Programming Exercise - Polynomial regression

May 8, 2019

1 Polinomial regression

When we have to deal with regression (linear or non) problems, it is usually necessary to solve an optimization problem. This happens because fitting a model means that we have to find some parameters that better approximate our data. In the course of this exercise, we will use two different kinds of optimizations: - **gradient descent**, which is an iterative algorithm; - **normal equations**, which is an analytical method.

The differences between the two approaches are that the former (eventually) requires a scaling of the features (in the case that there are different in order of magnitude) to be used. The latter doesn't require any scaling, but it can be slower.

It is up to us to choose the best optimization method to use, considering the dataset over which we will optimise the model.

1.0.1 Imports and definitions

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

import pandas as pd
import seaborn as sns

%matplotlib inline

from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

boston_dataset = load_boston()
```

[2]:
```
def gradient_descent_vectorized(x, y, theta = [[0], [0]], alpha = 0.01,␣
 ↪num_iters = 400, epsilon = 0.0001):
    J_history = np.zeros((num_iters))
    early_stop = -1;
    for k in range(num_iters):
        h = x.dot(theta)
        theta = theta - (alpha/m)*(x.T.dot(h-y))
        J_history[k] = compute_cost_vectorized(x, y, theta)
```

```
    return theta, J_history
def compute_cost_vectorized(x, y, theta):
    h = x.dot(theta)
    J = (h-y).T.dot(h-y)
    return J/(2*m)
def find_flat(history, epsilon = 0.001):
    for k in range(1, history.size):
        if (history[k-1] - history[k] < epsilon):
            return k;
    return -1
def normal_equations(x, y):
    return np.linalg.pinv(x.T.dot(x)).dot(x.T).dot(y)
def polynomial_features(x, degree):
    for i in range(1, degree):
        label = VARIABLE + '_%d'%(i+1)
        x[label] = x[VARIABLE]**(i+1)
    return x
```

Let's see the name of the features in the dataset:

[3]:
```
print(boston_dataset.keys())
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])
```

We can access to the description of the dataset and the explanation of the features in it.

[4]:
```
print(boston_dataset.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value
(attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000
sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0
otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
```

```
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by
town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/


This dataset was taken from the StatLib library which is maintained at Carnegie
Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that
address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential
Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In
Proceedings on the Tenth International Conference of Machine Learning, 236-243,
University of Massachusetts, Amherst. Morgan Kaufmann.
```

Let's see the first 5 examples in the dataset

[5]:
```python
boston = pd.DataFrame(boston_dataset.data, columns=boston_dataset.feature_names)
boston.head()
```

[5]:
```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0  296.0
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
```

```
3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0
4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0

   PTRATIO       B  LSTAT
0     15.3  396.90   4.98
1     17.8  396.90   9.14
2     17.8  392.83   4.03
3     18.7  394.63   2.94
4     18.7  396.90   5.33
```

In the dataset, as it can be seen, the **target feature** is missing, we can add it as follows:

[6]:
```python
boston['MEDV'] = boston_dataset.target
boston.head()
```

[6]:
```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0  296.0
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0
4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0

   PTRATIO       B  LSTAT  MEDV
0     15.3  396.90   4.98  24.0
1     17.8  396.90   9.14  21.6
2     17.8  392.83   4.03  34.7
3     18.7  394.63   2.94  33.4
4     18.7  396.90   5.33  36.2
```

The dataframe has some usefull utility functions too, like the one we can use to see the *spurious* examples, which count the number of null values inside the dataset:

[7]:
```python
boston.isnull().sum()
```

[7]:
```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
MEDV       0
dtype: int64
```

1.0.2 Plot the 'Pearson' Correlation matrix

There are a lot of features in the dataset, which can be a problem in the training phase when we have a huge number of training examples. Instead of considering all the features in the dataset, we can use the ones which are indipendent between them. The order of correlation between the features can be calculated using the **Pearson correlation index**, which is defined as follows:

$$\rho_{x,y} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}$$

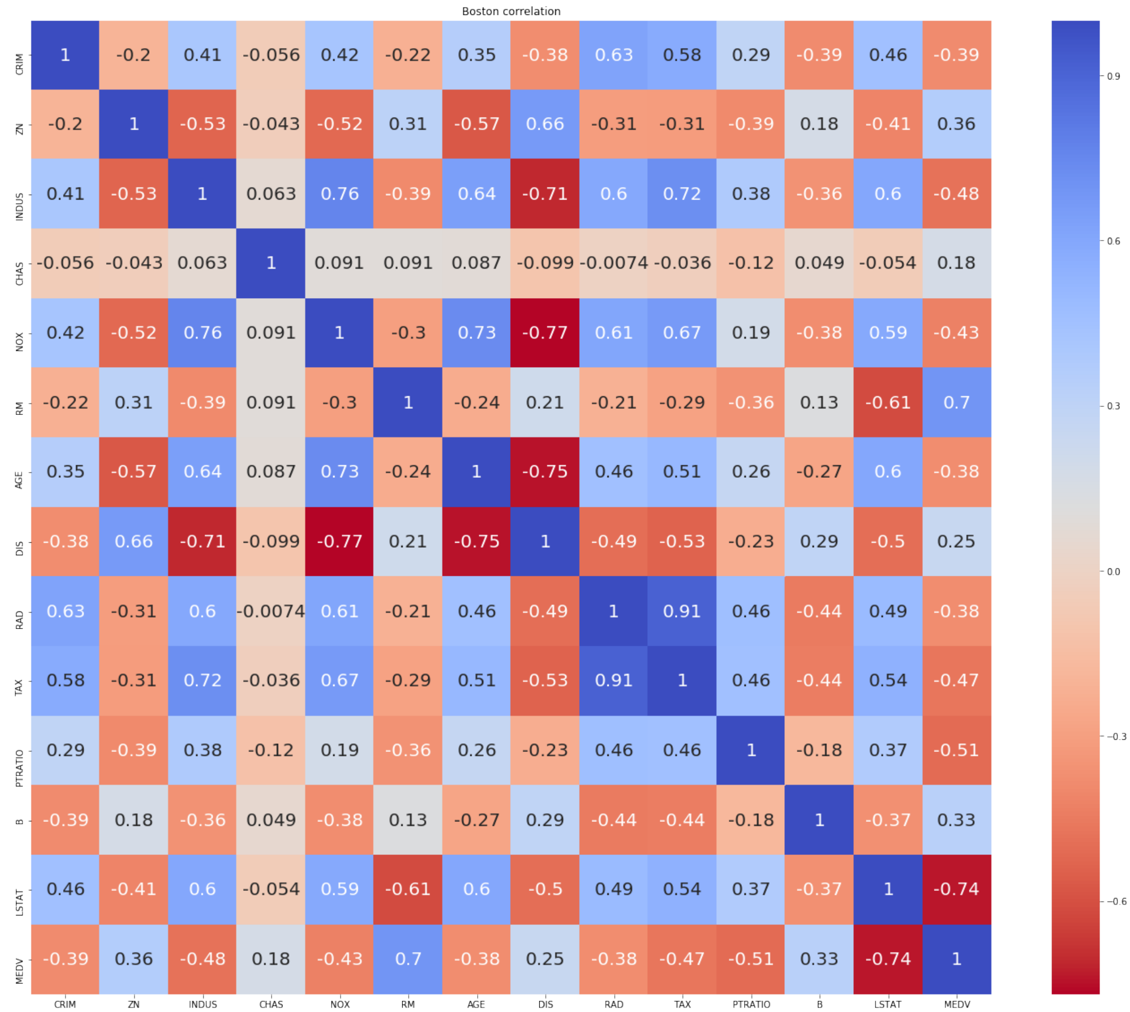
where: - the numerator represents the covariance - the denominator represents the product of the standard deviations

It is true that:

$$\rho_{x,y} \begin{cases} > 0 & \text{if } x \text{ and } y \text{ are positively correlated,} \\ = 0 & \text{if } x \text{ and } y \text{ are not correlated,} \\ < 0 & \text{if } x \text{ and } y \text{ are negatively correlated} \end{cases}$$

This index captures just the linear correlation, not the more complex ones (like the non-linear).

[8]:
```
f, (ax1) = plt.subplots(1, 1, figsize = (24, 20))
corr = boston.corr(method = "pearson")
sns.heatmap(corr, cmap = 'coolwarm_r', annot = True, annot_kws = {'size': 20},␣
↪ax = ax1)
ax1.set_title('Boston correlation')
plt.show()
```



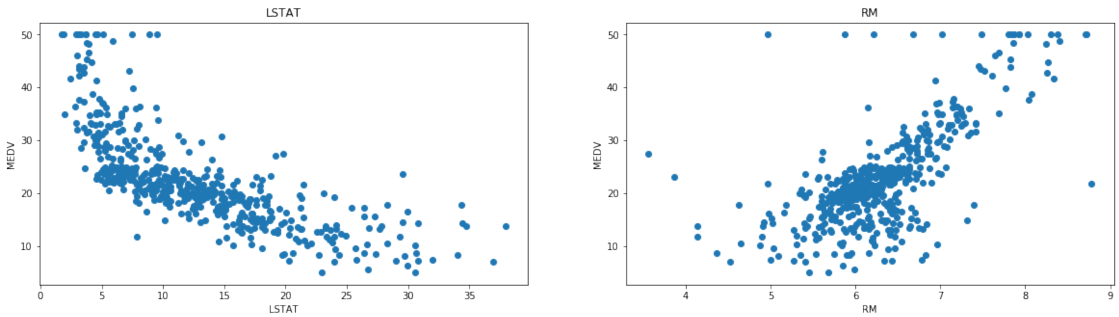
The table shows all the correlation indeces between every couple of features. We need to look for features that are correlated to the target feature: this can be seen in the last row of the Pearson matrix.

As it can be seen, there are two features that are highly correlated with the target feature: - **RM**, which has a correlation index of **0.7** - **LSTAT**, which has a correlation index of **-0.74**

Because of the high degree of correlation between these features, we should not use both of them in the training phase, because using both of them could potentially bring numerical instability in the solution.

[9]:
```
plt.figure(figsize=(20, 5))

