Tutorial

In this tutorial we will learn how to predict bike sharing demand using a a multiple linear regression model in Python 2.7.10 and PyCharm 2.7.3. This problem was posed in a Kaggle competition.

Problem Statement

The idea of bike sharing consists in picking up a bicycle from one station and returning the bicycle to any other in the network. Bike sharing systems generate a big amount of data such as, for example, departure and arrival locations, sharing date and time, travel time, etc. The knowledge extracted from these data might be very useful for studying and improving city mobility. The data used in the Kaggle competition came from Washington D. C.'s Capital Bikeshare program. The objective is to combine historical usage patterns with weather data in order to forecast bike rental demand in Washington, D.C.

Prerequisites

Before proceeding with the model, please check that the following packages are installed on your system:

- scipy
- numpy
- pandas
- sklearn
- matplotlib

Next, let's create a folder called **data**, download data sets (train.csv and test.csv) from Kaggle and save them in a newly created folder. Since files are really small (633.16 kb and 316.27 kb, accordingly), we will load them directly into Python.

Loading data and packages

The first thing we need to do is to load pandas package and two CSV files that we've just downloaded from Kaggle. Other packages can be added later to the top of the code.

```
# Loading packages
import pandas as pd

# Loading data
train = pd.read_csv('data/train.csv', parse_dates=[0])
test = pd.read_csv('data/test.csv', parse_dates=[0])
The training set contains the following variables:
```

- **datetime** – hourly date + timestamp
- **season** – 1 = spring, 2 = summer, 3 = fall, 4 = winter
- **holiday** – whether the day is considered a holiday
- **workingday** – whether the day is neither a weekend nor holiday

- **weather:** 1) Clear, Few clouds, Partly cloudy, Partly cloudy; 2) Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist; 3) Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds; 4) Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
- **temp** – temperature in Celsius
- **atemp** – "feels like" temperature in Celsius
- **humidity** – relative humidity
- **windspeed** – wind speed
- **casual** – number of non-registered user rentals initiated
- **registered** – number of registered user rentals initiated
- **count** – number of total rentals (**casual** + **registered**)

The data is comprised of hourly rental statistics spanning two years. The training set includes the first 19 days of each month, while the test set is the 20th to the end of the month. It is important to notice that the test set contains similar variables, excluding **casual**, **registered** and **count**. Why does the test set not contain these variables? That's because we need to predict them. In fact, the general idea of the competition is to build the model using the training set, then run the model on the test set and submit the achieved results to Kaggle. In other words we don't know correct predictions for the test set. The only way to get some idea of the efficiency of the model on the test set is to upload the output to Kaggle and check the rank that is estimated automatically by Kaggle using the evaluation function (see the next section).

Evaluation function

The evaluation metric that Kaggle uses in this competition to rank the results is the Root Mean Squared Logarithmic Error (RMSLE) that is calculated as follows:

$$J = \sqrt{\frac{1}{n}\sum_{i=1}^{n}[\ln(p_i+1)-\ln(a_i+1)]^2}$$

where,

- *n* is the number of hours in the test set
- *pi* is the predicted count
- *ai* is the actual count
- *ln*(*x*) is the natural logarithm

In our model the dependent variable will be **count** transformed into log domain (**log_count**). This transformation is made to linearize a relationship between dependent and independent variables. Please note that "1" is added to demand count in order to avoid infinities associated with times where demand is nil.

```
import numpy as np

for col in ['casual', 'registered', 'count']:
    train['log-' + col] = train[col].apply(lambda x: np.log1p(x))
```

Exploratory analysis of data

To better understand our data, we will perform a quick exploratory analysis. Let's first take a look at the data.

```
print train.head()
             datetime  season  holiday  workingday  weather  temp   atemp  \
0 2011-01-01 00:00:00       1        0           0        1  9.84  14.395
1 2011-01-01 01:00:00       1        0           0        1  9.02  13.635
2 2011-01-01 02:00:00       1        0           0        1  9.02  13.635
3 2011-01-01 03:00:00       1        0           0        1  9.84  14.395
4 2011-01-01 04:00:00       1        0           0        1  9.84  14.395

   humidity  windspeed  casual  registered  count
0        81          0       3          13     16
1        80          0       8          32     40
2        80          0       5          27     32
3        75          0       3          10     13
4        75          0       0           1      1
```

