Predictive Analysis on Beijing PM2.5 Data Data Set

This is the final project in the course "Meaningful Predictive Modeling". The project consists of making simple predictions on the structured data set "Beijing PM2.5 Data". The goal of the project is to deal with practical issues, such as training/test splits, avoiding overfitting, and validating that the results are significant.

Dataset Description

The "***Beijing PM2.5 Data***" is an open dataset on the website of **UCI's Machine Learning Repository** .

Abstract

This is a dataset that reports on the weather and the level of pollution each hour for five years at the US embassy in Beijing, China. It contains the PM2.5 data. Meanwhile, meteorological data from Beijing Capital International Airport are also included.

Data Set Information

The data's time period is between Jan 1st, 2010 to Dec 31st, 2014. Missing data are denoted as "NA".

Attribute Information

The complete feature list in the raw data is as follows:

- No: row number
- year: year of data in this row
- month: month of data in this row
- day: day of data in this row
- hour: hour of data in this row
- pm2.5: PM2.5 concentration (ug/m^3)
- DEWP: Dew Point (â„ƒ)
- TEMP: Temperature (â„ƒ)
- PRES: Pressure (hPa)
- cbwd: Combined wind direction
- Iws: Cumulated wind speed (m/s)
- Is: Cumulated hours of snow
- Ir: Cumulated hours of rain

Modeling Objective

In this project, we are going to use the "***Beijing PM2.5 Data***" to frame a prediction problem where, given the weather conditions and pollution for prior hours, we build a model to predict the pollution (i.e., the pm2.5 value) at the next hour.

Data Loading and Exploratory Data Analysis

In [1]:

```
import pandas as pd
import numpy as np
from datetime import datetime
import matplotlib.pyplot as plt
import seaborn as sns

# load data
def parse(x):
    return datetime.strptime(x, '%Y %m %d %H')

df = pd.read_csv('data/PRSA_data_2010.1.1-2014.12.31.csv',  parse_dates = [['year', 'month', 'day', 'hour']], index_col=0, date_parser=parse)
```

In [2]:

```
df.shape
```

Out[2]:

```
(43824, 9)
```

In [3]:

```
df.head()
```

Out[3]:

year_month_day_hour	No	pm2.5	DEWP	TEMP	PRES	cbwd	Iws	Is	Ir
2010-01-01 00:00:00	1	NaN	-21	-11.0	1021.0	NW	1.79	0	0
2010-01-01 01:00:00	2	NaN	-21	-12.0	1020.0	NW	4.92	0	0
2010-01-01 02:00:00	3	NaN	-21	-11.0	1019.0	NW	6.71	0	0
2010-01-01 03:00:00	4	NaN	-21	-14.0	1019.0	NW	9.84	0	0
2010-01-01 04:00:00	5	NaN	-20	-12.0	1018.0	NW	12.97	0	0

In [4]:

```
df.isnull().sum()
```

Out[4]:

```
No          0
pm2.5    2067
DEWP        0
TEMP        0
PRES        0
cbwd        0
Iws         0
Is          0
Ir          0
dtype: int64
```

In [5]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 43824 entries, 2010-01-01 00:00:00 to 2014-12-31 23:00:00
Data columns (total 9 columns):
No       43824 non-null int64
pm2.5    41757 non-null float64
DEWP     43824 non-null int64
TEMP     43824 non-null float64
PRES     43824 non-null float64
cbwd     43824 non-null object
Iws      43824 non-null float64
Is       43824 non-null int64
Ir       43824 non-null int64
dtypes: float64(4), int64(4), object(1)
memory usage: 3.3+ MB
```

In [6]:

```
df.describe()
```

Out[6]:

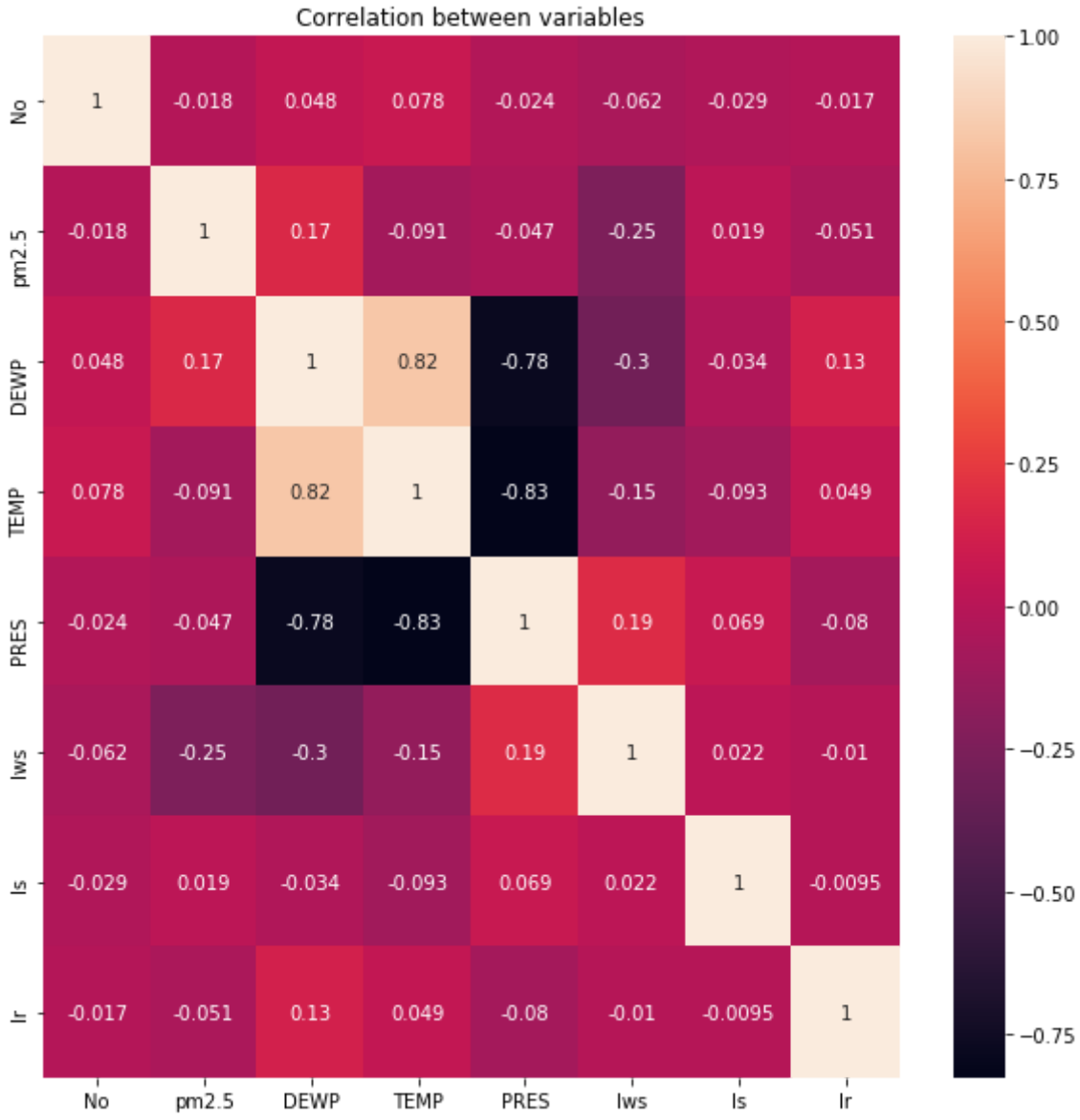
	No	pm2.5	DEWP	TEMP	PRES	Iws
count	43824.000000	41757.000000	43824.000000	43824.000000	43824.000000	43824.000000
mean	21912.500000	98.613215	1.817246	12.448521	1016.447654	23.889140
std	12651.043435	92.050387	14.433440	12.198613	10.268698	50.010635
min	1.000000	0.000000	-40.000000	-19.000000	991.000000	0.450000
25%	10956.750000	29.000000	-10.000000	2.000000	1008.000000	1.790000
50%	21912.500000	72.000000	2.000000	14.000000	1016.000000	5.370000
75%	32868.250000	137.000000	15.000000	23.000000	1025.000000	21.910000
max	43824.000000	994.000000	28.000000	42.000000	1046.000000	585.600000

In [7]:

```
attr_corr = df.corr()
fig, ax = plt.subplots(figsize = (10,10))
g= sns.heatmap(attr_corr,ax=ax, annot= True)
ax.set_title('Correlation between variables')
```

Out[7]:

```
Text(0.5, 1, 'Correlation between variables')
```

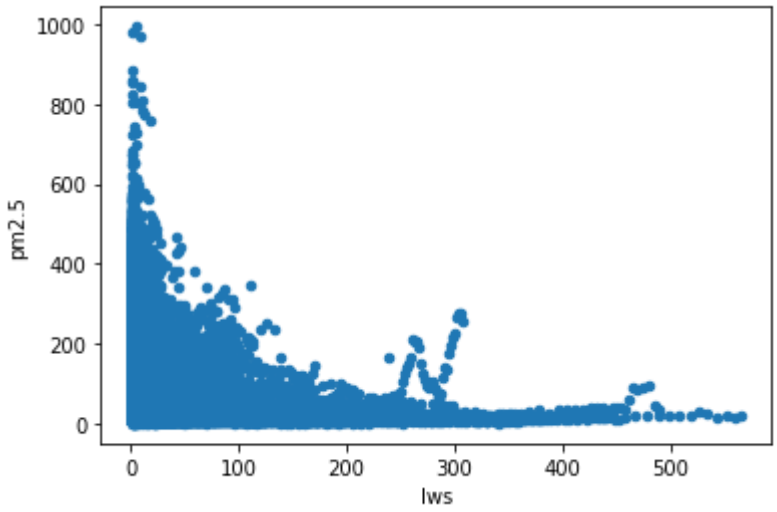


In [8]:

```
df.plot(x='Iws', y='pm2.5', kind='scatter')
```

Out[8]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x22a63bf1ec8>
```

