**Pattern Recognition And Machine Learning - EPFL - Fall 2015**
Emtiyaz Khan, Timur Bagautdinov, Carlos Becker, Ilija Bogunovic & Ksenia Konyushkova

# 3. Least Squares, Ridge Regression, and Overfitting

## 3.1 Goals

The goal of this exercise is to

- Implement and debug least-squares.
- Implement, debug and visualize basis function models.
- Understand overfitting.
- Implement Ridge regression.

## 3.2 Data and sample code

On the course website, you will find an archive `ex3code.zip`, which contains Matlab scripts and functions that you will need to finish this exercise. It also contains the dataset for this exercise. Download this file and extract it. Put it in the path of Matlab (if you don't know how to do this, refer to the Matlab guide from exercise session 1).

## 3.3 Least squares

**Exercise 3.1** Implementing and debugging least squares.

- Implement the following function (in a new file `leastSquares.m`) which implements the solution of the normal equations as discussed in the class.
  ```
  beta = leastSquares(y,tX)
  ```
  If you do not understand what `y` and `tX` are, refer to the previous exercise. Use the sample code shown in the lecture notes.
- To debug your code, you can use the output of the last exercise. Run gradient descent or grid search on the height-weight data from the last exercise, and make sure you get same value of $\boldsymbol{\beta}$ using both methods.
- Try various versions of least-square implementations given in the lecture notes. They must all give the same answer.

■

This is a useful method to debug your code, i.e. first implementing a simple method and then using it to check more complicated methods. If you have not finished exercise 2, please first finish implementing grid search method. If you are lagging behind, do not worry. You will get time later to catch up, but it is important that you finish previous exercises.

## 3.4 Least squares and basis function models

We will now implement and visualize a basis function model for the data `dataEx3.mat`.

As explained in the class, linear regression might not be suitable for nonlinear data. We will use polynomial basis functions to fit nonlinear data:

$$\phi_j(x) = x^j \tag{3.1}$$

Revise lecture notes. We will use different degree of polynomials, e.g. a two degree polynomial with $x$ and $x^2$, a three degree polynomial with $x$,$x^2$ and $x^3$, etc. The higher degree polynomials are more expressive and can capture fine details in the data. Is that a good thing? Think about it.

To check the fit, we will use a measure called the Root-Mean-Square-Error (RMSE). It is related to MSE as follows:

$$\text{RMSE}(\boldsymbol{\beta}) = \sqrt{2 * \text{MSE}(\boldsymbol{\beta})} \tag{3.2}$$

MSE is difficult to interpret since it involves a square, therefore RMSE is a more interpretable measure. There are better measures like $R^2$ but you can learn about them from the book "Introduction to Statistical learning".

Let us now implement polynomial regressions and visualize their predictions.

**Exercise 3.2** Implementing and visualizing polynomial regression.

- Run the script `visualize.m`. This script plots the data along with predictions using polynomial regression. In the provided script, the value of $\boldsymbol{\beta}$ is set to 0. Your goal is to find a good $\boldsymbol{\beta}$ using polynomial regression with degrees 1, 3, 7, and 12 respectively. You also need the function `computeCost.m` (for computation of RMSE, look inside the code). You wrote this function in the last exercise. You need to keep it in Matlab's path for your code to run (or copy the file over to this week's directory).
- Insert your function `leastSquares` in the script. You have to do this at two places.
- If the code runs successfully, you will see the data and the fit. You will clearly see why linear regression is not a good fit, while polynomial regression produces a better fit.
- Take a look at `myPoly.m` that we have provided with the code. What does it do? This code makes the $\boldsymbol{\Phi}$ matrix that we encountered in the lecture on Ridge Regression. Read the lecture notes and make sure you understand the function.
- If you look at the printed output of the script `visualize.m` (in Matlab's command window), you can see that RMSE decreases as we increase the degree of the polynomial. Does it mean that the fit gets better as we increase the degree? Which fit is the best in your view?

■

## 3.5 Evaluating model prediction performance

The answer to the last question should be clear if you followed the lecture. If not, discuss with others and clarify.

In practice, it matters that predictions are good for unseen examples, not only for training examples. To simulate the reality, we will now split our dataset into two parts: *training* and *testing*. We will fit the data using training data and compute RMSE on both test and training data.

**Exercise 3.3** Train and test datasets.

- Run the script `trainTestSplit.m`. This script is supposed to show the train and test splits for various polynomial degrees. Again, the value of $\boldsymbol{\beta}$ is set to 0. You need to insert the function `leastSquares` in here.
- Insert `leastSquares.m`. If the code runs successfully, you will see RMSE

values printed for degrees 3, 7 and 12. For each degree, there are again three RMSE values which correspond to the following three splits of the data.
  - 90% training, 10% testing
  - 50% training, 50% testing
  - 10% training, 90% testing
- Look at the training and test RMSE for degree 3. Does this makes sense? Why? Discuss with others if you are unclear.
- Now look at RMSE for other two degrees. Do these make sense? Why? Discuss with others if you are unclear.
- Which split is better? Why? Refer to the lecture notes if unclear.
- To split the data, we use the function `split.m`. Look inside this function and understand how it works. Do you think that the order of samples is important when doing the split?
- The test RMSE for degree 12 is ridiculously high for the split 10%-90%. Why do you think this is the case? The answer lies in numerical inaccuracies. Make sure you understand this.
- **BONUS:** Imagine you have 5000 samples instead of 50. Which split might be better in that situation?

■

## 3.6 Ridge Regression

The previous exercise shows overfitting when using complex models. Let us now correct it using Ridge regression, as discussed in the class.

**Exercise 3.4** Implementing and visualizing Ridge regression
- Write a function called `ridgeRegression.m`. The code should take the data as input, as well as the regularization parameter $\lambda$. Follow the equations derived in the lecture notes.

```
beta = ridgeRegression(y,tX,lambda)
```
- You can debug your code by setting $\lambda = 0$. This should essentially give the same answer as least-squares code. You can also check that for large value of lambda, RMSE should be really bad.
- Choose a split of 50%-50% and plot train and test errors vs $\lambda$ for polynomial degree 7. Follow the sample code given below. You should be able to reproduce the figure 3.1 shown below. Choose the value of $\lambda$ using `logspace()`. Plot the error wrt $\lambda$ using `semilogx()`

```
% given the split (yTr, XTr) and (yTe, XTe)
vals = logspace(-2,2,100)
for i = 1:length(vals)
    lambda = vals(i);
    % ridge regression
    [beta] = ridgeReg(yTr, XTr, lambda);
    % compute training error
    errTr(i) = computeCost(yTr, XTr, beta);
    % compute test error
    errTe(i) = computeCost(yTe, XTe, beta);
end
[errStar, lambdaStar] = min(errTe);
```

■

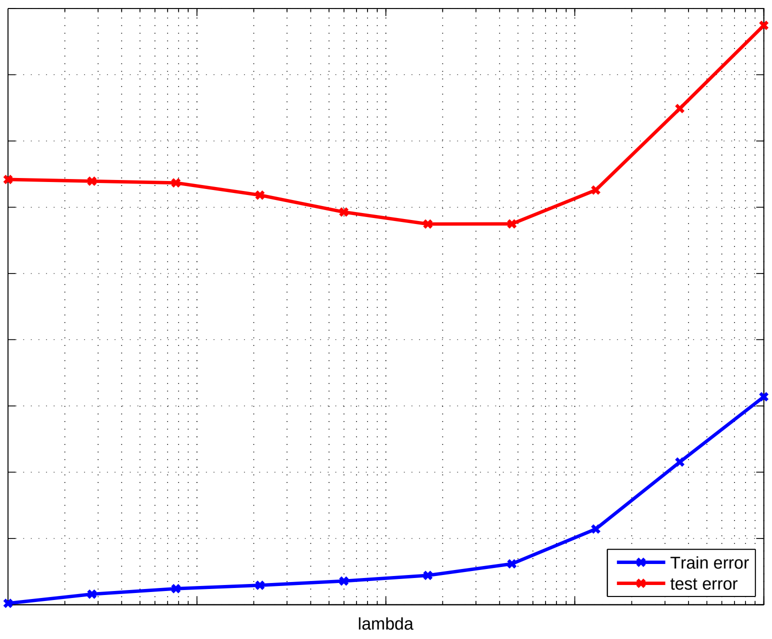

Figure 3.1: Effect of $\lambda$ on training and test errors