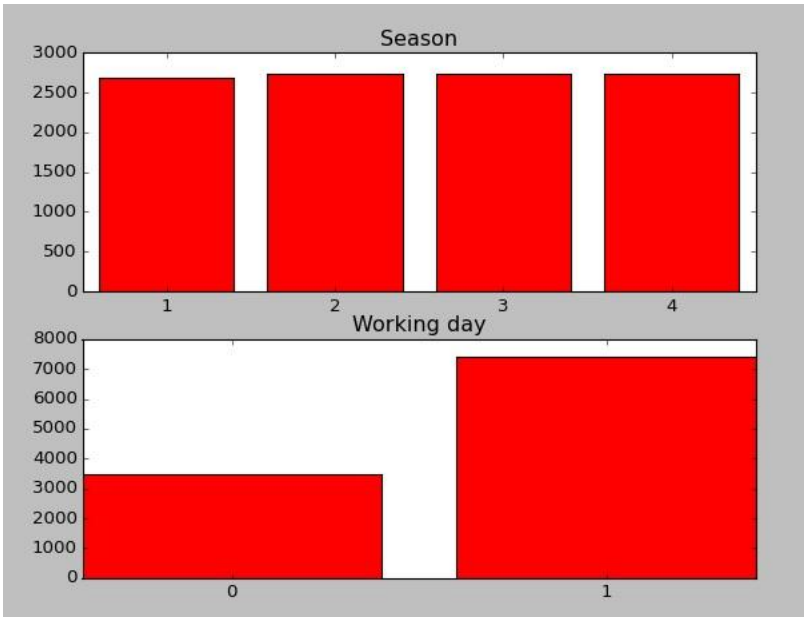
One of the most important pieces of information that can be gleaned from data is the number of times a particular value for each variable was observed. Essentially, this is summarized as a list of all possible outcomes combined with the number of times each occurred (i.e., the frequency of occurrence). This information is then displayed in a table or a barplot (for discrete variables) or histogram (for continuous variables).

Let's illustrate it with a couple of examples using variables such as **season**, **workingday**, **temp** and **atemp**.

Since **season** and **workingday** are discrete variables, we will display them in a barplot. To access the y (counts) and x (values) ticks properly, we will use **counts.values()** and **counts.keys()**, accordingly.

```
import matplotlib.pyplot as plt
import collections

fig, axes = plt.subplots(nrows=2)

counts = collections.Counter(train['season'].values)
axes[0].bar(counts.keys(), counts.values(), color='red',align='center')
axes[0].set(title='Season')
axes[0].set_xticks(counts.keys()) # aligns the bars to the center, rather
than left edge

counts = collections.Counter(train['workingday'].values)
axes[1].bar(counts.keys(), counts.values(), color='red',align='center')
axes[1].set(title='Working day')
axes[1].set_xticks(counts.keys())

plt.show()
```

From this chart we can see that the number of bike sharing demand's observations is distributed almost equally among four seasons, while most of the entries belong to working days rather than non-working days.

