Report for project 1

Adam P W Sørensen

October 8, 2018

In this project we used three different linear regression methods to approximate the Franke Function and some real world terrain data. The methods we use are Ordinary Least Square, Ridge, and Lasso regression. For each we use a 10-fold cross validation to pick the best model and then we preform statistical tests to ascertain the validity of our models. None of them are particularly good.

1 Introduction

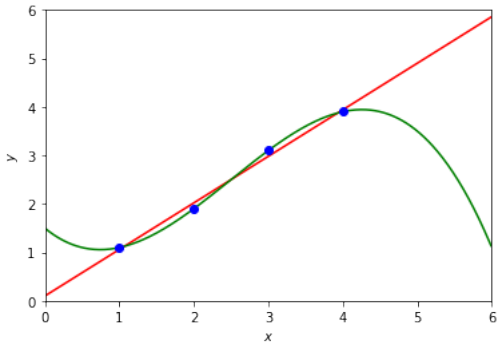
This project is concerned with the following problem: Suppose we know the values of a function f on a set of points $\{x_1, x_2, \ldots, x_n\}$, can we then predict $f(x)$ for any x? In full generality this is plainly impossible (though see [Hardin and Taylor, 2008]). But if we allow ourselves to believe that the function f is not too erratic, then we can certainly try to notice a pattern and use that for our predictions. Like in an IQ test. In fact we would also like to find a pattern even if we do not know $f(x_i)$ exactly, but only with some small noise thrown in. This corresponds more closely to a real world measurement scenario.

Linear Regression methods are a mathematically rigorous way of noticing these types of patterns. The methods produce a so-called parameter vector from the know values of f and then produce a prediction function that is linear in this parameter vector, and not in the inputs x. A typical usage is to try to find a polynomial approximation of f. If we have n data points $\{x_1, x_2, \ldots, x_n\}$ and we know the function values $f(x_i)$ one can perfectly fit a $n-1$ degree polynomial to this data. This will usually be the wrong approach. Suppose we measure the following.

x	1	2	3	4
f(x)	1.1	1.9	3.1	3.9

Any sane person would predict that $f(5)$ ought to be almost 5 (they might even predict that it is 5.1). Figure 1 shows a what happens when we try to predict from this using a basic linear regression method to get either a first or third degree polynomial.

Figure 1: An example of over fitting.



The green line shows the third degree polynomial, the red line is the first degree polynomial and the blue dots are the observations. We see that the third degree polynomial predicts that $f(5) \approx 3.5$. To avoid this type of problem one has to validate the model used. A simple test that would have worked here is to try to predict using only some of the observation, and then test against the remaining observations. In general we use more sophisticated test, one such test is called cross validation and will be discussed later.

In this report we will discuss three specific linear regression methods: Ordinary Least Squares, Ridge, and Lasso regression. We will talk about the weakness and strengths of each one. We will apply them to the so-called Franke Function and some real world terrain data and discuss how successful they are, as determined by statistical tests. It will unfortunately transpire that neither of our methods are particularly good for predicting the behaviour of Franke Function or the terrain data.

All the code I have implemented can be found in the jupyter notebook `Projekt 1.ipynb` at `https://github.com/adam-wie/ml-exercises`.

The report is structured as follows. In Section 2 we will introduce, more rigorously, linear regression and discuss the three specific methods we use. We will touch on their theory and how I have coded them. In Section 2.3 we turn our attention to model selection and validation, looking at cross validation and the bias and variance of a model. Again there is brief discussion of the theory and comments on the implementation. In sections 4 and 5 we look at how our models work on the Franke Function and the terrain data. Finally in Section 6 we summarize our findings.

2 Linear Regression Methods

Broadly speaking the idea of linear regression is as follows. We are given N data points each represented by a row vector $\boldsymbol{x_i}$ of some size p and the corresponding real measurements $z_1, z_2, \dots z_N$. Then we try to find a column vector $\boldsymbol{\beta}$ such that

$$\begin{pmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \vdots \\ \boldsymbol{x_n} \end{pmatrix} \boldsymbol{\beta} \approx \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix}.$$

The goal is to choose $\boldsymbol{\beta}$ such that the approximate equality holds as best as possible. Exactly how we measure what it means for two vectors to be approximately equal and if we put on other constraints will define the different regression methods.

