Regularized Polynomial Regression

Capoccia Leonardo, Basile Davide, Corvaglia Salvatore

June 12, 2019

1 Regularized Polynomial Regression

The **regularization technique** can prevent our model from overfitting the training data. The **overfitting** is a problem that occurs when we use a model which is too complex for the given dataset. This method gives us a very small loss on the training set, but it fails to generalize on unseen data, giving us wrong predictions. An opposite problem is the **underfitting**, which is a problem that occurs when we use a model which is too simple to explain the data or is not trained enough.

To prevent the *overfitting* problem, we can add a regularization term to the cost function, which depends on a **hyper-parameter** that weights a **regularization term**. The regularization term imposes a penalty on the complexity of the cost function. The regularization term is as follows:

$$\lambda \sum_{j=1}^{n} \theta_j^2$$

where λ is the hyper-parameter and the sum represents the penalty function.

Let's start this exercise by introducing some libraries, loading our Boston dataset and introducing the standard methods to apply the gradient descent and the normal equations without the regularization term.

```
In [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

        np.random.seed(10)

        boston_dataset = load_boston()
```

```
In [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):
            m = x.shape[0]
            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
        def feature_normalize(x):
            x_norm = x

            mu = np.zeros((1, x.shape[1]))
            sigma = np.zeros((1, x.shape[1]))

            mu = np.mean(x, axis = 0) # mean value
            sigma = np.std(x, axis = 0) # std deviation value

            for i in range(x.shape[1]):
                x_norm[:,i] = (x[:,i] - mu[i])/sigma[i]

            return x_norm, mu, sigma
```

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

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

The Boston dataset contains the prices of some houses in Boston along with some features. Let's see the meaning of the features:

```
In [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:

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

Out[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

It's important to note that in the training set shown, there isn't a price, which represents our target feature. Let's add it:

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

Out[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

For each feature, let's see if there are some spurious values (for example a `null` or `NaN` value):

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

```
Out[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
```

Now that we have seen that there is not any invalid value, we can go on with our experiment.

We can now create a DataFrame based on our dataset, so we can shuffle it efficiently. Hence, we can get a random permutation of the examples.

Using this approach, we can lower the chance that two any subsequent examples in the training set are correlated.

After the shuffle of the DataFrame, we will create three different datasets from our Boston dataset:

- The first one is the **training set**, with the 60% of the whole dataset;
- The second is the **validation set**. It contains the 20% of the Boston dataset;
- The last one is the **test set**, which contains the remaing 20% of the Boston dataset.

After the creation of these datasets, we will divide them into two different portions:

- `X_train`, `X_val`, `X_test`: they contain all the features used to make predictions for each dataset;
- `y_train`, `y_val`, `y_test`: they contains the target features for each portion of the dataset.

After the creation of these sets, we will add a column of 1 to the `X_train`, `X_val`, `X_test` so we can deal with the θ_0 parameter.

```
In [8]: VARIABLE = 'LSTAT'

        x = boston[VARIABLE].values.reshape((boston[VARIABLE].shape[0], 1)) #(506, 1)
        y = boston['MEDV'].values.reshape((x.shape[0], 1)) #(506, 1)
```

```
df = pd.DataFrame(np.concatenate([x, y], axis=1))

# Let's take the training set as the first 60% of the dataset,
# the validation set as the example between the 60% and to 80%,
# the test set will be the remmaining part of the dataset (20%)
train, val, test = np.split(df.sample(frac=1), [int(.6 * len(df)), int(.8*len(df))])

# Let's decompose between prediction features and target feature
X_train, y_train = train[0], train[1]
X_val, y_val = val[0], val[1]
X_test, y_test = test[0], test[1]

# Let's do a reshape to create the second dimension
X_train = np.reshape([X_train], (train.shape[0], 1))
y_train = np.reshape([y_train], (train.shape[0], 1))
X_val = np.reshape([X_val], (val.shape[0], 1))
y_val = np.reshape([y_val], (val.shape[0], 1))
X_test = np.reshape([X_test], (test.shape[0], 1))
y_test = np.reshape([y_test], (test.shape[0], 1))