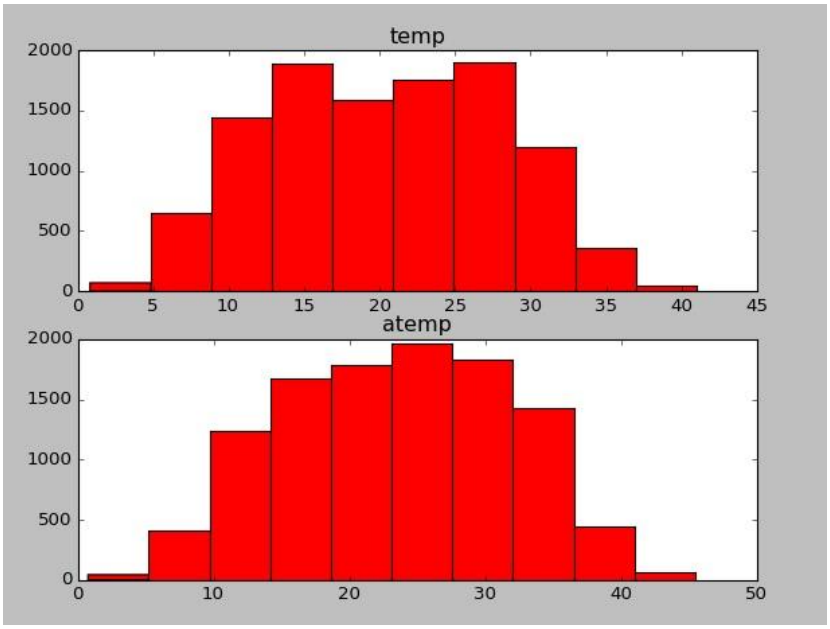


Now let's switch to continuous variables **temp** and **atemp**, and display them in a histogram (the probability distribution of a continuous variable).

Please notice that a histogram is constructed in a bit different way than a barplot. Instead of counting the frequency of particular value, the entire range of values is divided into a series of intervals (i.e. to "bin" the range of values) and the number of values falling into each interval is counted.

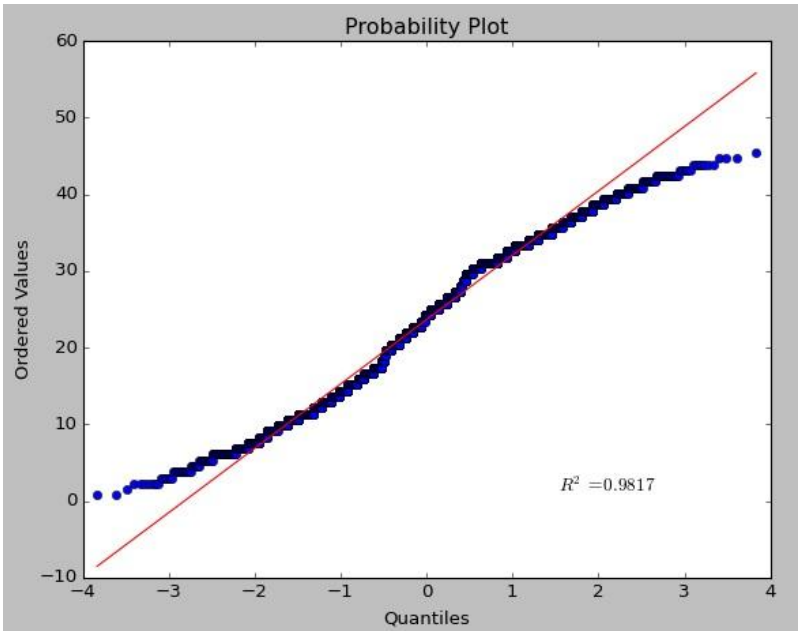
```
fig, axes = plt.subplots(nrows=2)
axes[0].hist(train['temp'].values, color='red')
axes[0].set(title='temp')
axes[1].hist(train['atemp'].values, color='red')
axes[1].set(title='atemp')
plt.show()
```

We can see that the pattern of **atemp** is symmetric and most of data entries have the "feels like" temperature between 20 and 30 degrees Celsius. The pattern of **temp** looks slightly bimodal and the typical "true" temperature is around 15 or 28 degrees Celsius.



The information that isn't readily apparent in these plots is whether the data from **temp** and **atemp** look normally distributed. Why do we need to test data for normality? This analysis helps, for instance, identify outliers. If the data has outliers, then such model as a linear regression will be unable to fit it. To perform the normality testing we will generate a QQ-plot.