features = ['LSTAT', 'RM']
target = boston['MEDV']

for i, col in enumerate(features):
    plt.subplot(1, len(features) , i+1)
    x = boston[col]
```

```
    y = target
    plt.scatter(x, y, marker='o')
    plt.title(col)
    plt.xlabel(col)
    plt.ylabel('MEDV')
```



1.0.3 Linear regression with one variable

The linear regression model isnt' always the best fit to analyze data, even if it adapts well to data.

It can happens that the algorithm fit the data on the training set used, but it can do bad on a validation or test set (which means on data that the algorithm has never seen).

Let's apply a linear regression on a unique feature

Let's build our custom dataset, using only the feature we will consider

[10]:
```
VARIABLE = 'LSTAT' #'RM'

x = pd.DataFrame(np.c_[boston[VARIABLE]], columns = [VARIABLE])
x.head()
y = boston['MEDV'].values.reshape((y.shape[0], 1))

x = np.concatenate([np.ones((x.shape[0], 1)), x], axis = 1)

m = x.shape[0]
n = x.shape[1]

print('# Training examples: ', m)
print('# Features : ', n)
```

```
# Training examples:  506
# Features :  2
```

We can now train the model using the gradient descent method:

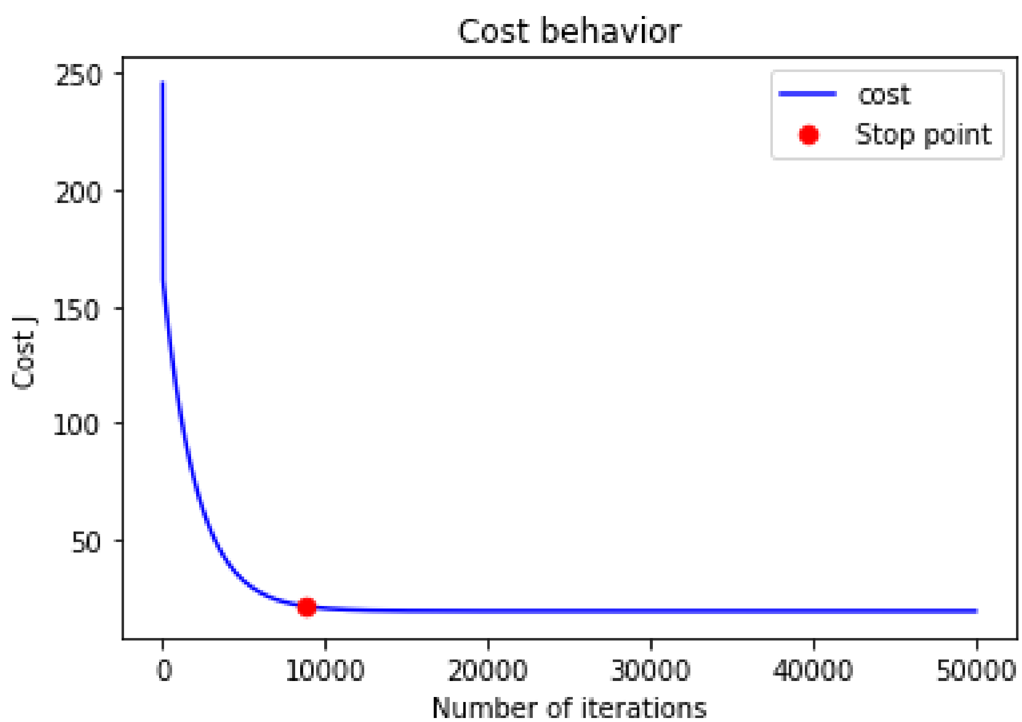
[11]:
```
theta = np.zeros((2,1))
num_iters = 50000
alpha = 0.001
```