# Let's add the columns of 1 in order to deal with theta_0
X_train = np.c_[np.ones((X_train.shape[0], 1)), X_train]
X_val = np.c_[np.ones((X_val.shape[0], 1)), X_val]
X_test = np.c_[np.ones((X_test.shape[0], 1)), X_test]
```

2 Linear regression with regularization

When using the regularization, the cost function represents a tradeoff between the classical cost function and a term that imposes to the parameters not to grow. This behavior is reached using a regularization term, which is a penalty function on the parameters' values.

It is important to note that the regularization is not applied to the parameter θ_0 as it is a value used to weight the x_0 feature, which is 1 by convention.

The functions implemented in the following code are:

Regularized cost function

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

It is important to note that the regularization term doesn't include the θ_0 parameter.

Gradient of the regularized

$$\frac{\partial J(\theta)}{\partial \theta} = \left[\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x^{(i)}) - y^{(i)}\big)x^{(i)} + \frac{\lambda}{m}\theta_r\right]$$

with:

$$\theta_r = \begin{bmatrix} 0 \\ \theta_1 \\ \dots \\ \theta_n \end{bmatrix}$$

We use θ_r because we have chosen not to regularize on the first parameter of the parameter vector (θ_0).

Gradient descent with regularization

The update rule to apply on each iteration is:

$$\theta_0 = \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x^{(i)}) - y(i)\big)x_0^{(i)}$$

$$\theta_j = \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x^{(i)}) - y(i)\big)x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] \text{ , for j=1,...,n}$$

Here is shown with greater evidence that **the θ_0 parameter is not regularized**.
The second equation can be rewritten as:

$$\theta_j = \theta_j\big(1 - \alpha\frac{\lambda}{m}\big) - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x^{(i)}) - y(i)\big)x_j^{(i)} \text{ , for j=1,...,n}$$

The θ_j multiplicative factor is usually a value smaller than 1. Consequently, we can state that the regularization term is just a way to shrink the parameter vector at each iteration.

Normal equation with regularization

We can use an analytical method to find the values that minimize the cost function called **Normal Equations**. The regularized normal equations are:

$$\theta = \Big(X^TX + \lambda\begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}\Big)X^Ty$$

where the matrix is a $(n+1)\times(n+1)$ matrix. This approach has some advantages over the gradient descent method:

- there is no need to do the feature scaling in case of features values with different scales (this always happen when dealing with polynomial regression),

- there is no need to choose the learning rate hyper-parameter needed to the gradient descent algorithm because there are no iterations at all.

The main disadvantage is the need to calculate the inverse of a matrix, which is very resource consuming and computation intensive.

```
In [9]: def compute_cost_reg(theta, x, y, lamda = 0):
            h = x.dot(theta)
            m = x.shape[0]
            J = np.sum((h-y)**2) / (2 * m)
            theta_1 = theta
            theta_1[0] = 0
            J = J + lamda / (2 * m) * np.sum(theta_1 ** 2)
            return J

        def gradient_reg(theta, x, y, lamda = 0):
            m = x.shape[0]
            theta_r = theta
            theta_r[0] = 0
            h = x.dot(theta)
            grad = (x.T.dot(h-y) / m) + (lamda/m)*theta_r
            return grad.flatten()

        def gradient_descent_reg(theta, X, y, _lambda=0, alpha = 0.001,
                num_iters = 20000, epsilon = 0.0001):
            for k in range(num_iters):
                grad = gradient_reg(theta, X, y, _lambda).reshape((theta.shape[0],1))
                theta = theta - np.multiply(alpha, grad)
            return theta

        def normal_equation_reg(x, y, lamda):
            matrix_for_regularization = np.identity(x.shape[1])
            matrix_for_regularization[0][0] = 0
            inverse = np.linalg.pinv(np.add(x.T.dot(x), np.multiply(lamda,
                            matrix_for_regularization)))
            return (inverse.dot(x.T)).dot(y)
```

3 Part 1. Comparing different degrees of hypothesis

In this part of the exercise, we will compare different degrees of hypothesis function. To choose the best hypothesis function, we will calculate the parameters vector θ using the non-regularized cost function. We will complete two steps:

1. Calculate the parameter vector using a given degree of the hypothesis function on the training set and determine the loss on the same set;

2. Calculate the cost function on the validation set using the θ value calculated at the previous step.

We will draw a graph of the costs calculated to facilitate the analysis.
Let's start by training the models, starting from a degree of 1 to degree 26:

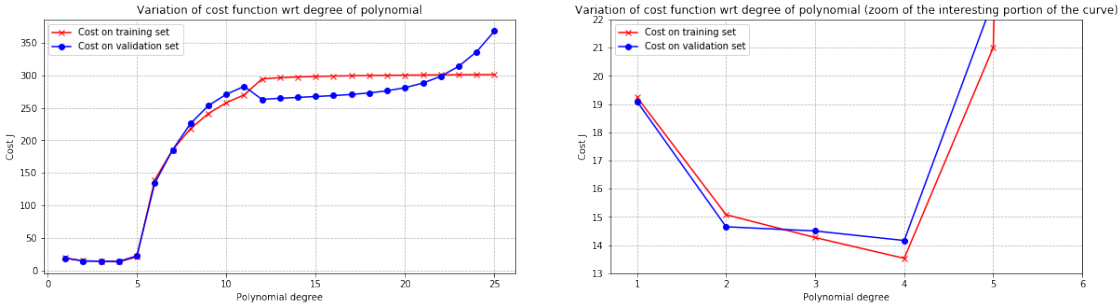
In [10]:
```
dataframe = pd.DataFrame(X_train[:,1], columns=[VARIABLE])
dataframe_val = pd.DataFrame(X_val[:,1], columns=[VARIABLE])