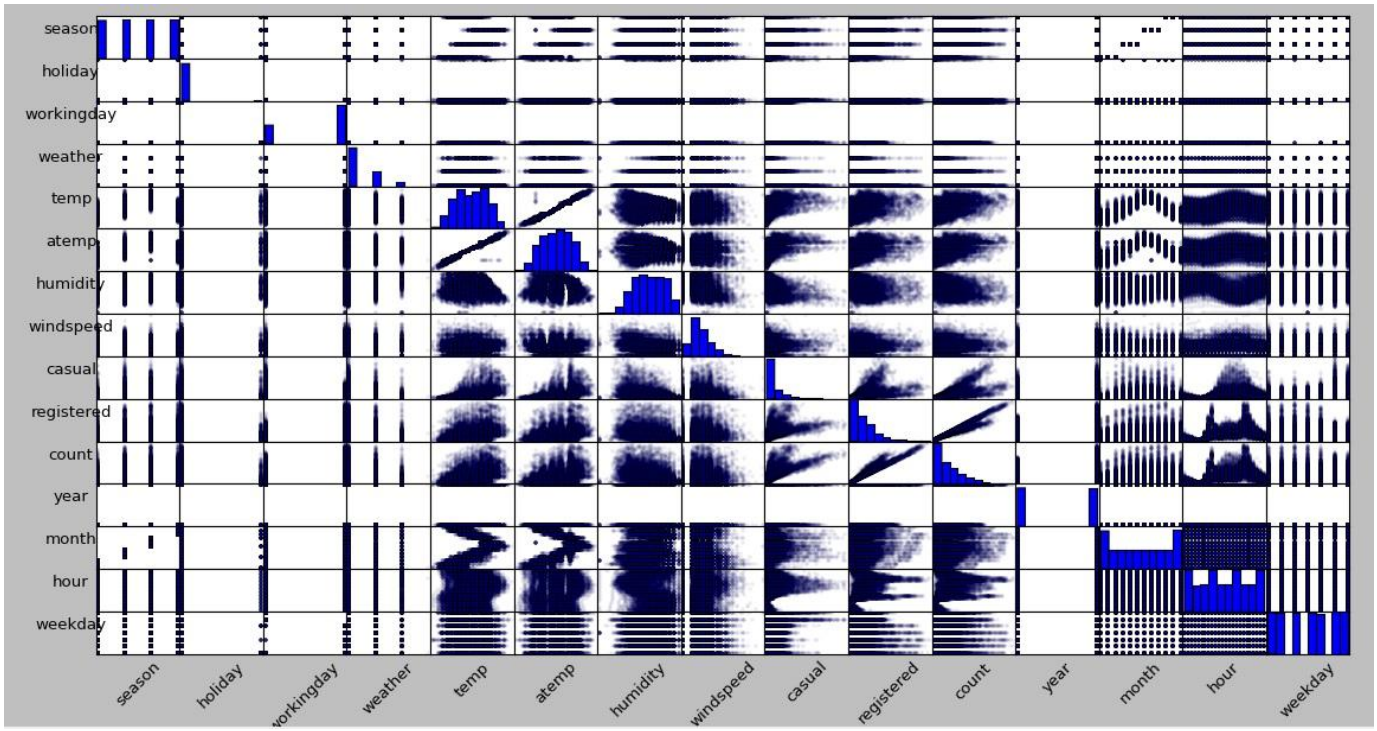
```
import scipy.stats as stats

plt.figure()
graph = stats.probplot(train['atemp'], dist="norm", plot=plt)
plt.show()
```



Till now we have not analysed the dependency of bike sharing demand on different variables. We have only checked the frequencies of particular values of variables. To analyse relationships between variables we can use a scatter matrix. (Please notice that the code below may take several seconds (but usually less than a minute) to build a scatter matrix.

```
sm = pd.scatter_matrix(train, alpha=0.05, figsize=(10,10), diagonal='hist')
.
# === This code is needed to display all labels properly ===
[s.xaxis.label.set_rotation(45) for s in sm.reshape(-1)]
[s.yaxis.label.set_rotation(0) for s in sm.reshape(-1)]

# Offset label when rotating to prevent overlap of figure
[s.get_yaxis().set_label_coords(-0.3,0.5) for s in sm.reshape(-1)]

# Hide all ticks
[s.set_xticks(()) for s in sm.reshape(-1)]
[s.set_yticks(()) for s in sm.reshape(-1)]
# ============================================================

plt.show()
```