```
theta, J_history = gradient_descent_vectorized(x, y, theta, alpha, num_iters)

stop_point = find_flat(J_history)

print(theta)
print("Early stop at step: {}".format(stop_point))
print("Cost at early stop: {}".format(J_history[stop_point]))
```

```
[[34.55363291]
 [-0.95003687]]
Early stop at step: 8806
Cost at early stop: 21.320640024786734
```

Let's show how the cost varies with respect to the iteration number

[12]:
```
plt.plot([i for i in range(num_iters)], J_history, 'b', label = 'cost')
if (stop_point != -1):
    #plt.axhline(y=J_history[stop_point], color='r', linestyle='-.')
    plt.plot(stop_point, J_history[stop_point], '.r', label = 'Stop point', ␣
 ↪markersize=13)
plt.xlabel('Number of iterations') # Set the xaxis label
plt.ylabel('Cost J') # Set the yaxis label
plt.title('Cost behavior')
plt.legend()
plt.show()
```



As it can be seen in the graph, the cost function decreases smoothly. The red point represents the point where the cost flattens out: after that point, the cost function decrease of less than *0.001* units at each iteration.

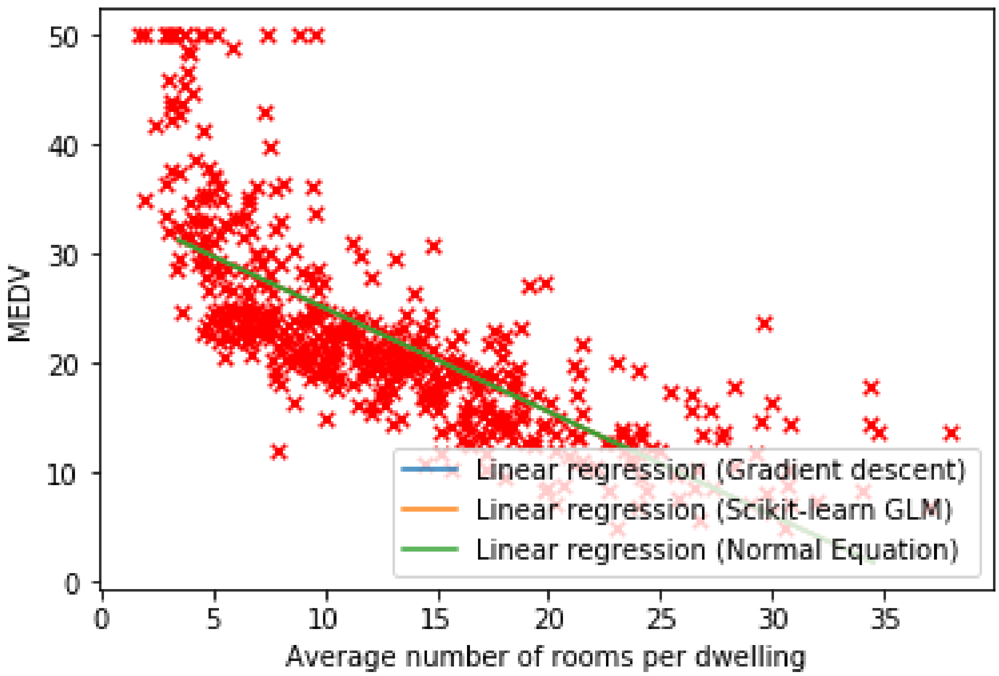
Let's calculate the θ parameter in closed form, using the **normal equation** method:

[13]:
```
theta_ne = normal_equations(x, y)
print("The parametes (using the normal equations) are:\n{}".format(theta_ne))
cost = compute_cost_vectorized(x, y, theta_ne)
```

```
The parametes (using the normal equations) are:
[[34.55384088]
 [-0.95004935]]
```

We can show the different parameters on the graph:

[14]:
```
xx = np.arange(3.5,35)
yy = theta[0] + theta[1] * xx

# Plot gradient descent
plt.scatter(x[:,1], y, s=30, c='r', marker='x', linewidths=1)
plt.plot(xx,yy, label='Linear regression (Gradient descent)')

# Compare with Scikit-learn Linear regression
regr = LinearRegression()
regr.fit(x[:,1].reshape(-1,1), y.ravel())
plt.plot(xx, regr.intercept_ + regr.coef_ * xx, label='Linear regression␣
 ↪(Scikit-learn GLM)')

# Compare with Normal Equations
plt.plot(xx, theta_ne[0] + theta_ne[1] * xx, label='Linear regression (Normal␣
 ↪Equation)')

#plt.xlim(-2,10)
plt.xlabel('Average number of rooms per dwelling')
plt.ylabel('MEDV')
plt.legend(loc=4);
```



The three lines in the graph coincide, and for this reason we can see only the green one.
Let's now calculate the error committed using the *root mean square error*:

[15]:
```
y_pred = np.zeros((x.shape[0], 1))
y_pred = x.dot(theta)

result = np.sqrt(mean_squared_error(y, y_pred))
print("The error done by the model is: {}".format(result))
```

```
The error done by the model is: 6.2034641322672694
```

2 Polinomial regression

We have seen simple models like the linear ones, but we can use use more complex models as the non-linear models.

[16]:
```
dataframe = pd.DataFrame(x[:, 1], columns = [VARIABLE])
new_data = polynomial_features(dataframe, 1)
new_data.head()
```

[16]:

	LSTAT
0	4.98
1	9.14
2	4.03
3	2.94
4	5.33

2.1 Comparing higher order hypothesis function

We'll use a polynomial hypothesis function, considering the n-th grade as:

$$h_\theta(x) = \theta_0 + \sum_{i=1}^{k} \theta_i x^i$$

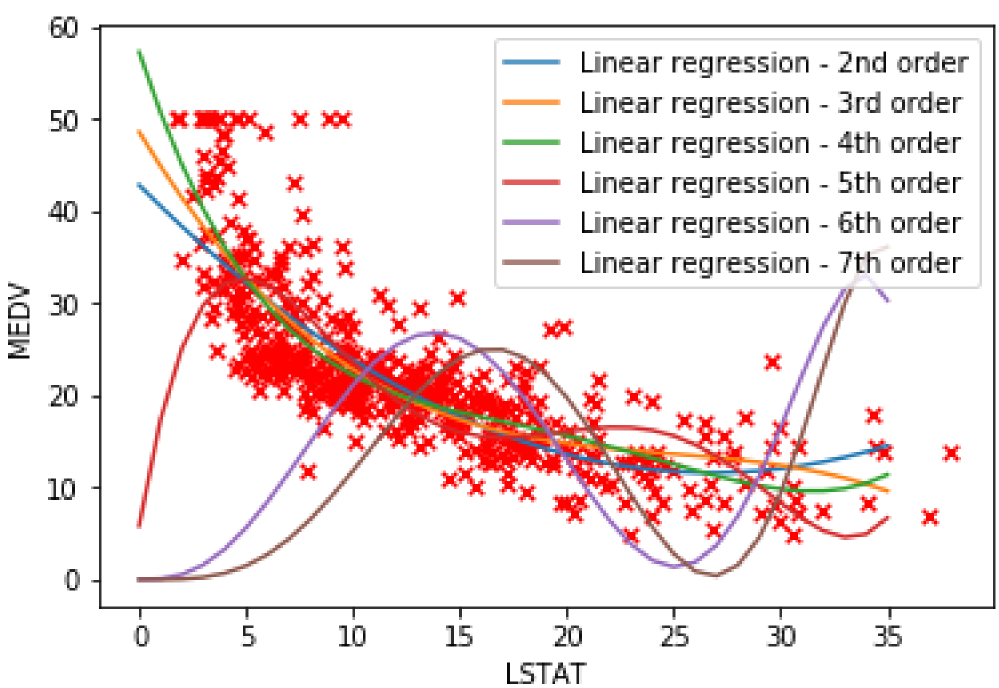
Below we will calculate the parameters for each model from the 2^{nd} to the 7^{th} degree.

[17]:
```
new_data = polynomial_features(dataframe, 2)
x_2 = np.concatenate([x, new_data.iloc[:,1].values.reshape((y.shape[0], 1))],␣
 ↪axis = 1)
theta_ne_2 = normal_equations(x_2, y)