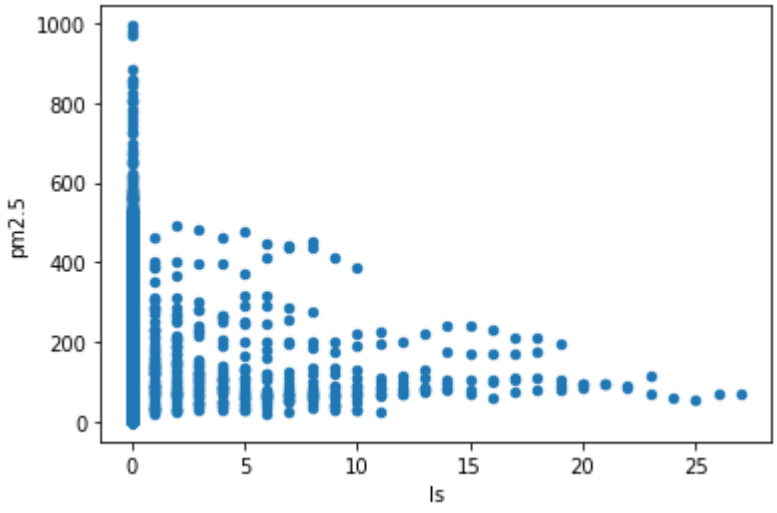


In [9]:

```
df.plot(x='Is', y='pm2.5', kind='scatter')
```

Out[9]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x22a639451c8>
```

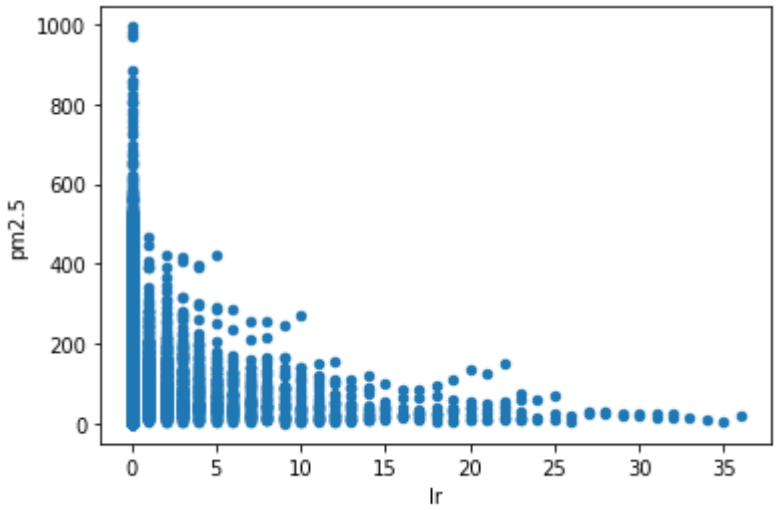


In [10]:

```
df.plot(x='Ir', y='pm2.5', kind='scatter')
```

Out[10]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x22a63815f88>
```

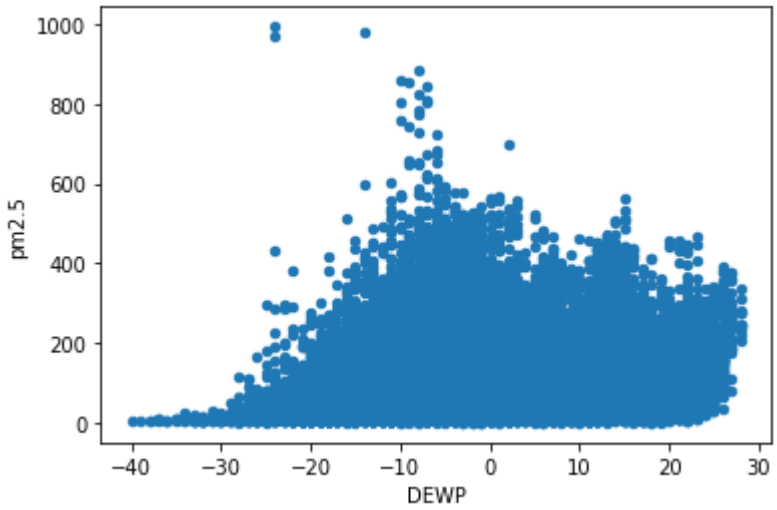


In [11]:

```
df.plot(x='DEWP', y='pm2.5', kind='scatter')
```

Out[11]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x22a637aac48>
```

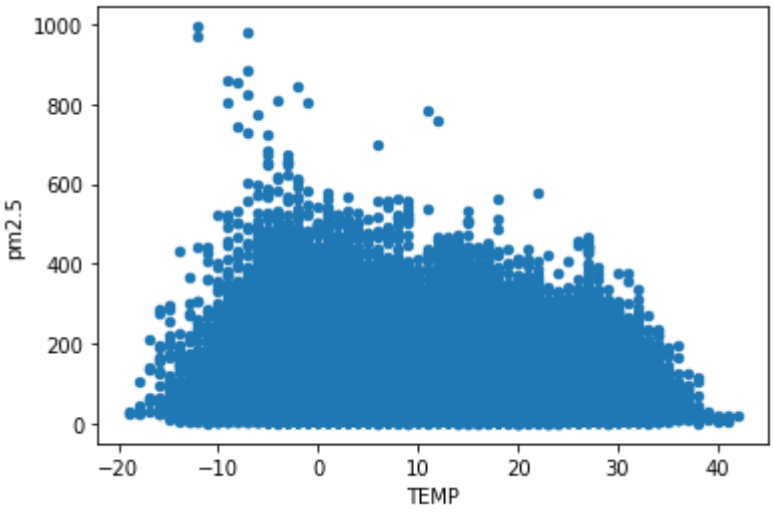


In [12]:

```
df.plot(x='TEMP', y='pm2.5', kind='scatter')
```

Out[12]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x22a63960108>
```



Data Analysis conclusion:

- The EDA reveals that there are NA values for "pm2.5" for the first 24 hours. And there are also a few scattered NA values for "pm2.5" later in the dataset. We will need to deal with these NA values properly in data preparation.
- "cbwd" (wind direction) is a categorical attribute. In data preparation, we need to encode this attribute into either numeric labels or dummy variables.
- The heatmap shows that none of the prediction attributes is much less correlated with the target variable "pm2.5" than others. So, no attributes will be ommitted by feature selection; i.e., all the attributes will be used as features to build the model.
- The scatter plots show that each weather attribute has a certain correlation with "pm2.5". For example, from the scatter plot of "pm2.5" vs. "Ir", it is pretty clear that is was extremely unlikely to have “polluted” days during rainy days. And the scatter plot of "pm2.5" vs. "Iws" shows high wind speed usually reduces the level of "pm2.5".

Data Preparation

In [13]:

```
# Firstly, we encode the categorical attribute "cbwd" into dummy variables
df_numeric = pd.get_dummies(df, drop_first=True, dtype=float)
df_numeric.head()
```

Out[13]:

year_month_day_hour	No	pm2.5	DEWP	TEMP	PRES	Iws	Is	Ir	cbwd_NW	cbwd_SE
2010-01-01 00:00:00	1	NaN	-21	-11.0	1021.0	1.79	0	0	1.0	0.0
2010-01-01 01:00:00	2	NaN	-21	-12.0	1020.0	4.92	0	0	1.0	0.0
2010-01-01 02:00:00	3	NaN	-21	-11.0	1019.0	6.71	0	0	1.0	0.0
2010-01-01 03:00:00	4	NaN	-21	-14.0	1019.0	9.84	0	0	1.0	0.0
2010-01-01 04:00:00	5	NaN	-20	-12.0	1018.0	12.97	0	0	1.0	0.0