In our case we are interested in finding a polynomial that approximates a function f of two variables. When we want a polynomial of degree d we will first find N points in the plane $(x_1, y_1), (x_2, y_2), \dots (x_N, y_N)$ and record the values $z_1 = f(x_1, y_1), z_2 = f(x_2, y_2) \dots z_N = f(x_N, y_N)$. The data points will not only be the pairs (x_i, y_i) but instead the row vectors

$$\boldsymbol{x_i} = \begin{pmatrix} 1 & x_i & y_i & x_i^2 & x_i y_i & y_i^2 & \cdots & y_i^d \end{pmatrix},$$

containing all expressions $x_i^k y_i^l$ with for k, l non-negative integers with $k + l \leq d$. To ease notation we put

$$X = \begin{pmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \vdots \\ \boldsymbol{x_N} \end{pmatrix}, \quad \text{and} \quad \boldsymbol{z} = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_N \end{pmatrix}.$$

The goal is then to find $\boldsymbol{\beta}$ such that

$$X\boldsymbol{\beta} \approx \boldsymbol{z}. \tag{1}$$

Given such a $\boldsymbol{\beta}$ we can then *predict*, or perhaps more accurately estimate, the value of f at a given point (x, y) by computing

$$\begin{pmatrix} 1 & x & y & x^2 & xy & y^2 & \cdots & y^d \end{pmatrix} \boldsymbol{\beta} = \beta_1 + \beta_2 x + \beta_3 y + \beta_4 x^2 + \cdots. \tag{2}$$

The expression on the right hand side is our polynomial approximation of f.

We now consider three kinds of linear regression: Ordinary Least Square, Ridge and Lasso. Below we briefly discuss the theory of each and how they were implemented in the code. It is all implemented in the third code block in the jupyter notebook, and in the fourth it is tested against the functionalities provided by sci-kit learn.

2.1 Ordinary Least Square Regression

In the Ordinary Least Square (OLS) method (described in Section 3.2 of [Hastie et al., 2009]) we measure the approximate equality in equation (1) in 2-norm. That is we are looking for a $\boldsymbol{\beta}$ the minimizes

$$\|X\boldsymbol{\beta} - \boldsymbol{z}\|_2,$$

where $\|\cdot\|_2$ denotes the 2-norm, i.e. the usual euclidean distance. From equation (3.6) in [Hastie et al., 2009] we see that this is achieved by setting

$$\boldsymbol{\beta} = (X^T X)^{-1} X^T \boldsymbol{z}.$$

In the code this is implemented by 2 classes and one ancillary function. The function `polymatrix` takes a matrix with two columns, called $\boldsymbol{x}$ and $\boldsymbol{y}$, and and a natural number d, it produces the following matrix

$$\begin{pmatrix} \mathbf{1} & \boldsymbol{x} & \boldsymbol{y} & \boldsymbol{x}^2 & \boldsymbol{x}\boldsymbol{y} & \boldsymbol{y}^2 & \cdots & \boldsymbol{y}^d \end{pmatrix},$$

where all multiplications are done pointwise. In other words `polymatrix` constructs the matrix we called X above. To implement `polymatrix` we first note that picking non-negative k, l such that $0 < k+l \leq d$ is the same as picking $k-i, i$ such that $1 \leq k \leq d$ and $0 \leq i \leq k$. Now `polymatrix` first computes a list where each entry is a column vector of the form $\boldsymbol{x}^{k-i}\boldsymbol{y}^i$, and then use `np.hstack` to combine them and a column vector of ones into a single matrix.

The class `myOLS` has a single variable `beta` and two functions `train` and `predict`. The function `train` just uses numpy to implement equation (3.6) in [Hastie et al., 2009], simlarly `predict` implements the predict equation (2) using numpy. The whole reason for this class is just to make the next one have cleaner code.

