Labs
Machine Learning Course
Fall 2024

EPFL
School of Computer and Communication Sciences
Nicolas Flammarion & Martin Jaggi
www.epfl.ch/labs/mlo/machine-learning-cs-433

Problem Set 3, Sept 26, 2024
(Least Squares, Ridge Regression, and Overfitting)

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.

Setup, data, and sample code. Obtain the folder `labs/ex03` of the course GitHub repository

github.com/epfml/ML_course

We will continue to use the dataset `height_weight_genders.csv` as well as a new dataset `dataEx3.csv` in this exercise. We have provided sample code that already contains useful snippets of code required for this exercise.

You will be working in the notebook `ex03.ipynb` for the exercises of this week, by filling in the corresponding functions in the provided template code.

1 Least Squares and Linear Basis Functions Models

1.1 Least squares

Exercise 1:

- Fill in the notebook function `least_squares(y, tx)` which implements the solution of the normal equations as discussed in the class. This function should return the optimal weights and the mean-squared error. Hint: You should not try to solve a linear system $Ax = b$ by using the `numpy.linalg.inv` function.
- 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 a similar resulting $\boldsymbol{w}$ vector using all three methods.

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 the grid search method. If you are lagging behind, do not worry. You will get the opportunity to catch up later, but it is important that you eventually take time to finish previous exercises.

1.2 Least squares with a linear basis function model

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

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

$$\phi_j(x) := x^j \tag{1}$$

As we have seen in the lecture notes, the technique of feature expansion by the linear basis function model does allow us to still use linear regression techniques, to fit nonlinear data (recall that in our first simple setting, we assume that each input point is just one real value). As a result, we will be able to fit the data using different degrees of polynomials, e.g. a degree two polynomial (which is a linear combination of 1, x and x^2), or a degree three polynomial (which is a linear combination of 1, x, x^2 and x^3), etc.. Higher degree polynomials are more expensive to compute and to fit, but can capture finer details in the data, which results in more expressive models. Think about the pros and cons of choosing a very high or very low degree.

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

$$\text{RMSE}(\boldsymbol{w}) := \sqrt{2 \cdot \text{MSE}(\boldsymbol{w})} \tag{2}$$

The magnitude of MSE can be difficult to interpret since it involves a square, while RMSE provides a more interpretable measure on the same scale as the error of one point. There are better measures in terms of statistical properties, like R^2, but we don't need these for now. See the book "Introduction to Statistical learning" if you're interested in more details.

Let us now implement polynomial regression, using the technique of linear basis functions, and visualize the predictions.

Exercise 2:

The goal of this exercise is to plot the data along with predictions using polynomial regression. Your goal is to find a good $\boldsymbol{w}$ using polynomial regression, when using polynomials of degrees 1, 3, 7, and 12 respectively. You might want to reuse the function from the previous exercise to calculate the RMSE.

- Fill in the notebook function `build_poly(x, degree)`. The input of this function is the vector of the data examples $x_n \in \mathbb{R}$ for $1 \leq n \leq N$. As an output, the function must return the extended feature matrix

$$\widetilde{\boldsymbol{\Phi}} := \begin{bmatrix} \boldsymbol{\phi}(x_1) \\ \cdot\cdot \\ \boldsymbol{\phi}(x_n) \\ \cdot\cdot \\ \boldsymbol{\phi}(x_N) \end{bmatrix} \quad \text{where} \quad \boldsymbol{\phi}(x_n) := [1, x_n, x_n^2, x_n^3, \ldots, x_n^{\text{degree}}]$$

  that is the matrix formed by applying the polynomial basis functions to all input data, for the degree of $j = 0$ up to $j =$`degree`.

  When finished, you must COPY your implementation to the separate file `build_polynomial.py` for the plot function to work.

- Fill in the notebook function `polynomial_regression()`. 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.

- 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?

2 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:

The notebook function `train_test_split_demo()` is supposed to show the train and test splits for various polynomial degrees.

- To split the data, please fill in the notebook function `split_data(x, y, ratio, ...)`. Do you think that the order of samples is important when doing the split?
- Fill in the notebook function `train_test_split_demo()`. 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 make sense? Why?
- Now look at RMSE for the other two degrees. Do these make sense? Why?
- Which split is better? Why?
- The test RMSE for degree 12 is ridiculously high for the split 10%-90%. Why do you think this is the case?
- **BONUS:** Imagine you have 5000 samples instead of 50. Which split might be better in that situation?

3 Ridge Regression

The previous exercise shows overfitting when using complex models. Let us now correct it using Ridge Regression, defined as

$$\min_{\boldsymbol{w}} \quad \frac{1}{2N} \sum_{n=1}^{N} \left[y_n - \boldsymbol{x}_n^\top \boldsymbol{w} \right]^2 \; + \; \lambda \|\boldsymbol{w}\|_2^2$$

- Fill in the notebook function `ridge_regression()`. 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.
- Play with the demo `ridge_regression_demo()` by choosing a split of 50%-50% and plot train and test errors vs λ for polynomial degree 7. You should get a similar plot as Figure 1.