costs_train, costs_val = [], []
_thetas_train = []
for degree in range(1, 26):

    # Calculate the features for the training and the validation set
    new_data = polynomial_features(dataframe, degree)
    new_data_val = polynomial_features(dataframe_val, degree)
    _x_train = np.concatenate([np.ones((dataframe.shape[0], 1)), new_data], axis=1)
    _x_val = np.concatenate([
           np.ones((dataframe_val.shape[0],1)), new_data_val], axis=1)

    # Train the model
    theta = normal_equations(_x_train, y_train)

    # Calculate the cost on the training set and save it
    _cost_train = compute_cost_vectorized(_x_train, y_train, theta)
    costs_train.append(_cost_train[0][0])

    # Calculate the cost on the validation set and save it
    _cost_val = compute_cost_vectorized(_x_val, y_val, theta)
    costs_val.append(_cost_val[0][0])
```

We can now plot the costs calculated in the previous step:

In [11]:
```
# Plot gradient descent
plt.figure(figsize=(20, 5))

# Create stuff to show
labels = np.arange(1, 26)

# First subplot (left plot)
plt.subplot(1, 2 , 1)
plt.plot(labels, costs_train, '-x', c='r', label='Cost on training set')
plt.plot(labels, costs_val, '-o', c='b', label='Cost on validation set')
plt.grid(linestyle='--', linewidth=.7)

plt.title('Variation of cost function wrt degree of polynomial')
plt.xlabel('Polynomial degree')
plt.ylabel('Cost J')
plt.legend(loc=2);
```

```
# Second subplot
plt.subplot(1, 2 , 2)
plt.plot(labels, costs_train, '-x', c='r', label='Cost on training set')
plt.plot(labels, costs_val, '-o', c='b', label='Cost on validation set')
plt.grid(linestyle='--', linewidth=.7)

plt.title('Variation of cost function wrt degree of polynomial
 (zoom of the interesting portion of the curve)')
plt.xlabel('Polynomial degree')
plt.ylabel('Cost J')
plt.legend(loc=2)

plt.xlim(0.7,6)
plt.ylim(13,22)
```

Out[11]: (13, 22)



This graph shows that for a fourth-degree hypothesis function, the cost of the training set and the one on the validation set is the lowest, which means that the correct choice for the hypothesis is that one.

On the other hand, the graph shows that for the lower order models, the two costs are higher than the previous one, which means that we have a problem with **high bias**, which means we are **underfitting** the model. For the higher degrees, the cost calculated on the training set is almost the same, while the one on the validation set becomes higher and higher: this is a situation where we're **overfitting** the model, which means that we have a problem with **high variance**.

4 Part 2. Choice of the regularization parameter

In this part of the experiment, we will use the hypothesis function as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

We can choose the hypothesis function by doing some reasoning as in the previous section. In this section, we will try to find the best value of regularization parameter λ to avoid the overfitting on the training set.

To do so, we will apply the **regularized normal equations** using several values of the λ parameter and choosing the one which performs better.

Let's create an array with all the λ parameter values. We will start from a value of 0.01, and we will double it up until we encounter a threshold value. In this example, the threshold is 100.