The class `polynomialOLS` is what we use to do our OLS regression. When initiating an instance of this class, one has to give the degree of the polynomial we want to approximate by. The functions `train` and `predict` both expect a matrix `XY` with two columns, each row having a coordinate pair x, y, in addition `train` needs a column vector $\boldsymbol{z}$ of real function values. They both first use `polymatrix` to build the right matrix, as discussed above, and the simply call the `train` and `predict` from `myOLS`. The design choice of making the `train` and `predict` functions take care of transforming the input from x, y pairs to the polynomial matrix, was to make later code more readable by avoiding having to make the transformation before each call. In addition `polynomialOLS` has function to return the computed vector $\boldsymbol{\beta}$, a function to print some info about the model, and a function I use for plotting. I suspect the latter is quite unnecessary, and could be removed if I were better at using numpy's vectorize functionality.

2.2 Ridge Regression

The theory for Ridge regression is described in Section 3.4.1 of [Hastie et al., 2009]. For ridge regression we need a parameter $\lambda > 0$ in addition to the input data matrix X and a true value vector $\boldsymbol{z}$. The method is similar to OLS, but the inclusion of λ is used to reduce the variance of the computed $\boldsymbol{\beta}$ which in OLS can change a lot with small changes of inputs. When choosing β we again want to minimize the 2-norm of $\boldsymbol{z} - X\boldsymbol{\beta}$ but at the same time also the expression

$$\lambda \sum_{i=1}^{p} \beta_i^2 \quad \text{where} \quad \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{p-1} \end{pmatrix}.$$

Note that β_0 is not included, that is we don't penalize the constant term of our polynomial. Like for the OLS case there is an explicit formula for $\boldsymbol{\beta}$. For this, we do not include the column of 1's in the matrix X and we assume that each column of X and $\boldsymbol{z}$ are centered (i.e. has mean 0), then by equation (3.44) in [Hastie et al., 2009]:

$$\boldsymbol{\beta} = \left(X^T X + \lambda I\right)^{-1} X^T \boldsymbol{z}. \tag{3}$$

This only gives the $\beta_1, \beta_2, \ldots, \beta_{p-1}$.

If we do not assume that X is centered we let $\bar{x}_i$ be the mean of the i'th column of X and let $\bar{z}$ be the mean of $\boldsymbol{z}$. Then we replace X with the matrix $(x_{ij} - \bar{x}_j)_{i,j}$ and $\boldsymbol{z}$ with $(z_i - \bar{z})$. Following programming conventions we will still call them X and $\boldsymbol{z}$. We still use equation (3) to compute the $\beta_1, \beta_2 \ldots, \beta_{p-1}$, however we put

$$\beta_0 = \bar{z} - \begin{pmatrix} \bar{x}_1 & \bar{x}_2 & \cdots & \bar{x}_{p-1} \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{p-1} \end{pmatrix}, \tag{4}$$

The term we subtract can be thought of as retransforming the intercept out of the assumption that the columns are centered.

In the code the class `polynomialRidge` implements Ridge regression. To initiate an instance of this class one needs to give the degree of the polynomial we want to approximate by, and the parameter λ. (`lambda` has its own python meaning so we cannot use that for the parameter). Again the most interesting functions are `train` and `predict`. As in the OLS case we use `polymatrix` to go from two columns of x's and y's too the all the terms of polynomial. The heavy duty of centering and uncentering is done in `train`, which just uses numpy to implement the maths described above. Because of that `predict` is just as straight forward as it was in the OLS case. In both `train` and `predict` we must delete the column of 1's coming from `polymatrix` as the intercept is treated different from all other β parameters. As for OLS we also have some extra functions that are used later to print and extract parameters.