new_data = polynomial_features(dataframe, 3)
x_3 = np.concatenate([x_2, new_data.iloc[:,2].values.reshape((y.shape[0], 1))],␣
 ↪axis = 1)
theta_ne_3 = normal_equations(x_3, y)

new_data = polynomial_features(dataframe, 4)
x_4 = np.concatenate([x_3, new_data.iloc[:,3].values.reshape((y.shape[0], 1))],␣
 ↪axis = 1)
theta_ne_4 = normal_equations(x_4, y)

new_data = polynomial_features(dataframe, 5)
x_5 = np.concatenate([x_4, new_data.iloc[:,4].values.reshape((y.shape[0], 1))],␣
 ↪axis = 1)
theta_ne_5 = normal_equations(x_5, y)

new_data = polynomial_features(dataframe, 6)
x_6 = np.concatenate([x_5, new_data.iloc[:,5].values.reshape((y.shape[0], 1))],␣
 ↪axis = 1)
theta_ne_6 = normal_equations(x_6, y)

new_data = polynomial_features(dataframe, 7)
x_7 = np.concatenate([x_6, new_data.iloc[:,6].values.reshape((y.shape[0], 1))],␣
 ↪axis = 1)
theta_ne_7 = normal_equations(x_7, y)
```

We can now fit the lines using the parameters just found, in order to draw them on a graph. In this way we can do a visual comparison of the different models.

[18]:
```
xx = np.arange(0,36)
yy_2 = theta_ne_2[0] + theta_ne_2[1] * xx + theta_ne_2[2] * xx**2
yy_3 = theta_ne_3[0] + theta_ne_3[1] * xx + theta_ne_3[2] * xx**2 +␣
 ↪theta_ne_3[3] * xx**3
yy_4 = theta_ne_4[0] + theta_ne_4[1] * xx + theta_ne_4[2] * xx**2 +␣
 ↪theta_ne_4[3] * xx**3 + theta_ne_4[4] * xx**4
yy_5 = theta_ne_5[0] + theta_ne_5[1] * xx + theta_ne_5[2] * xx**2 +␣
 ↪theta_ne_5[3] * xx**3 + theta_ne_5[4] * xx**4 + theta_ne_5[5] * xx**5
```

```
yy_6 = theta_ne_6[0] + theta_ne_6[1] * xx + theta_ne_6[2] * xx**2 +␣
→theta_ne_6[3] * xx**3 + theta_ne_6[4] * xx**4 + theta_ne_6[5] * xx**5 +␣
→theta_ne_6[6] * xx**6
yy_7 = theta_ne_7[0] + theta_ne_7[1] * xx + theta_ne_7[2] * xx**2 +␣
→theta_ne_7[3] * xx**3 + theta_ne_7[4] * xx**4 + theta_ne_7[5] * xx**5 +␣
→theta_ne_7[6] * xx**6 + theta_ne_7[7] * xx**7


# Plot gradient descent
plt.scatter(x[:,1], y, s=30, c='r', marker='x', linewidths=1)
plt.plot(xx,yy_2, label='Linear regression - 2nd order')
plt.plot(xx,yy_3, label='Linear regression - 3rd order')
plt.plot(xx,yy_4, label='Linear regression - 4th order')
plt.plot(xx,yy_5, label='Linear regression - 5th order')
plt.plot(xx,yy_6, label='Linear regression - 6th order')
plt.plot(xx,yy_7, label='Linear regression - 7th order')