In [14]:

```
# We then construct the dataset for the supervised modeling problem as predicting the p
ollution level for the next hour
# based on the weather conditions and pollution over the previous 24 hours.
# In constructing the dataset, the data rows with NA values either in feature columns o
r in the target column are dropped.

def series_to_supervised(data, target, window_size=1, n_steps=0, dropnan=True):
        """
        Frame a time series as a supervised learning dataset.
        Arguments:
            data: Sequence of observations as a Pandas DataFrame.
            tareget: List of the target column names.
            window_size: Length of historical window for the observations (X). (n>=1; t
ime steps: [-n,-n+1,..., -1] )
            n_steps: Number of time steps ahead for the prediction (y). (n>=0; time ste
p: n; 0 is the index of "now")
            dropnan: Boolean whether or not to drop rows with NaN values.
        Returns:
            Pandas DataFrame for supervised learning.
        """

        df_tars = data[target]
        df_vars = data

        n_tars = df_tars.shape[1]
        n_vars = data.shape[1]

        cols, names = list(), list()
        # input sequence (t-n, ... t-1)
        for i in range(window_size, 0, -1):
            cols.append(data.shift(i))
            names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]

        # forecast sequence (t+n)
        if n_steps == 0:
            cols.append(df_tars)
            names += [('tar%d(t)' % (j+1)) for j in range(n_tars)]
        else:
            cols.append(df_tars.shift(-n_steps))
            names += [('tar%d(t+%d)' % (j+1, n_steps)) for j in range(n_tars)]

        # put it all together
        agg = pd.concat(cols, axis=1)
        agg.columns = names
        # drop rows with NaN values
        if dropnan:
            agg.dropna(inplace=True)
        return agg

df_numeric = df_numeric.drop(['No'], axis=1)
df_windowed = series_to_supervised(df_numeric, ['pm2.5'], window_size=24, n_steps=0)
print(df_windowed.shape)
print(df_windowed.head())

# We select two segments from the constructed dataset for visualizing the prediction. E
ach of the segment covers two days of data.
# The time periods of the segments are as follows: [2014-02-25 00:00:00, 2014-02-26 23:
00:00] , [2014-10-15 00:00:00, 2014-10-16 23:00:00]
# For these two periods, there is no NA values of "pm2.5" over the previous 24 hours in
```

```
the raw data. Therefore, we can plot the
# prediction for each timestamp in the period as a time series.
df_viz_1 = df_windowed.loc['2014-02-25 00:00:00':'2014-02-26 23:00:00', :]
df_viz_2 = df_windowed.loc['2014-10-15 00:00:00':'2014-10-16 23:00:00', :]
```

```
(37596, 241)
                     var1(t-24)  var2(t-24)  var3(t-24)  var4(t-24)  \
year_month_day_hour
2010-01-03 00:00:00       129.0       -16.0        -4.0      1020.0
2010-01-03 01:00:00       148.0       -15.0        -4.0      1020.0
2010-01-03 02:00:00       159.0       -11.0        -5.0      1021.0
2010-01-03 03:00:00       181.0        -7.0        -5.0      1022.0
2010-01-03 04:00:00       138.0        -7.0        -5.0      1022.0

                     var5(t-24)  var6(t-24)  var7(t-24)  var8(t-24)  \
year_month_day_hour
2010-01-03 00:00:00        1.79         0.0         0.0         0.0
2010-01-03 01:00:00        2.68         0.0         0.0         0.0
2010-01-03 02:00:00        3.57         0.0         0.0         0.0
2010-01-03 03:00:00        5.36         1.0         0.0         0.0
2010-01-03 04:00:00        6.25         2.0         0.0         0.0

                     var9(t-24)  var10(t-24)  ...  var2(t-1)  var3(t-1)  \
year_month_day_hour                           ...
2010-01-03 00:00:00         1.0          0.0  ...       -8.0       -6.0
2010-01-03 01:00:00         1.0          0.0  ...       -7.0       -6.0
2010-01-03 02:00:00         1.0          0.0  ...       -8.0       -6.0
2010-01-03 03:00:00         1.0          0.0  ...       -8.0       -7.0
2010-01-03 04:00:00         1.0          0.0  ...       -8.0       -7.0

                     var4(t-1)  var5(t-1)  var6(t-1)  var7(t-1)  var8(t-1)
\
year_month_day_hour
2010-01-03 00:00:00     1027.0      55.43        3.0        0.0        0.0
2010-01-03 01:00:00     1027.0      58.56        4.0        0.0        0.0
2010-01-03 02:00:00     1026.0      61.69        5.0        0.0        0.0
2010-01-03 03:00:00     1026.0      65.71        6.0        0.0        0.0
2010-01-03 04:00:00     1025.0      68.84        7.0        0.0        0.0

                     var9(t-1)  var10(t-1)  tar1(t)
year_month_day_hour
2010-01-03 00:00:00        1.0         0.0     90.0
2010-01-03 01:00:00        1.0         0.0     63.0
2010-01-03 02:00:00        1.0         0.0     65.0
2010-01-03 03:00:00        1.0         0.0     55.0
2010-01-03 04:00:00        1.0         0.0     65.0

[5 rows x 241 columns]
```

In [15]:

```
# We normalize features by using MinMaxScaler
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(df_windowed)

viz_1_scaled = scaler.transform(df_viz_1)
viz_2_scaled = scaler.transform(df_viz_2)
```

In [16]:

```
scaled[:20, :]
```

Out[16]:

```
array([[0.12977867, 0.35294118, 0.24590164, ..., 1.        , 0.        ,
        0.09054326],
       [0.14889336, 0.36764706, 0.24590164, ..., 1.        , 0.        ,
        0.06338028],
       [0.15995976, 0.42647059, 0.2295082 , ..., 1.        , 0.        ,
        0.06539235],
       ...,
       [0.16498994, 0.47058824, 0.2295082 , ..., 1.        , 0.        ,
        0.07042254],
       [0.17102616, 0.47058824, 0.2295082 , ..., 1.        , 0.        ,
        0.06136821],
       [0.1498994 , 0.47058824, 0.2295082 , ..., 0.        , 1.        ,
        0.05331992]])
```