```
In [12]: selected_degree = 4

         lamdas = [0, .01]
         lamda = .01
         while lamda <= 100:
             lamda = lamda*2
             lamdas.append(lamda)
         print("We'll try these values of lambda to see how the cost function behaves:\n{}"
            .format(lamdas))
```

```
We'll try these values of lambda to see how the cost function behaves:
[0, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.28, 2.56, 5.12, 10.24, 20.48,
40.96, 81.92, 163.84]
```

We can now create the training and the validation set upon wich we can operate:

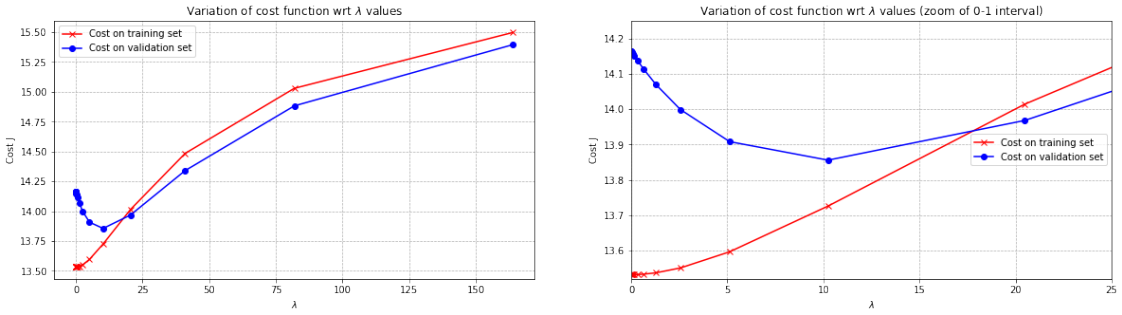
```
In [13]: dataframe = pd.DataFrame(X_train[:,1], columns=[VARIABLE])
         new_data = polynomial_features(dataframe, selected_degree)
         x_train_part_2 = np.concatenate([np.ones((dataframe.shape[0], 1)), new_data], axis=1)

         dataframe = pd.DataFrame(X_val[:,1], columns=[VARIABLE])
         new_data = polynomial_features(dataframe, selected_degree)
         x_val_part_2 = np.concatenate([np.ones((dataframe.shape[0], 1)), new_data], axis=1)
```

Let's test on both datasets the different values of the regularization parameter and let's keep track of the cost function:

```
In [14]: costs_train, costs_val = [], []

         # Iterate over all lambda values
         for lamda in lamdas:

             # Train the model using a value of lambda at a time
             _theta_train = normal_equation_reg(x_train_part_2, y_train, lamda)

             # Calculate the cost on the training set and save it
             _cost_train = compute_cost_vectorized(x_train_part_2, y_train, _theta_train)
             costs_train.append(_cost_train[0][0])

             # Calculate the cost on the validation set and save it
             _cost_val = compute_cost_vectorized(x_val_part_2, y_val, _theta_train)
             costs_val.append(_cost_val[0][0])
```

We can now plot the costs on the training set and the validation set with respect to the λ values:

```
In [15]: # Plot gradient descent
         plt.figure(figsize=(20, 5))

         plt.subplot(1, 2, 1)
         plt.plot(lamdas, costs_train, '-x', c='r', label='Cost on training set')
         plt.plot(lamdas, costs_val, '-o', c='b', label='Cost on validation set')

         plt.title('Variation of cost function wrt $\lambda$ values')
         plt.xlabel('$\lambda$')
         plt.ylabel('Cost J')
         plt.legend(loc=2);
         plt.grid(linestyle='--', linewidth=.7)

         plt.subplot(1, 2, 2)
         plt.plot(lamdas, costs_train, '-x', c='r', label='Cost on training set')
         plt.plot(lamdas, costs_val, '-o', c='b', label='Cost on validation set')