From this plot we can observe that there is an evident linear relationship between **temp** and **atemp**, as well as **count** and **registered**. Also it can be noticed that **count**, **registered** and **casual** are quite sensitive to **temp** and **atemp**. Furthermore, **hour** would need to be categorized into, e.g. morning, day, evening and night, or so, in order to better represent a functional relationship with **count**, **casual** and **registered**.



Feature engineering

The next (usually the most critical) step is to build features that will be used for training our model. There are no definite path for feature engineering. It is very much guided by available data. Therefore data exploration that we performed in the previous step may lead to useful conclusions.

In generally the following additional tricks might be helpful:

- Converting continuous variables into categorical variables
- Merging or splitting of features to get better predictor variables
- Considering the square or cube (or using non-linear models) of the features to provide better insights
- Forward selection: start with the strongest feature (the one that has highest correlation with a predicted variable) and keep adding more features (computationally expensive)
- Backward selection: start with all the features and remove the weakest features (computationally expensive)
- Using Principal Component Analysis (PCA) or Singular Value Decomposition (SVD) to find the right combination of features

In this tutorial we will only split the feature **datetime** into new variables: the year, month, day of week and hour. This can be easily done using *pandas* packages.

```
temp_train = pd.DatetimeIndex(train['datetime'])
train['year'] = temp_train.year
train['month'] = temp_train.month
train['hour'] = temp_train.hour
train['weekday'] = temp_train.weekday

temp_test = pd.DatetimeIndex(test['datetime'])
test['year'] = temp_test.year
test['month'] = temp_test.month
test['hour'] = temp_test.hour
test['weekday'] = temp_test.weekday

#Features vector
features = ['season', 'holiday', 'workingday', 'weather',
        'temp', 'atemp', 'humidity', 'windspeed', 'year',
         'month', 'weekday', 'hour']
```

Model building

We will model the relationships between **features** and **log-count** using multiple linear regression model. In other words we will assume that the relationship between each feature and **log-count** is linear. What does it mean? Basically it means that in the equation that defines Y we will not use any non-linear transformation of independent variables (i.e. no powers, no sinuses, etc.).

Note: In our model **log-count** denotes the "dependent" variable whose values we wish to predict, and **features** denote the "independent" variables from which we wish to predict it.

First we split the data into training and validation sets. You may wonder what's the difference between test (uploaded from test.csv) and validation sets. The thing is that the validation set is used for model selection, while the test set for final model prediction error. We will use 95% of data for training and 5% for validation. Ideally, data entries should be randomly mixed to guarantee more generalized splitting, but we will ignore it in this tutorial.

```
newtraining, validation = train[:int(0.95*len(train))],
train [int(0.95*len(train)):]
```

Next we will train the linear regression model on the **newtraining** set. Least squares method will search for the best fitting regression equation to minimize the difference between predictions and true values of the dependent variable.

```
import statsmodels.api as st

X = st.add_constant(newtraining[features])
model = st.OLS(newtraining['log-count'],X)   # OLS stands for Ordinary Least
Squares
f = model.fit()
```

To get some idea about the quality of our model, we should call **f.summary()** that will return the following result.

```
print f.summary()
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:              log-count   R-squared:                       0.483
Model:                            OLS   Adj. R-squared:                  0.483
Method:                 Least Squares   F-statistic:                     678.3
Date:                Thu, 10 Sep 2015   Prob (F-statistic):               0.00
Time:                        22:54:20   Log-Likelihood:                -12522.
No. Observations:                8709   AIC:                         2.507e+04
Df Residuals:                    8696   BIC:                         2.516e+04
Df Model:                          12
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
const       -867.1349     50.300    -17.239      0.000      -965.735  -768.535
season        -0.0790      0.041     -1.911      0.056        -0.160     0.002
holiday        0.0705      0.081      0.871      0.384        -0.088     0.229
workingday     0.0169      0.040      0.428      0.669        -0.061     0.094
weather       -0.0260      0.019     -1.348      0.178        -0.064     0.012
temp          -0.0230      0.012     -1.938      0.053        -0.046     0.000
atemp          0.0642      0.011      5.878      0.000         0.043     0.086
humidity      -0.0130      0.001    -18.682      0.000        -0.014    -0.012
windspeed      0.0057      0.001      3.864      0.000         0.003     0.009
year           0.4325      0.025     17.295      0.000         0.383     0.481
month          0.0812      0.014      5.963      0.000         0.055     0.108
weekday        0.0285      0.009      3.133      0.002         0.011     0.046
hour           0.0984      0.002     58.673      0.000         0.095     0.102
==============================================================================
Omnibus:                      109.323   Durbin-Watson:                   0.518
```

```
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               113.304
Skew:                          -0.279   Prob(JB):                     2.49e-25
Kurtosis:                       2.987   Cond. No.                     9.26e+06
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 9.26e+06. This might indicate that there
are
strong multicollinearity or other numerical problems.
```

Quantities of interest can be extracted directly from the fitted model. Type dir(results) for a full list. Here are some examples:

```
print f.params

const        -867.134881
season         -0.078964
holiday         0.070463
workingday      0.016938
weather        -0.026020
temp           -0.023014
atemp           0.064197
humidity       -0.013016
windspeed       0.005714
year            0.432451
month           0.081241
weekday         0.028459
hour            0.098397
print f.rsquared  # 0.483492732406
```

What are a few key things we learn from this output?

- `season`, `holiday`, `workingday`, `weather` and `temp` have insignificant **p-values**, whereas other predictors do. Thus we fail to reject the null hypothesis for these 5 predictors (it means that there is no association between those features and count).
- A statistical measure R-squared is low (equal to 0.483), which means that our model provides insufficiently good fit to the data.

The core for understanding why our model provides poor results lies in assumptions of linear models. In fact, linear models rely upon a lot of assumptions, for instance, the features should not be correlated with each other, residuals should not be dependent, there should be no significant outliers, etc. If those assumptions are violated (which they usually are at the first stage of the analysis), R-squared and p-values become less reliable.

Though it is evident that out model does not have sufficient explanatory power, let's complete our experiment for education purposes, assuming that we have achieved a good model.

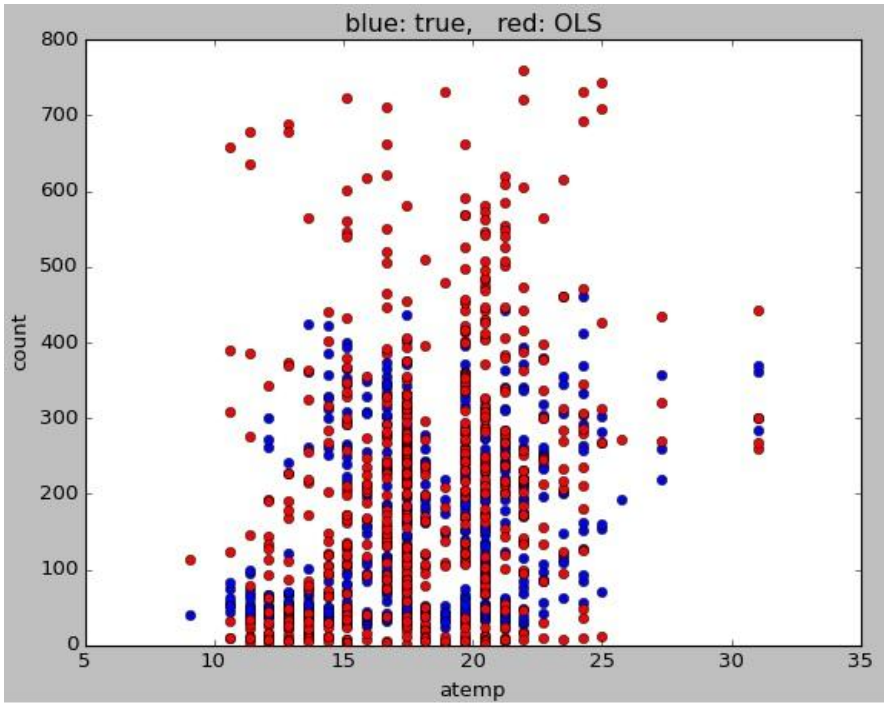
Once we trained the model, the next thing to do is to check how the model performs on the unseen data from the validation set, for which we know the values of 'log-count' (please recall that in contrast to the validation set, the test set does not contain the column "count").

In the below code we should apply a small trick. Since validation data has columns with all identical values (e.g. **year**, **season**), we need to add the constant manually to the DataFrame (otherwise the prediction function will not work).

```
validnew=validation[features]
validnew.insert(0, 'const', 1)
ypred = f.predict(validnew)
```

When dealing with multiple linear regression, fits to data are no lines. For instance, if we had two independent variables X1 and X2, the resulting fit would describe a plane in three dimensional space. If the number of independent variables is greater than 2, the resulting fit becomes difficult to visualize. Therefore, we will visualize the true and predicted relationships between independent variables and **count** separately using a simple 2D plot. As an independent variable let's select **atemp** that according to summary statistics has a p-value<0.05 (i.e. significant variable). From the plot below it's evident that some variance in **count** is still insufficiently explained by **atemp**.

```
fig, ax = plt.subplots()
plt.plot(validnew['atemp'], np.expm1(ypred), 'o', validation['atemp'],
np.expm1(validation['log-count']), 'ro');
ax.set_title('blue: true, red: OLS')
ax.set_xlabel('count')
ax.set_ylabel('atemp')
plt.show()
```



The same way we can analyze the impact of other variables on **count**.

The final step is to apply the multiple linear regression model to the test set and convert the predicted log-demand (i.e. **log-count**) into linear scale.

```
testnew=test[features]
testnew.insert(0, 'const', 1)
ypredtest = f.predict(testnew)
result = [round(np.expm1(x)) for x in ypredtest]
# or result = np.round(np.expm1(ypredtest)) since ypredtest is a NumPy array
```

We will save the prediction result in a separate CSV file with two columns ("datetime" and "count") as requested by the competition rules and upload this file to the Kaggle submission system.

```
df=pd.DataFrame({'datetime':test['datetime'], 'count':result})

df.to_csv('output/linearregression_output.csv', index = False,
columns=['datetime','count'])
```

Kaggle will calculate a public score that should be equal to 1.04630 for this submission. Since the entry was not submitted during the competition, the score will not be saved in a Public Leaderboard. As expected, we obtained a very low rank using linear regression model.

Homework

1. Follow the tutorial to create a basic multiple regression model and submit results on Kaggle.

2. Try to improve this model in order to slightly increase a score on a Kaggle leaderboard. Use the following starting points:

- Create additional informative features (independent variables). Explore the data and application area (i.e. bike sharing) to get ideas. For instance, what could we do with the variable **hour**? Could we represent it as a categorical variable with values **night**, **morning**, **day**, **evening**? Create a scatter matrix for the new variables to see how they impact on the dependent variable.

- When we fit a linear regression model to a particular data set, many problems may occur. Most common among these are the following:
  - ✓ Non-linearity of the response-predictor relationships.
  - ✓ Correlation of residuals (error terms)
  - ✓ Non-constant variance of residuals (error terms)

- ✓ Outliers
- ✓ Collinearity

In practice, identifying and overcoming these problems is as much **an art as a science**. The task is to check the model in terms of these issues and try to resolve as much as you can.
For instance, non-linearity of the dependent-independent relationship can be resolved by "transforming" the independent variable (taking it logarithm, square, etc.).

3. When finished (Wednesday 23rd as the latest), please send me your model and a short report.