2.3 Lasso Regression

Lasso regression is similar to Ridge regression in that it uses a parameter λ to lessen the variance of the computed β_i, however unlike Ridge and OLS it is much more likely to set a given β_i equal to 0. We will see this in action later. The theory of Lasso regression is in Section 3.4.2 of [Hastie et al., 2009]. Like in Ridge regression we try to minimize not only $\|\boldsymbol{z} - X\boldsymbol{\beta}\|_2$ but at the same time also an extra term, which in the Lasso case is

$$\lambda \sum_{i=1}^{p} |\beta_i|.$$

So it is the 1-norm of $\boldsymbol{\beta}$ (without its first row) instead of the 2-norm in Ridge. Here there is no closed form expression for $\boldsymbol{\beta}$.

The implementation of Lasso regression is in the class `polynomialLasso`. We use sci-kit learns Lasso regression functionality and `polynomialLasso` just acts as a wrapper that takes care to transform the input appropriately. In all respects it is the same as how `polynomialOLS` used the class `myOLS`.

3 Model Selection and Verification

3.1 Cross Validation

To choose between different models I used cross validation, implemented in the fifth code block of the jupyter notebook. I build the code on the theory described in Section 7.10.1 of [Hastie et al., 2009]. In the optimal situation where we have easy access to lots of data, we would split our data in to three parts, one part for training our models, one part for used to decide between models, and then a final part to test the reliability of our chosen model. We are actually in this situation, but will pretend we are not.

The purpose of cross validation is to not keep a partition of the data specifically for validation, and instead use the training data for validation. It works as follows. Suppose we have a model M and our training data are data points $\{\boldsymbol{x}_i\}_{i=1}^N$ with corresponding true values $\{z_i\}_{i=1}^N$. We fix a natural number k, which is the number of so-called folds we will use. Then we partition $\{1, 2, \ldots, N\}$ into k almost equally big sets $A_1, A_2, \ldots, A_k$. In symbols

$$\{1, 2, \ldots, N\} = \bigsqcup_{j=1}^{k} A_j.$$

For each j we train our model M on $\{1, 2, \ldots, N\} \setminus A_j$ and call the corresponding prediction function f_j. Now we estimate the square error the model will lead to on an

unseen data point, denoted cv, by

$$cv = \frac{1}{N}\sum_{j=1}^{k}\sum_{i\in A_j}(f_j(\boldsymbol{x}_i) - z_i)^2.$$

Note that the prediction of $\boldsymbol{x}_i$ is done by a model not trained on $\boldsymbol{x}_i$. In case k evenly divides N so that all the A_j are of size N/k we can write this as

$$cv = \frac{1}{k}\sum_{j=1}^{k}\left[\frac{1}{|A_j|}\sum_{i\in A_j}(f_j(\boldsymbol{x}_i) - z_i)^2\right].$$

Here the terms enclosed in $[\cdot]$ is just the mean squared error the model M makes when trained on $\{1, 2, \ldots, N\}\backslash A_j$ and tested against A_j. So we can think of the cross validation error estimate as an average of mean squared errors.

Obviously different choices of k will lead to different estimated errors. The choice of k seems more art than science, but apparently $k = 10$ is an established best practise.

In the code I have implemented a function `cross_validation` to do cross validation. It takes 4 arguments, the number of folds `k`, a two column matrix `XY` of input data, a column vector `z` or true values, and a model `model` which has `train` and `predict` functions. Clearly these functions should work on the data points and true values. The idea is to use the regression methods I have implemented above as one of the models.

For the implementation we first record the length of the columns, N say, in `XY` and then use numpy functions to partition $\{1, 2, \ldots, N\}$ into `k` sets. We loop over the folds and use an indexing mask to pick out all the data not in the current fold, train the model on it, and record the squared differences between predictions from the current fold and the real data. Finally we return the average of all these squared errors.

Numpy contains a function called `cross_val_score` that I would have liked to compare my code to. However I couldn't get it to play well with my regression methods. The documentation for `cross_val_score` (`http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html`) says the estimator, what I called model, should implement a `fit` function. But even if I rename my `train` to `fit`, or indeed just add the line `self.fit = self.train` to the `__init__` of my classes, I still get errors asking for more functions to be implemented. I chose not to track through the numpy documentation to figure out what exactly I needed to implement. Instead, I am contended to note the the predicted errors seem somewhat close to the actual errors we get when testing against unseen data.

3.2 Verification

Once a model is chosen, we want to see how stable and accurate it is. To this end, in the sixth code block of the jupyter notebook, I have done basic implementations of

mean squared error functions, R^2 score, and variance of the entries of a vector. There are formulas for these which I just translated to numpy. In the seventh code block they are tested against the native numpy functions and give the same results.

In the eighth code block I implement a function `BV_estimate` to record the bias and variance of our model, I also implement `beta_variance` to record the variance of the β_i's. Both are based on the ideas of cross fold validation. For `BV_estimate` we proceed similarly to cross fold validation and partition $\{1, 2, \ldots, N\}$, but this time into $k+1$ folds $\{A_j\}_{j=1}^{k+1}$. We now successively train the on each of $\{1, 2, \ldots N\} \setminus (A_j \cup A_{k+1})$ to get a prediction function f_j which is tested against A_{k+1}. This way we get multiple predictions for each point in A_{k+1} and so can compute the models bias and variance as

$$\text{bias}^2 = \frac{1}{|A_{k+1}|} \sum_{i \in A_{k+1}} \left[\frac{1}{k} \sum_{j=1}^{k} (f_j(\boldsymbol{x}_i) - z_i) \right]$$

and

$$\text{variance} = \frac{1}{|A_{k+1}|} \sum_{i \in A_{k+1}} \left[\text{Var}(f_j(\boldsymbol{x}_i)) \right],$$

where the $\text{Var}(f_j(\boldsymbol{x}_i))$ is taken as j varies and i is kept fixed. The theory of this is described in Section 7.3 of [Hastie et al., 2009], but the implementation is mostly based on the piazza note on Bias and Variance of a model.

The function `beta_variance` is again build in a way very similar to `cross_validation`. This time instead of recording prediction errors I record the coefficient vector $\boldsymbol{\beta}_j$ of each training session, and then compute the variance of each entry.

A reasonable critique of both methods is that I should perhaps repeat the k-fold process a number of times to get more $\boldsymbol{\beta}$s or point predictions to get more accurate variances. However, if 10 folds works for cross prediction I reason that it will work fine for these too.

4 Results on the Franke Function (parts a,b,c)

The code blocks 9-17 looks at the Franke Function and the different linear regression methods. In block 9 the data is setup, I pick 1000 pairs (x, y) in $[0, 1]^2$ and record their real values with a little added noise. I used cross validation for each regression method and picked the model that gave the smallest predicted error. The results are summarized in Table 1.

For each of the best methods I computed the actual mean squared error and R^2 score against unseen data. I also found the model vaiance and bias. These results are summarized in Table 2.

Table 1: Method selection for the Franke Function.

Regression Method	Polynomial degree	lambda parameter	Predicted error
OLS	5		0.044361
Ridge	5	0.1	0.049839
Lasso	4	0.1	0.124234

Table 2: Model validation for the Franke Function.

Regression Method	Mean squared error	R^2	bias^2	model variance
OLS	0.046539	0.636927	0.043213	0.000132
Ridge	0.058986	0.539831	0.054887	0.000056
Lasso	0.130758	-0.020091	0.128723	0.000007

On the positive side, we note that the actual mean squared errors are somewhat close to the predicted values. We also see that as expected the bias of OLS is smallest, then Ridge and then Lasso, and the variances go the other way. Other than that, there is not a lot of positive things to say about these models. They have abysmal R^2 scores. In particular I am puzzled by the negative score for the Lasso regression, if one looks at the model, we will see that all parameters except the intercept are zero. However, an R^2 score of 0 is suppose to indicate that your model is no better then a hyperplane. So the Lasso model I found manages to be a bad hyperplane model. This explanation for this is that the average of the z_i we trained on is different from the average of the unseen z_i.

Let us now take closer look at coefficients (see code blocks 54, 57, and 66). The first thing that springs out, is that as noted that all the Lasso coefficients, except the intercept, are 0 and that they have no variance. There is a small variance on the intercept, but it is of the order of 10^{-5}. Thus the Lasso model is very confident in its poor prediction. That all the parameters are set to zero is a feature of the Lasso model, but in this case it works against it.

The next thing to notice is how big some of the coefficients of the OLS regression are. For instance β_3, which corresponds to the x^2 term is -28.83003238, and there is a variance on it of 8.30320758. So we get a 95% confidence interval (rounded off) for β_4 of

$$[-28.83 - 2 \times 8.30; -28.83 + 2 \times 8.30] = [-45.43; -12.23].$$

So I am 95% sure the real value of β_3 is between -45 and -12. Which I read to mean that the computation is almost meaningless.

Turning to Ridge regression we see much smaller coefficients and much smaller variance of the β_i. Looking again at β_3, the x^2 term, here it is -1.5575589 with a variance of

Table 3: Model selection for terrain data.

Regression method	mean squared error	R^2	bias^2	model variance
Ridge	27383.656382	0.767798	24636.566560	5.116623

0.00327892, which gives a 95% confidence interval of β_3 of

$$[-1.56411674; -1.55100106].$$

The variance for all the β_i are similarly small so we will get similar confidence intervals.

Looking at all the data, it seems that the Ridge regression giving a fifth degree polynomial with a λ parameter of 0.1 gives the best model. While it has a worse R^2 score than the OLS model, its much more stable behaviour gives it the nudge.

5 Results for Terrain Data

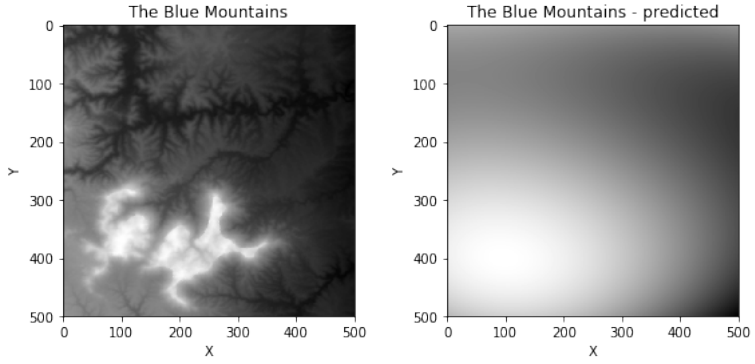
For the terrain data I chose a picturesque mountain range just outside of Sydney called the Blue Mountains. I again used cross validation to select the model with the smallest predicted error (code block 68), and I got Ridge regression for a third degree polynomial with a λ parameter of 0.5. Running statistical tests as above, I got the results shown in Table 3

The first thing to note is the enormous mean squared error. This number was expected to be bigger than for the Franke Function, from the simple fact that the terrain data function takes values in a much bigger range. I also note that we have an almost respectable R^2 score of approximately 0.77, nothing impressive but notably better than for the Franke Function. There is very little variance on the β_i's except for the intercept, where the variance is 14.2955179, leading to a confidence interval on β_0 of

$$[745.8323202; 803.0143918]$$

For a final test, I plotted a 500 by 500 corner of the map, and the same corner as predicted by the model in code block 76, see Figure 2.

Figure 2: The Blue mountains and the Ridge regression prediction.



Looking at it, we can see that the model picks up the very bright spot in the lower left corner and sort of picks up the dark lines running along the X-axis around Y-coordinate 100–200. In a way the predicted map looks like a very blurry version of the real map.

6 Conclusion

From the modelling of the Franke Function and the terrain date, we see that the linear regression methods acts as the theory predicts. The Ridge and Lasso regressions are more stable than the Ordinary Least Squares regression and Lasso is much more likely to set a given β_i to 0. None of the linear regression methods are particularly well suited for these particular problems. Hence other methods are needed if we want useful predictions.

References

[Hardin and Taylor, 2008] Hardin, C. S. and Taylor, A. D. (2008). A peculiar connection between the axiom of choice and predicting the future. *Amer. Math. Monthly*, 115(2):91–96.

[Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning*. Springer Series in Statistics. Springer, New York, second edition.