         plt.title('Variation of cost function wrt $\lambda$ values (zoom of 0-1 interval)')
         plt.xlabel('$\lambda$')
         plt.ylabel('Cost J')
         plt.legend(loc=5)
         plt.xlim(-.02, 25)
         plt.ylim(13.52, 14.25)
         plt.grid(linestyle='--', linewidth=.7, which='both')
```



The best choice of the regularization parameter is the choice which minimizes the cost on the validation set. The best value is $\lambda = 10.24$.

In the left graph, we can see two behaviors:

- Using a small regularization term can lower the cost on the training set while increasing the cost on the validation set. In this case, we can assert that we have a problem with **high variance**, which is synonymous of **overfitting**;
- Using a bigger regularization term can increase the costs on the training set and on the validation set, making them comparable. In this case, we can assert that we have a problem with **high bias**, which is synonymous of **underfitting**;

5 Part 3. Diagnosing problems with the learning curves

By now, we have chosen the degree of the hypothesis function and the best value possible for the regularization parameter. The last step in this experiment is to diagnose a problem with **high variance** or **high bias** based on the **learning curves**, which are a way to show how the cost functions go by varying the number of training examples used to train the model.

We will start by taking a training example to estimate the parameter θ using the normal equations. We will use this value to calculate the costs on the training and on the validation set. Hence, we will increment the number of training examples by 1, repeating the same procedure until the number of training examples will represent the whole training set.

Let's start by doing so:

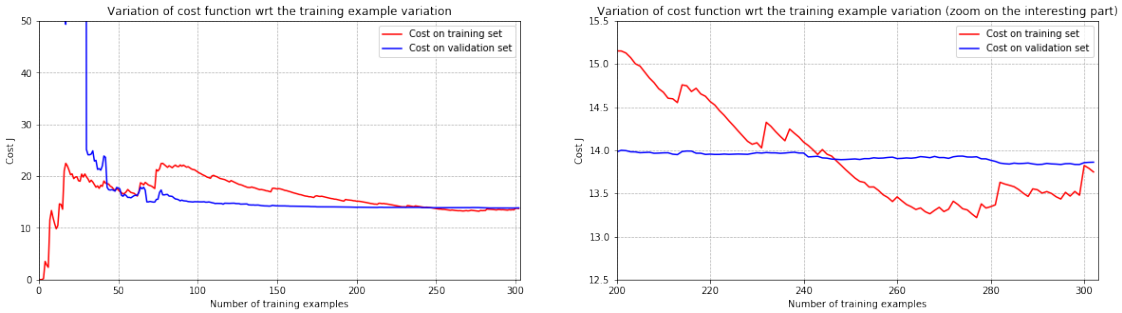
In [16]:
```
lamda = 10.24
costs_train, costs_val, training_examples = [], [], []

for num_examples in range(1, X_train.shape[0]):

    # Keep track of the number of training examples used to train the model
    training_examples.append(num_examples)

    # Let's take the first part of the training set
    x_train = X_train[:num_examples]

    # Create the dataset used for the training phase
    dataframe = pd.DataFrame(x_train[:,1], columns=[VARIABLE])
    new_data = polynomial_features(dataframe, selected_degree)
    x_train = np.concatenate([np.ones((dataframe.shape[0], 1)), new_data], axis=1)

    # Let's train the model
    _theta = normal_equation_reg(x_train, y_train[:num_examples], lamda)

    # Calculate the cost on the training set and save it
    _cost_train = compute_cost_vectorized(x_train, y_train[:num_examples], _theta)
    costs_train.append(_cost_train[0][0])

    # Calculate the cost on the validation set and save the value into an array
    _cost_val = compute_cost_vectorized(x_val_part_2, y_val, _theta)
    costs_val.append(_cost_val[0][0])
```

Let's plot the costs when the number of training examples used in the training phase varies:

In [17]:
```
# Plot gradient descent
plt.figure(figsize=(20, 5))

plt.subplot(1, 2, 1)
plt.plot(training_examples, costs_train, '-', c='r', label='Cost on training set')
plt.plot(training_examples, costs_val, '-', c='b', label='Cost on validation set')

plt.title('Variation of cost function wrt the training example variation')
```

```
plt.xlabel('Number of training examples')
plt.ylabel('Cost J')
plt.legend(loc=1);
plt.grid(linestyle='--', linewidth=.7)

plt.xlim(0, 303)
plt.ylim(0,50)


plt.subplot(1, 2, 2)
plt.plot(training_examples, costs_train, '-', c='r', label='Cost on training set')
plt.plot(training_examples, costs_val, '-', c='b', label='Cost on validation set')

plt.title('Variation of cost function wrt the training example variation (zoom on the i
plt.xlabel('Number of training examples')
plt.ylabel('Cost J')
plt.legend(loc=1);
plt.grid(linestyle='--', linewidth=.7)

plt.xlim(200, 303)
plt.ylim(12.5,15.5)
```

Out[17]: (12.5, 15.5)



The graphs above show how the cost functions behave when the number of training examples varies.

Initially, when the number of training examples is small, the cost calculated on the training set is very low because it's easy to overfit the data in such a case, while the cost on the validation set is very high since the model overfits the training set and doesn't generalize on unseen data.

When the number of training examples increases, the model fits the data. Hence, the cost calculated on the training set increases while the cost calculated on the validation set lowers. This happens because the model cannot overfit the data when using the regularization term, so it is more efficient in generalizing on unseen events.

After an exact point, even if the number of training examples increases, the value of the cost function doesn't decrease, which means that adding new training examples doesn't improve the model. This event is usually related to a problem with **high bias**, but this is not the case since

the value of the cost function is low. Another possibility would be to have the two curves to be one far from the other with a gap between them. In this case, adding new training examples would be preferred since the cost function on the validation set would decrease: this would be a clear indication of a problem with **high variance**.