Building Regression Model to Perform Prediction

In [17]:

```
import random

# Shuffle the dataset
random.shuffle(scaled)

# Split the dataset into training, validation, and test samples by using 70%/15%/15% sp
lits
N = scaled.shape[0]
idx_valid = int(N * 0.7)
idx_test = int(N * 0.85)
train = scaled[: idx_valid, :]
valid = scaled[idx_valid : idx_test, :]
test = scaled[idx_test : , :]

train_X, train_y = train[:, :-1], train[:, -1]
valid_X, valid_y = valid[:, :-1], valid[:, -1]
test_X, test_y = valid[:, :-1], valid[:, -1]
print(train_X.shape, train_y.shape, valid_X.shape, valid_y.shape, test_X.shape, test_y.
shape)
```

```
(26317, 240) (26317,) (5639, 240) (5639,) (5639, 240) (5639,)
```

In [18]:

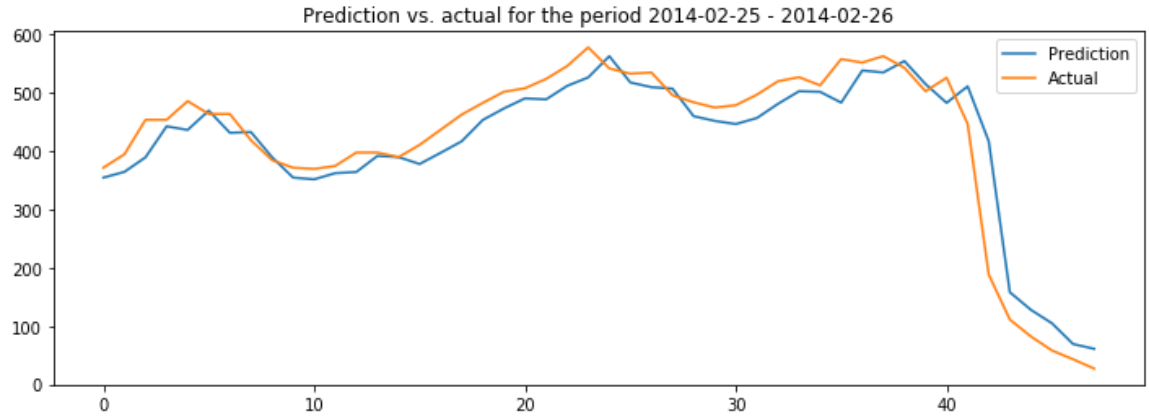
```
from sklearn import linear_model

# As in the course, we use the "Ridge" model from scikit-learn, which allows to impleme
nt regression with a regularizer
def MSE(model, X, y):
    predictions = model.predict(X)
    differences = [(a - b) ** 2 for (a, b) in zip(predictions, y)]
    return sum(differences) / len(differences)

best_model = None
best_MSE = None
best_lamb = None

# Train the model for a range of regularization coefficients, and select the best lambd
a that derives the least validation error
for lamb in [0.001, 0.1, 1.0, 10.0, 20.0, 40.0, 60.0, 80.0, 100.0]:
    model = linear_model.Ridge(lamb, fit_intercept=False)
    model.fit(train_X, train_y)

    MSE_train = MSE(model, train_X, train_y)
    MSE_valid = MSE(model, valid_X, valid_y)

    print('lambda = %.3f, training/valid error = %.5f/%.5f' %(lamb, MSE_train, MSE_vali
d))
    if not best_model or MSE_valid < best_MSE:
        best_model = model
        best_MSE = MSE_valid
        best_lamb = lamb

print('The best model: lambda =  %.3f, valid error = %.5f' %(best_lamb, best_MSE) )

MSE_test = MSE(best_model, test_X, test_y)
print('test error = %.5f' %(MSE_test))
```

```
lambda = 0.001, training/valid error = 0.00064/0.00062
lambda = 0.100, training/valid error = 0.00064/0.00062
lambda = 1.000, training/valid error = 0.00064/0.00062
lambda = 10.000, training/valid error = 0.00074/0.00068
lambda = 20.000, training/valid error = 0.00082/0.00075
lambda = 40.000, training/valid error = 0.00095/0.00087
lambda = 60.000, training/valid error = 0.00106/0.00097
lambda = 80.000, training/valid error = 0.00115/0.00106
lambda = 100.000, training/valid error = 0.00122/0.00113
The best model: lambda =  1.000, valid error = 0.00062
test error = 0.00062
```