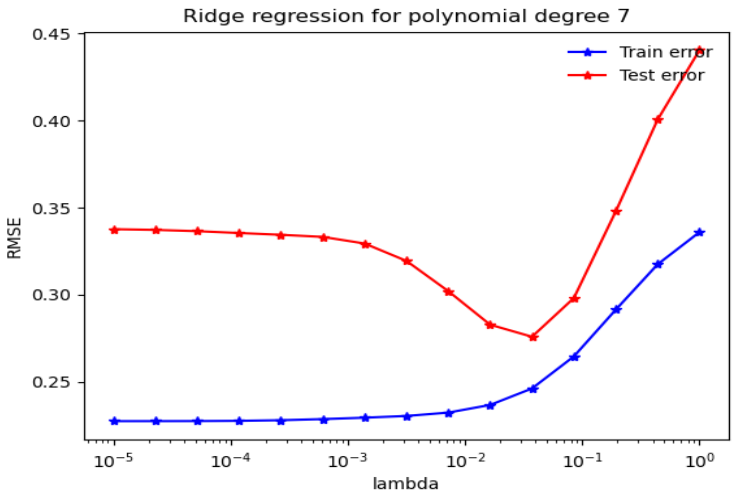

Figure 1: Ridge Regression Demo.

Theory Exercises

1. Warm-Up

   (a) Show that the sum of two convex functions is convex.
   Hint: use the definition of convexity,

   $$f : X \to \mathbb{R} \text{ is convex} \quad \Leftrightarrow \quad \forall \boldsymbol{x}, \boldsymbol{y} \in X, \forall \lambda \in [0,1] : f(\lambda \boldsymbol{x} + (1-\lambda)\boldsymbol{y}) \leq \lambda f(\boldsymbol{x}) + (1-\lambda) f(\boldsymbol{y}).$$

   (b) How do you solve the linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$? When is it not possible, and why?
   Hint: Invertible matrix

   (c) What is the computational complexity of
   - Grid search?
   - (one step of) Gradient Descent for linear regression with MSE cost?
   - (one step of) Stochastic Gradient Descent for linear regression with MSE cost?

   If needed, refresh your memory of the complexity of algebraic operations.

   (d) Consider a problem with two input variables, $\boldsymbol{x} = (x_1, x_2)$, and one output variable y. Given the two samples below, find the coefficients $\boldsymbol{w} = (w_1, w_2)$ of the linear relationship $\boldsymbol{x}^\top \boldsymbol{w} = w_1 x_1 + w_2 x_2 = y$.

	x_1	x_2	y
Sample 1	400	-201	200
Sample 2	-800	401	-200

   Do the exercise again, but with a slight change in the inputs: x_1 for sample 1 is now 401 instead of 400

	x_1	x_2	y
Sample 1	401	-201	200
Sample 2	-800	401	-200

   Compare the resulting $\boldsymbol{w} = (w_1, w_2)$ for both cases. Familiarize yourself with the concept of condition number as a way to diagnose ill-conditionning. You can find condition number calculators online or use `numpy.linalg.cond`).

2. Cost functions

   A cost function defines how you evaluate a solution, and you might have different requirements depending on the problem. Using the MSE, if a your model makes an error of 5 on a sample, you add 25/2 to the cost of your model, regardless of the target. You might want to penalize this differently if you care about the relative error; an output of 1005 when 1000 was expected might be OK, but mistaking a 6 for a 1 might not. In this case, you can use a function that takes the relative error of the target y_n into account, like this one:

   $$\mathcal{L}_n(f(\boldsymbol{x}_n, \boldsymbol{w}), y_n) := \frac{(f(\boldsymbol{x}_n, \boldsymbol{w}) - y_n)^2}{{y_n}^2 + \epsilon}.$$

   Where f is the model and ϵ is a small constant to avoid divisions by zero. Note that we have defined the cost function per example here. You can imagine the total cost function being defined as $\mathcal{L} := \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n$.

   (a) Try the function on some [prediction, target] pairs, or plot it, to see how it behaves (by hand or using Python or the wolfram alpha website, no need to code)

   (b) Compute its gradient, assuming a standard linear regression $f(\boldsymbol{x}_n, \boldsymbol{w}) := \boldsymbol{x}_n^\top \boldsymbol{w}$

   (c) How would you implement the gradient? Again, no need to code - try to find a formula using standard matrix operations, along with element-wise multiplication and summation/product over columns/rows.

   (d) How sensitive is this function to outliers? Compare two cases: the target is 1, but in one case our model assigns it 10, and in the other 100. (e.g. with $\varepsilon = 1$) How does the error changes? Compare with the following cost function, for the n data example:

   $$\mathcal{L}'_n(f(\boldsymbol{x}_n, \boldsymbol{w}), y_n) := \big(\log(f(\boldsymbol{x}_n, \boldsymbol{w}) + 1) - \log(y_n + 1)\big)^2.$$

   Note: The higher the error on a sample is, relative to the other samples, the more your model will try to fit this sample.