#plt.ylim(-2,55)
#plt.xlim(-2,13)
plt.xlabel('LSTAT')
plt.ylabel('MEDV')
plt.legend(loc=1);
```



When we want to choose the model the right model, we need to do some comparison on the features in the training set and their relation with respect to the target features.

When we find the models that are correct from a conceptual point of view, we can choose the better among them by using a metrics like the *root mean squared error*.

2.2 Calculating root mean squared error and comparison

The definition of the **RMSE** is:

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{i=0}^{N}(\hat{y}_i - y_i)^2}{N}}$$

In order to calculate this: - we can iterate over the dataset taking the target feature; - do a prediction step using θ_{ne} found earlier - apply the formula for RMSE.

This will be done on the training set, even if it should be done on a test set for better comparisons.

Let's calculate the predictions.

[19]:
```
pred_2 = theta_ne_2[0] + theta_ne_2[1] * x[:,1] + theta_ne_2[2] * x[:,1]**2
pred_3 = theta_ne_3[0] + theta_ne_3[1] * x[:,1] + theta_ne_3[2] * x[:,1]**2 +␣
 ↪theta_ne_3[3] * x[:,1]**3
pred_4 = theta_ne_4[0] + theta_ne_4[1] * x[:,1] + theta_ne_4[2] * x[:,1]**2 +␣
 ↪theta_ne_4[3] * x[:,1]**3 + theta_ne_4[4] * x[:,1]**4
pred_5 = theta_ne_5[0] + theta_ne_5[1] * x[:,1] + theta_ne_5[2] * x[:,1]**2 +␣
 ↪theta_ne_5[3] * x[:,1]**3 + theta_ne_5[4] * x[:,1]**4 + theta_ne_5[5] * x[:
 ↪,1]**5
pred_6 = theta_ne_6[0] + theta_ne_6[1] * x[:,1] + theta_ne_6[2] * x[:,1]**2 +␣
 ↪theta_ne_6[3] * x[:,1]**3 + theta_ne_6[4] * x[:,1]**4 + theta_ne_6[5] * x[:
 ↪,1]**5 + theta_ne_6[6] * x[:,1]**6
pred_7 = theta_ne_7[0] + theta_ne_7[1] * x[:,1] + theta_ne_7[2] * x[:,1]**2 +␣
 ↪theta_ne_7[3] * x[:,1]**3 + theta_ne_7[4] * x[:,1]**4 + theta_ne_7[5] * x[:
 ↪,1]**5 + theta_ne_7[6] * x[:,1]**6 + theta_ne_7[7] * x[:,1]**7
```

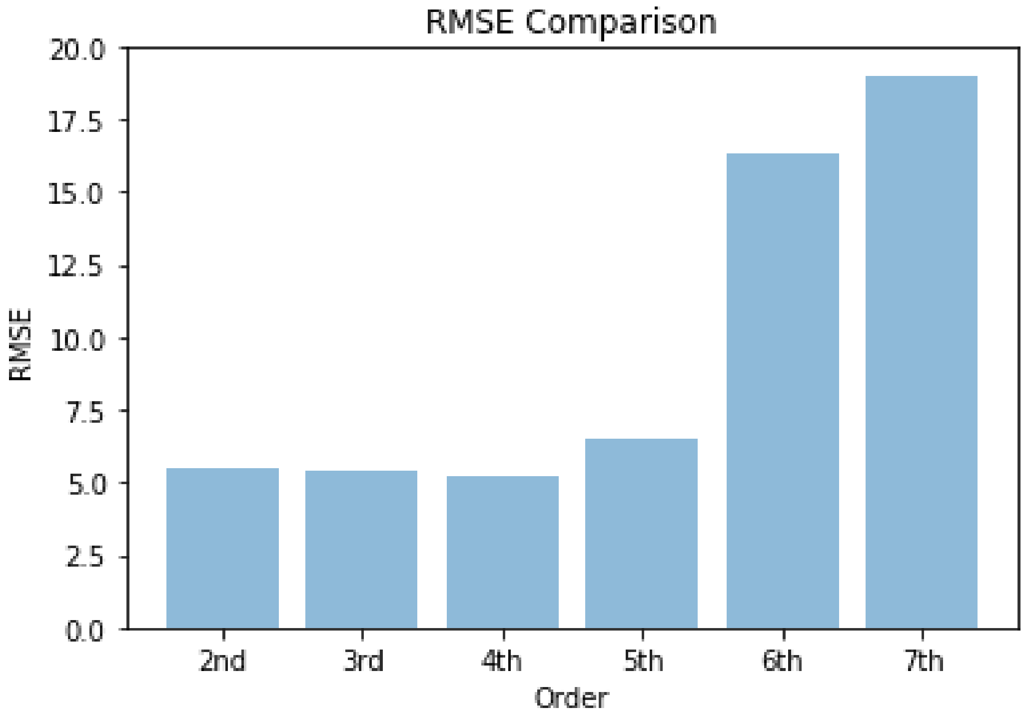
Let's calculate the RMSE metrics

[20]:
```
rmse_2 = np.sqrt(mean_squared_error(y, pred_2))
rmse_3 = np.sqrt(mean_squared_error(y, pred_3))
rmse_4 = np.sqrt(mean_squared_error(y, pred_4))
rmse_5 = np.sqrt(mean_squared_error(y, pred_5))
rmse_6 = np.sqrt(mean_squared_error(y, pred_6))
rmse_7 = np.sqrt(mean_squared_error(y, pred_7))
```

Let's now print a bar chart to see how the different degrees for the hypothesis work

[21]:
```
objects = ('2nd', '3rd', '4th', '5th', '6th', '7th')
y_pos = np.arange(len(objects))
performance = [rmse_2, rmse_3, rmse_4, rmse_5, rmse_6, rmse_7]

plt.bar(y_pos, performance, align='center', alpha=0.5)
plt.xticks(y_pos, objects)
plt.ylabel('RMSE')
plt.xlabel('Order')
```

```
plt.title('RMSE Comparison')

plt.show()
```



Here we can see that the lower cost comes with a polynomial of 4^{th} degree, basing our considerations only on the training set. The right thing to do would be to see wich polynomial hypothesis function works better on a validation dataset.

The problem here is that the model can overfit the data, giving us low error on the training set, but higher errors on test or validation set.

3 Let's take an interpolation of LSTAT and RM

The new hypothesis function will be:

$$h_\theta(x) = \theta_0 + \theta_1 x_{\mathrm{LSTAT}} + \theta_2 x_{\mathrm{RM}}$$

With this hypothesis function we will do a multivariate regression.

[22]:
```
VARIABLE_1 = 'LSTAT'
VARIABLE_2 = 'RM'

x = pd.DataFrame(np.c_[boston[VARIABLE_1], boston[VARIABLE_2]], columns =␣
↪[VARIABLE_1, VARIABLE_2])
x.head()
```

[22]:
```
   LSTAT     RM
0   4.98  6.575
1   9.14  6.421
2   4.03  7.185
3   2.94  6.998
4   5.33  7.147
```

[23]:
```
dataset_as_matrix = np.concatenate([np.ones((x.shape[0], 1)), x.LSTAT.values.
 ↪reshape((y.shape[0], 1)), x.RM.values.reshape((y.shape[0], 1))], axis = 1) #␣
 ↪concatenate ones for theta_0

theta_ne_int = normal_equations(dataset_as_matrix, y)

print(theta_ne_int)
```

```
[[-1.35827281]
 [-0.64235833]
 [ 5.09478798]]
```

4 Using validation and test datasets

We'll divide the dataset into two parts: - **Training set**, used to train the model (60% of the original dataset); - **Validation set**, used to test the trained model in order to get the right hyper-parameters and to see how the trained model performs with respect to the variation of these parameters (20% of the original dataset); - **Test set**, used to test the model and to get the performance metrics on unseed data (the remaing 20% of the orginal dataset).

[24]:
```
training_dimension = int(m * 0.6)
validation_dimension = int(m * 0.2)
test_dimension = m - training_dimension - validation_dimension

print("Dimension of the training set: {}".format(training_dimension))
print("Dimension of the validation set: {}".format(validation_dimension))
print("Dimension of the test set: {}".format(test_dimension))
```

```
Dimension of the training set: 303
Dimension of the validation set: 101
Dimension of the test set: 102
```

Let's show the first five example rows in the training set

[25]:
```
# Training set
df_training = boston.loc[0:training_dimension-1]
df_training.head()
```

[25]:
```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0  296.0
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
```

```
3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0   222.0
4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0   222.0

   PTRATIO       B  LSTAT  MEDV
0     15.3  396.90   4.98  24.0
1     17.8  396.90   9.14  21.6
2     17.8  392.83   4.03  34.7
3     18.7  394.63   2.94  33.4
4     18.7  396.90   5.33  36.2
```

Let's show the first five example rows in the validation set

[26]:
```
# Validation set
df_validation = boston.loc[training_dimension:
 ↪training_dimension+test_dimension-1]
df_validation.head()
```

[26]:
```
        CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
303  0.10000  34.0   6.09   0.0  0.433  6.982  17.7  5.4917  7.0  329.0
304  0.05515  33.0   2.18   0.0  0.472  7.236  41.1  4.0220  7.0  222.0
305  0.05479  33.0   2.18   0.0  0.472  6.616  58.1  3.3700  7.0  222.0
306  0.07503  33.0   2.18   0.0  0.472  7.420  71.9  3.0992  7.0  222.0
307  0.04932  33.0   2.18   0.0  0.472  6.849  70.3  3.1827  7.0  222.0

     PTRATIO       B  LSTAT  MEDV
303     16.1  390.43   4.86  33.1
304     18.4  393.68   6.93  36.1
305     18.4  393.36   8.93  28.4
306     18.4  396.90   6.47  33.4
307     18.4  396.90   7.53  28.2
```

Let's show the first five example rows in the test set

[27]:
```
# Test set
df_test = boston.loc[training_dimension+test_dimension: m-1]
df_test.head()
```

[27]:
```
         CRIM   ZN  INDUS  CHAS    NOX     RM    AGE     DIS   RAD    TAX  \
405  67.92080  0.0   18.1   0.0  0.693  5.683  100.0  1.4254  24.0  666.0
406  20.71620  0.0   18.1   0.0  0.659  4.138  100.0  1.1781  24.0  666.0
407  11.95110  0.0   18.1   0.0  0.659  5.608  100.0  1.2852  24.0  666.0
408   7.40389  0.0   18.1   0.0  0.597  5.617   97.9  1.4547  24.0  666.0
409  14.43830  0.0   18.1   0.0  0.597  6.852  100.0  1.4655  24.0  666.0

     PTRATIO       B  LSTAT  MEDV
405     20.2  384.97  22.98   5.0
406     20.2  370.22  23.34  11.9
407     20.2  332.09  12.13  27.9
408     20.2  314.64  26.40  17.2
409     20.2  179.36  19.78  27.5
```

4.1 Let's train the model using the training set

We'll train the model using a combination of two features, using the following hypothesis functions:

$$h_\theta(x) = \theta_0 + \theta_1 x_{\text{LSTAT}}\ x_{\text{RM}}^2$$

$$h_\theta(x) = \theta_0 + \theta_1 x_{\text{LSTAT}}^2\ x_{\text{RM}}$$

[28]:
```
TRAINING_VARIABLES = ['LSTAT', 'RM']

x_rm = pd.DataFrame(np.c_[df_training[TRAINING_VARIABLES]], columns =␣
 ↪TRAINING_VARIABLES)
x_lstat = pd.DataFrame(np.c_[df_training[TRAINING_VARIABLES]], columns =␣
 ↪TRAINING_VARIABLES)

y_training = df_training['MEDV'].values.reshape((df_training.shape[0], 1))

# Combine the columns obtaining a new column having the wanted features
x_rm['VAR'] = x_rm['LSTAT'] * x_rm['RM']**2
x_lstat['VAR'] = x_lstat['LSTAT']**2 * x_lstat['RM']
```

[29]:
```
# Dataset having RM squared
x_rm.head()
```

[29]:

	LSTAT	RM	VAR
0	4.98	6.575	215.288513
1	9.14	6.421	376.835263
2	4.03	7.185	208.045627
3	2.94	6.998	143.977692
4	5.33	7.147	272.254316

[30]:
```
# Dataset having LSTAT squared
x_lstat.head()
```

[30]:

	LSTAT	RM	VAR
0	4.98	6.575	163.062630
1	9.14	6.421	536.407772
2	4.03	7.185	116.690867
3	2.94	6.998	60.487913
4	5.33	7.147	203.038408

[31]:
```
# Discard the first two columns as we will use only the "VAR" column
adapted_training_set_rm =  pd.DataFrame(np.c_[x_rm["VAR"]], columns = ["VAR"])␣
 ↪# This training set has only the "VAR" column
x_rm = np.concatenate([np.ones((x_rm.shape[0], 1)), adapted_training_set_rm],␣
 ↪axis = 1) # Add a columns of 1

# Convert in dataframe and display it
dataframe_rm = pd.DataFrame(x_rm[:], columns = ["CONST", "VAR"])
dataframe_rm.head()
```

[31]:

	CONST	VAR
0	1.0	215.288513
1	1.0	376.835263
2	1.0	208.045627
3	1.0	143.977692
4	1.0	272.254316

[32]:
```
# Discard the first two columns as we will use only the "VAR" column
adapted_training_set_lstat =  pd.DataFrame(np.c_[x_lstat["VAR"]], columns =
 ↪["VAR"]) # This training set has only the "VAR" column
x_lstat = np.concatenate([np.ones((x_lstat.shape[0], 1)),
 ↪adapted_training_set_lstat], axis = 1) # Add a columns of 1

# Convert in dataframe and display it
dataframe_lstat = pd.DataFrame(x_lstat[:], columns = ["CONST", "VAR"])
dataframe_lstat.head()
```

[32]:

	CONST	VAR
0	1.0	163.062630
1	1.0	536.407772
2	1.0	116.690867
3	1.0	60.487913
4	1.0	203.038408

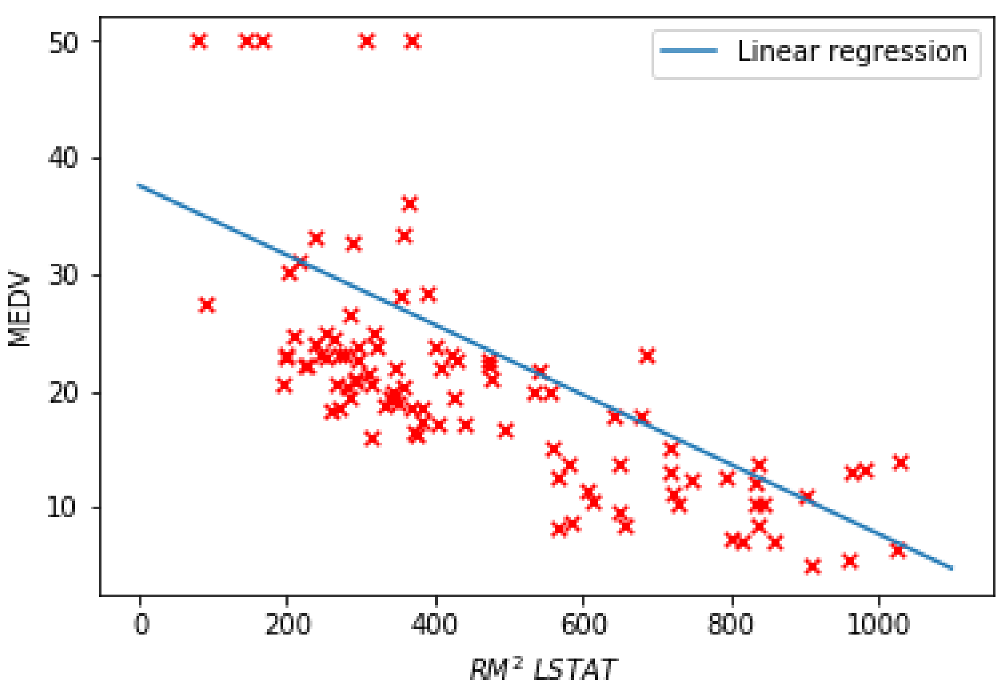
[33]:
```
# Let's train the model and get the cost
x_rm = np.concatenate([dataframe_rm], axis = 1)
x_lstat = np.concatenate([dataframe_lstat], axis = 1)

theta_ne_rm = normal_equations(x_rm, y_training)
cost_rm = compute_cost_vectorized(x_rm, y_training, theta_ne_rm)
print("The trained model having RM squared has parameters:\n{}\nIt has a cost
 ↪of {}\n\n".format(theta_ne_rm, cost_rm))

theta_ne_lstat = normal_equations(x_lstat, y_training)
cost_lstat = compute_cost_vectorized(x_lstat, y_training, theta_ne_lstat)
print("The trained model having LSTAT squared has parameters:\n{}\nIt has a
 ↪cost of {}".format(theta_ne_lstat, cost_lstat))
```

```
The trained model having RM squared has parameters:
[[ 3.76188776e+01]
 [-2.98815284e-02]]
It has a cost of [[14.84208928]]


The trained model having LSTAT squared has parameters:
[[ 2.99835436e+01]
 [-5.00082409e-03]]
It has a cost of [[15.71867252]]
```

4.2 Let's use the validation set to see how the models perform

[34]:
```
# Let's adapt the validation set
x = pd.DataFrame(np.c_[df_validation[TRAINING_VARIABLES]], columns =␣
 ↪TRAINING_VARIABLES)
y_validation = df_validation['MEDV'].values.reshape((df_validation.shape[0], 1))

x = np.concatenate([np.ones((x.shape[0], 1)), x], axis = 1) # Add a columns of 1
```

[35]:
```
xx = np.arange(0,1100)
yy = theta_ne_rm[0] + theta_ne_rm[1] * xx

# Plot gradient descent
plt.scatter(x[:,1] * x[:,2]**2, y_validation, s=30, c='r', marker='x',␣
 ↪linewidths=1)
plt.plot(xx,yy, label='Linear regression')

plt.xlabel('$RM^2\ LSTAT$')
plt.ylabel('MEDV')
plt.legend(loc=1);
```

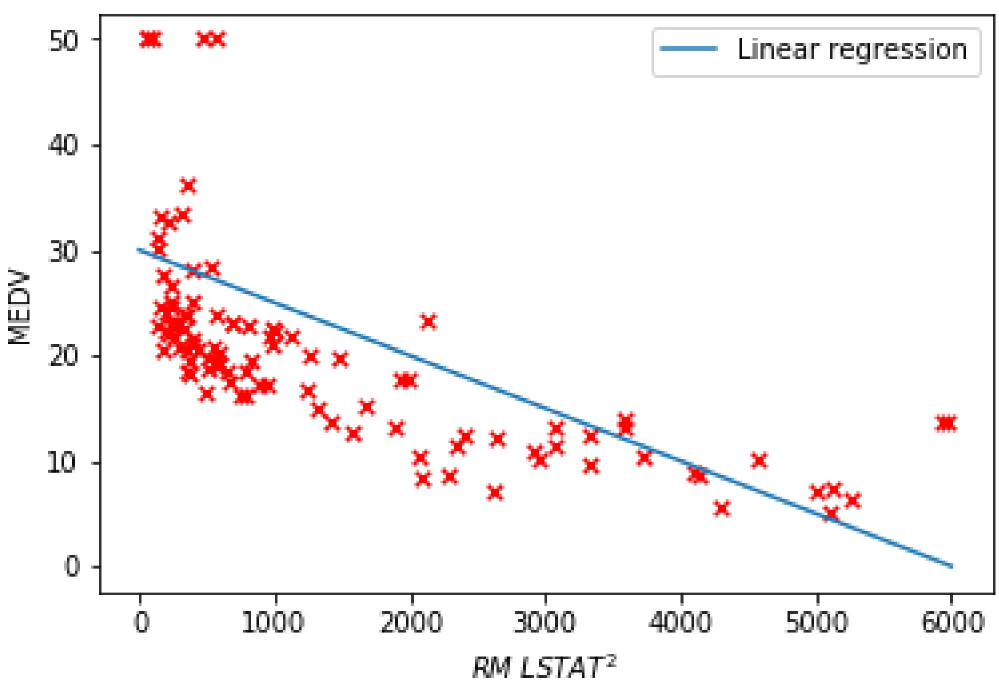


[36]:
```
# Let's see the RMSE for the validation set
predictions = theta_ne_rm[0] + theta_ne_rm[1] * x[:,1] * x[:,2]**2
rmse_rm = np.sqrt(mean_squared_error(y_validation, predictions))
```

```
print("The RMSE value for the validation set using RM squared is: {}".
 ↪format(rmse_rm))
```

```
The RMSE value for the validation set using RM squared is: 7.620990352107873
```

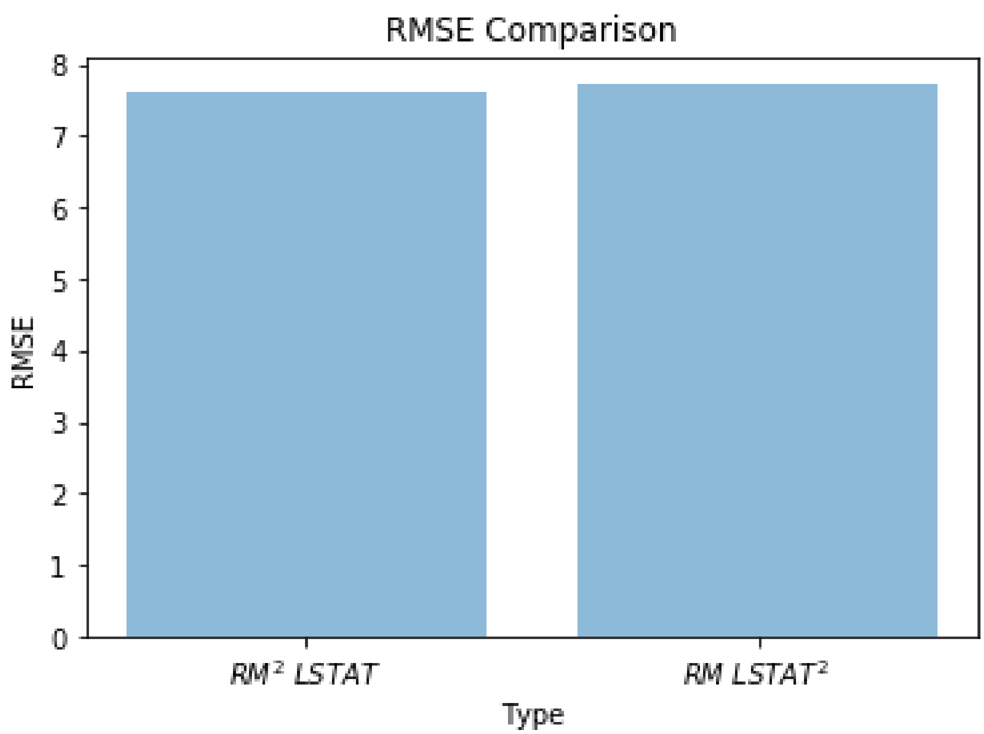
[37]:
```
xx = np.arange(0,6000)
yy = theta_ne_lstat[0] + theta_ne_lstat[1] * xx

# Plot gradient descent
plt.scatter(x[:,1]**2 * x[:,2], y_validation, s=30, c='r', marker='x',␣
 ↪linewidths=1)
plt.plot(xx,yy, label='Linear regression')

plt.xlabel('$RM\ LSTAT^2$')
plt.ylabel('MEDV')
plt.legend(loc=1);
```



[38]:
```
# Let's see the RMSE for the validation set
predictions = theta_ne_lstat[0] + theta_ne_lstat[1] * x[:,1]**2 * x[:,2]
rmse_lstat = np.sqrt(mean_squared_error(y_validation, predictions))

print("The RMSE value for the validation set using LSTAT squared is: {}".
 ↪format(rmse_lstat))
```

```
The RMSE value for the validation set using LSTAT squared is: 7.730818593681602
```

[39]:
```
objects = ('$RM^2\ LSTAT$', '$RM\ LSTAT^2$')
y_pos = np.arange(len(objects))
performance = [rmse_rm, rmse_lstat]

plt.bar(y_pos, performance, align='center', alpha=0.5)
plt.xticks(y_pos, objects)
plt.ylabel('RMSE')
plt.xlabel('Type')
plt.title('RMSE Comparison')

plt.show()
```



Using the two values obtained for the root mean squared error, we can see that the model implementing the hypothesis function

$$h_\theta(x) = \theta_0 + \theta_1 x_{\text{LSTAT}}\ x_{\text{RM}}^2$$

works better, having a lower RMSE value. The validation set is used for doing these types of choices and reasonings.

It is for this reason that the chosen model used for testing purposes will be the one implementing the formula written above.

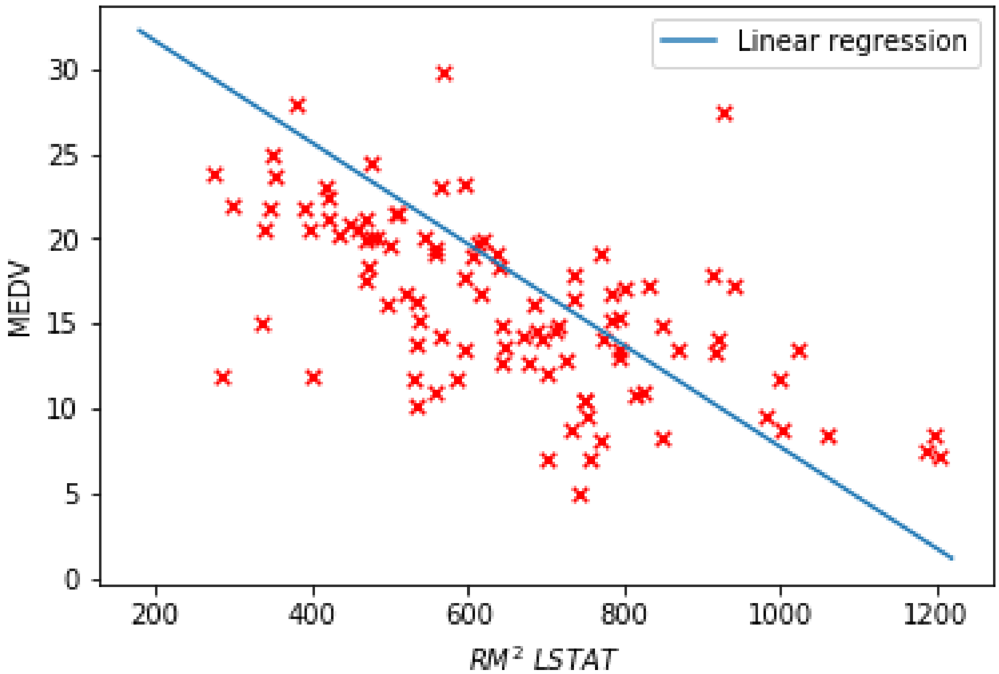
4.3 Let's use the test set to see how the model performs

[40]:
```
# Let's adapt the test set
x = pd.DataFrame(np.c_[df_test[TRAINING_VARIABLES]], columns =␣
 ↪TRAINING_VARIABLES)
y = df_test['MEDV'].values.reshape((df_test.shape[0], 1))

x = np.concatenate([np.ones((x.shape[0], 1)), x], axis = 1) # Add a columns of 1

# Convert in dataframe and display it
dataframe = pd.DataFrame(x[:], columns = ["CONST", "LSTAT", "RM"])
dataframe.head()
```

[40]:

	CONST	LSTAT	RM
0	1.0	22.98	5.683
1	1.0	23.34	4.138
2	1.0	12.13	5.608
3	1.0	26.40	5.617
4	1.0	19.78	6.852

[41]:
```
x = np.concatenate([dataframe], axis = 1)

xx = np.arange(180,1220)
yy = theta_ne_rm[0] + theta_ne_rm[1] * xx

# Plot gradient descent
plt.scatter(x[:,1] * x[:,2]**2, y, s=30, c='r', marker='x', linewidths=1)
plt.plot(xx,yy, label='Linear regression')

#plt.ylim(-2,55)
#plt.xlim(-2,13)
plt.xlabel('$RM^2\ LSTAT$')
plt.ylabel('MEDV')
plt.legend(loc=1);
```



[43]:
```
# Let's see the RMSE for the validation set
predictions = theta_ne_rm[0] + theta_ne_rm[1] * x[:,1] * x[:,2]**2
rmse = np.sqrt(mean_squared_error(y, predictions))

print("The RMSE value for the test set is: {}".format(rmse))
```

```
The RMSE value for the test set is: 5.591531084890972
```

[ ]:
```

```