders for this key
    num_hours = 0   # The number of hours with this key
    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, riders/num_hours)
            #logging.info("{0}\t{1}".format(old_key, riders/num_hours))
            average_riders = 0
            riders = 0
            num_hours = 0
        old_key = this_key
        riders += float(count)
        num_hours += float(1)


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

reducer()
```

```
import sys
import string
import logging

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

```
def mapper():
    """
    In this exercise, for each turnstile unit, you will determine the date and time
    (in the span of this data set) at which the most people entered through the unit.

    The input to the mapper will be the final Subway-MTA dataset, the same as
    in the previous exercise. You can check out the csv and its structure below:
    https://www.dropbox.com/s/meyki2wl9xfa7yk/turnstile_data_master_with_weather.csv

    For each line, the mapper should return the UNIT, ENTRIESn_hourly, DATEn, and
    TIMEn columns, separated by tabs. For example:
    'R001\t100000.0\t2011-05-01\t01:00:00'

    Since you are printing the output of your program, printing a debug
    statement will interfere with the operation of the grader. Instead,
    use the logging module, which we've configured to log to a file printed
    when you click "Test Run". For example:
    logging.info("My debugging message")
    Note that, unlike print, logging.info will take only a single argument.
    So logging.info("my message") will work, but logging.info("my","message") will not.
    """

    for line in sys.stdin:
        data = line.strip().split(",")
        if len(data) != 22 or data[1] == 'UNIT':
            continue
        #logging.info(type(data[2]))
        print "{0}\t{1}\t{2}\t{3}".format(data[1], data[6], data[2], data[3])
mapper()

import sys
import logging

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

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

```
    for line in sys.stdin:
        data = line.strip().split("\t")
        if len(data) != 4:
            continue
        this_key, count, date, time = data

        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}".format(old_key, reg_count))
            max_entries = 0
            datetime = ''

        old_key = this_key
        if float(count) > max_entries:
            max_entries = float(count)
            datetime = date + ' ' + time
        elif float(count) == max_entries and date[8:10] > datetime[8:10]:
            datetime = date + ' ' + time
        elif float(count) == max_entries and date[8:10] == datetime[8:10] and time[:2] > datetim

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

reducer()
```