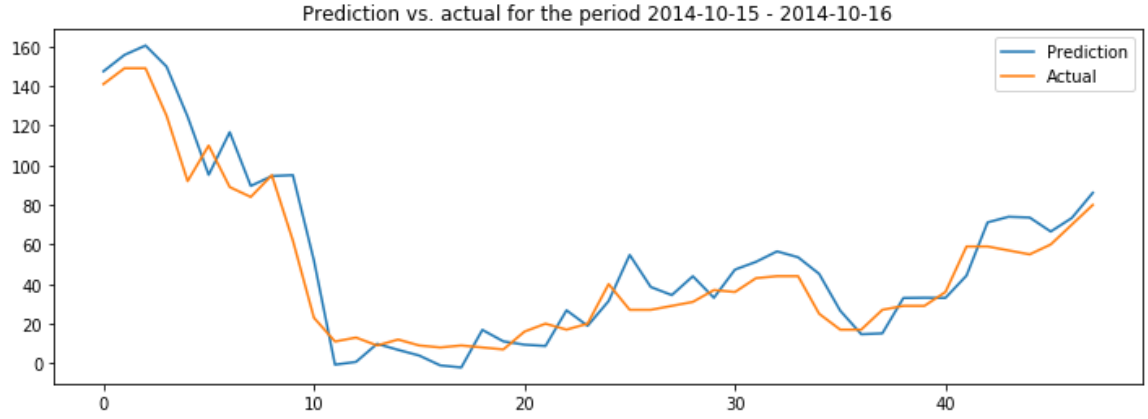
In [19]:

```
# Visualize the predictions on the Visualization datasets

for viz_scaled, period in [(viz_1_scaled, '2014-02-25 - 2014-02-26'), (viz_2_scaled, '2
014-10-15 - 2014-10-16')]:

    # Get the X, y from the dataset
    viz_X, viz_y = viz_scaled[:, :-1], viz_scaled[:, -1]

    # Use the model to generate the predictions
    viz_yhat = best_model.predict(viz_X)
    MSE_test = MSE(best_model, viz_X, viz_y)

    # invert scaling for predictions
    viz_yhat = viz_yhat.reshape((len(viz_yhat), 1))
    inv_yhat = np.concatenate((viz_X, viz_yhat), axis=1)
    inv_yhat = scaler.inverse_transform(inv_yhat)
    prediction = inv_yhat[:,-1]

    # invert scaling for actual
    inv_y = scaler.inverse_transform(viz_scaled)
    actual = inv_y[:,-1]

    print('Test MSE: %.5f' % MSE_test)
    # Plot the prediction vs. actual
    plt.figure(figsize=(12, 4))
    plt.title('Prediction vs. actual for the period {}'.format(period), fontsize=12)
    plt.plot(prediction, label='Prediction')
    plt.plot(actual, label='Actual')

    plt.legend()
    plt.show()
    print('\n')
```

```
Test MSE: 0.00218
```




```
Test MSE: 0.00019
```



Conclusion:

By properly splitting the data into training, validation, and test sets, we can successfully avoid overfitting, and build a model in good quality to predict the pollution level based on the weather conditions and pollution over the previous 24 hours.

In [ ]: