FYS-STK3155 Project 1

Didrik Sten Ingebrigtsen

October 10, 2020

Chapter 1

Abstract

Linear regression is an easy and highly interpretable form of machine learning. It does however still require some tuning, that quickly becomes less intuitive once the dimensionality and/or the complexity of the data rises. In this report we will describe a process for tuning by splitting the data into three sets, and using resampling methods to estimate how well different hyperparameters perform using only two of them. Lastly, we discuss how well this tuning process is at finding the best models, by testing it on the third set.

Chapter 2

Introduction

The main objective of this project was to implement code for simple and regularised linear regression methods, and to find optimal hyperparameters and their associated errors for both the Franke function and actual terrain data. To do this, the question of how to measure what is optimal is key, and within that lies an important discussion of bias and variance.

To solve this problem, we have firstly implemented bootstrap and k-fold resampling techinques, and tested their capabilities on the predictable and flexible Franke function. After the next section, where we describe linear regression more precisely, is a section on this validation of my implementation and techniques. Then, we apply this to actual terrain data, and tune our hyperparameters. Lastly, we discuss the results, and further implications on linear regression and hyperparameter optimalisation.

Chapter 3

Theory

3.1 Linear regression

Linear regression is a class of regression where the predictor variable is taken through some set of functions to make a design matrix X where each row represents a datapoint and each column a function.[1] Then, a vector of parameters β is multiplied to it, and a regularisation term λR (where λ is the regularisation coefficient) that could be a function of both the design matrix and the parameters could also be subtracted. Since I am working with terrain, and for simplicities sake, I have limited myself to these functions going into the design matrix being a basis expansion (making the columns of X polynomial). To sum up, this is my model:

$$\tilde{y} = X\beta - \lambda R$$

In here, each columns of an n-dimensional X is on the form $x_0^{p_0} \cdot x_1^{p_1} \cdot \ldots \cdot x_{n-1}^{p_{n-1}}$ for all p that satisfies the condition $\sum_{i=0}^{n} p_i \leq P$ where P is the maximum polynomial, a hyperparameter.

The assumptions needed for linear regression with the design matrix made with these polynomials is that our dependent variable is composed of some

[1] This, as well asthe other theory, is adapted from Hjorth-Jensen(2020) and Hastie et. al (2009)

underlying polynomial function and an error term:

$$y = f(x) + \varepsilon$$

3.2 Fitting and overfitting

To make a concrete model, we have to fit a β to our X and y data. This is done by defining some cost function, a function taking in our model and some data and our model and returning error, and minimizing this function. The by far most common cost function, at least for linear regression, is the residual sum of squares (RSS). This is the sum of the squared differences from our prediction and our actual data. Minimizing this is the same as finding the points where the first order partial derivative is 0, and the second order partial derivative is positive, both with regards to β

$$RSS = (y - X\beta)^T(y - X\beta) \tag{3.1}$$

$$\frac{\partial RSS}{\partial \beta} = -2X^T(y - X\beta) \tag{3.2}$$

$$\frac{\partial^2 RSS}{\partial \beta \partial \beta^T} = 2X^T X \tag{3.3}$$

Given that X is invertible (which we'll come back to later), $\frac{\partial^2 RSS}{\partial \beta \partial \beta^T} > 0$,and so any value of β that satisfies $\frac{\partial RSS}{\partial \beta} = 0$, is an optimal β, or a $\hat{\beta}$.

$$\frac{\partial RSS}{\partial \beta} = -2X^T(y - X\beta) \tag{3.4}$$

$$0 = X^T(y - X\beta) \tag{3.5}$$

$$\hat{\beta} = (X^T X)^{-1} X^T y \tag{3.6}$$

If our dependent variable was just a polynomial of a manageably low degree, this would be enough. However, because of our noise term ε, minimizing

RSS doesn't just fit our $\hat{\beta}$ to the actual underlying structure. Therefore, the RSS score found for $\hat{\beta}$ isn't a good estimate of how good our model would do to for any new data. To see how good a model is, we therefore split the data into several parts, at least a training and testing set. Training data is used to find some $\hat{\beta}$, and the testing data is held back until we want to evaluate the model. If we want to select hyperparameters like P and λ, we should also evaluate our model on some dataset that isn't the training set, but also not the testing set, so that we get an honest evaluation of how good the model is on unseen data in the end, and don't tailor neither the parameters or the hyperparameters to the testing data. That ss called overfitting, and would give us an inflated belief in our own model.

3.3 Regularisation

Another way to prevent overfitting, other than to split the data smartly, is to regularise it. Regularising is giving large parameter values some penalty. This prevents overfitting because parameters that have little predictive value in the training set, are highly unstable. If it turns out that this parameter has some higher importance in another dataset, then a large parameter value could make the estimate of the dependent variable explode. It's better for it to be small, and to therefore miss a potential structure, than to have it large and be likely to mistake small noise for an actual structure.

The two ways of regularisation we'll be looking at in this project is lasso and ridge regularisation. Both penalise high values for parameters, but their functions for determining how much to punish them is slightly different. Lasso subtracts the $L1$ distance, while ridge subtracts the $L2$ distance.

3.4 Resampling

When evaluating the model, and especially when data is scarce, it is very useful to resample. Resampling is a class of methods where you use your data more than one time, in different configurations, to see how different

your model and errors could be given some hyperparameters. Conceptually, a high variance between errors estimated when evaluating a model trained on different samples of your data would indicate that the model is overfitting to the sampled data. Contrary, a low variance could indicate that the search for features isn't too aggressive, and could actually possibly be more aggressive. You can fit a bit more to the data. The consistent errors the models trained on the different samples make is called bias. These two values, the bias and the variance, often have a trade-off. A low variance often gives high bias, and vice versa. Resampling techniques can help find a good trade-off between them.

In this project, we will be using two common resampling techniques, bootstrap and k-fold. Bootstrap works by keeping a constant testing or validation set, and then selecting as many values as there are in the training set from it with replacement, and doing this many times. K-fold on the other hand mixes up training and testing or validation data into one set first, and then divides it into k different sets. Then, one by one, each so-called fold is used as testing or validation data, while the others are used to train on.

To study the relationship between bias and variance, we will be using bootstrap resampling and looking at the variation and bias of the samples different predictions for each datapoint. The formulas needed to do this can be derived from the expectation value of mean squared error like this.

$$
\begin{aligned}
E[MSE] &= E\left[(\boldsymbol{y}-\tilde{\boldsymbol{y}})^2\right] = E\left[(\boldsymbol{f}+\boldsymbol{\epsilon}-\tilde{\boldsymbol{y}})^2\right] && (3.7)\\
&= E\left[(\boldsymbol{f}+\boldsymbol{\epsilon}-\tilde{\boldsymbol{y}}+E[\tilde{\boldsymbol{y}}]-E[\tilde{\boldsymbol{y}}])^2\right] && (3.8)\\
&= E\left[(\boldsymbol{f}-E[\tilde{\boldsymbol{y}}])^2\right] + E\left[(\tilde{\boldsymbol{y}}-E[\tilde{\boldsymbol{y}}])^2\right] + E\left[\epsilon^2\right] && (3.9)\\
&+ E\left[2\boldsymbol{f}\epsilon - 2\epsilon E[\tilde{\boldsymbol{y}}]\right] + E\left[2\epsilon E[\tilde{\boldsymbol{y}}] - 2\epsilon\tilde{\boldsymbol{y}}\right] && (3.10)\\
&+ E\left[2\boldsymbol{f}E[\tilde{\boldsymbol{y}}] - 2E[\tilde{\boldsymbol{y}}] - 2\boldsymbol{f}\tilde{\boldsymbol{y}} + 2\tilde{\boldsymbol{y}}E[\tilde{\boldsymbol{y}}]\right] && (3.11)\\
&= E\left[(\boldsymbol{f}-E[\tilde{\boldsymbol{y}}])^2\right] + E\left[(\tilde{\boldsymbol{y}}-E[\tilde{\boldsymbol{y}}])^2\right] + \sigma^2 + 2\left(\boldsymbol{f}-E[\tilde{\boldsymbol{y}}]\right)\left(E[\tilde{\boldsymbol{y}}-E[\tilde{\boldsymbol{y}}]\right) && (3.12)\\
&= E\left[(\boldsymbol{f}-E[\tilde{\boldsymbol{y}}])^2\right] + E\left[(\tilde{\boldsymbol{y}}-E[\tilde{\boldsymbol{y}}])^2\right] + \sigma^2 && (3.13)
\end{aligned}
$$

Here, the first term is the bias and the second the variation of our model, and the third the variance of the data noise. The bias term is like an integrative term that does something similar to summing up the errors, giving more weight to big ones than small ones. If it is high, there is some consistent error. The variance term doesn't relate to the real data at all, but only looks at how different each prediction of the same datapoint is, while the noise term is both hard to estimate, and hard to do anything with once the data is collected.

Chapter 4

Implementation

4.1 Data

Our end goal is to make robust code that can handle real terrain data, and make a polynomial that fits well to it. Stated another way, we want to be able to make a function that takes in a coordinate, and returns height. To do this, let's start by considering how terrain data looks, and how it relates to polynomials.

Terrain is not smooth. If you were to get terrain data of infinite resolution, you would have to have a polynomial of infinite degrees to describe it perfectly. Since we have no ambition of being able to make some general claim about the nature of terrain, or apply our fitted model to anywhere but the exact area we have trained on, overfitting might not seem like a big risk. Any higher degree of complexity might seem as if it just describes our terrain more detailed. However, since we don't have continous data with infinite resolution, but only discrete samples, a polynomial of very high degree, fitted to this data, does run the risk of only being correct at our samples, and being very unpredictable in between them. This therefore seems like a great candidate for regularized linear regression. We optimally want to have a polynomial of high degree, but we want to penalize large parameters that might be unstable.

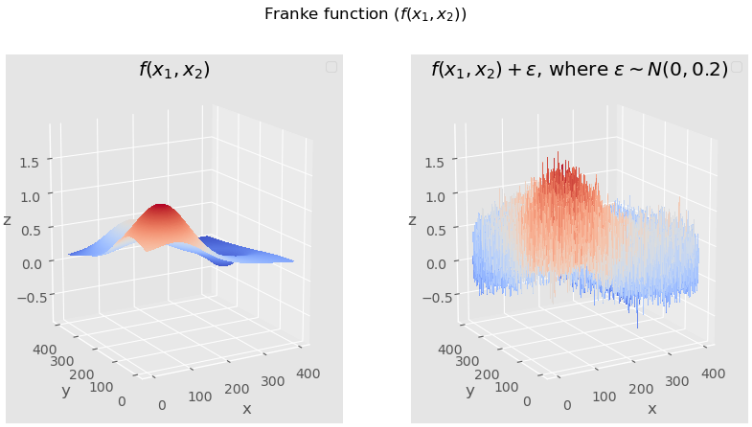

Figure 4.1: Franke function ($f(x_1, x_2)$) data, with the left being just $f(x_1, x_2)$ and the right being $f(x_1, x_2) + \varepsilon$, where $\varepsilon \sim N(0, 0.2)$. The right one is the one being used as data.

To establish a procedure for doing this, and verifying that it works, we will be using the Franke function, and an added noise term. This is flexible, because we can test different values for noise to see how well it generalises, and we also know exactly how the underlying structure is. The Franke function looks like this in writing

$$f(x_0, x_1) = \frac{3}{4}\exp\left(-\frac{(9x_0-2)^2}{4} - \frac{(9x_1-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x_0+1)^2}{49} - \frac{(9x_1+1)}{10}\right) + \frac{1}{2}\exp\left(-\frac{(9x_0-7)^2}{4} - \frac{(9x_1-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x_0-4)^2 - (9x_1-7)^2\right).$$

, and plotted out in a square with $x_1, x_2 \in [-0.9, 1.1]$ it looks like *fig 4.1*.

To handle this data, we need to split it into a training, validation and test set. Usually, choosing which parts of a dataset are put into these subsets is done at random or semi-random. The main reason for doing this is to prevent data bias. However, because our predictor variable is two dimensional and very easily interpretable, we argue that selecting training data on a grid ensures a low bias better than selecting randomly would. This gives an even distribution of points, and makes it so that there are no areas with low data

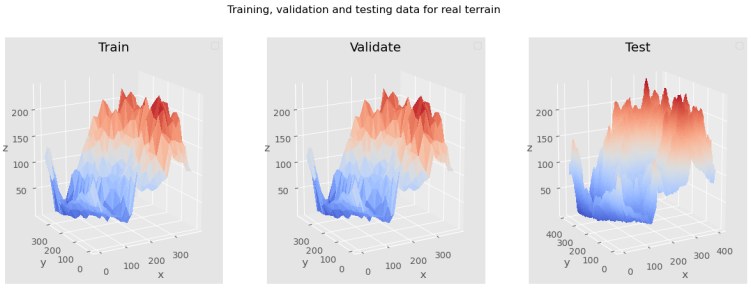

Figure 4.2: The actual terrain we want to try out our methods on in the end, with one plot for each of our datasets.

Figure 4.3: Portion of total data that is divided into the training, validation and testing sets respectively for different values of $x_{sparsity}$.

$x_{sparsity}$	**Training**	**Validation**	**Testing**
1	1	0	0
2	1/4	1/4	1/2
3	1/9	1/9	7/9
4	1/16	1/16	7/8
5	1/25	1/25	23/25
10	1/100	1/100	98/100
20	1/400	1/400	398/400

coverage. The one potential bias this gives is for structures appearing in a grid themselves, but we have no reason to believe these are prevalent in terrain data.

There is another problem with this selection of data however, namely flexibility. Where as a random split grants fine tuned control of how large each subset should be, while as this grid gap parameter $x_sparsity$ is very course. This could be improved by making a more advanced function that selects a specific amount for training and validation per grid area, but we have not prioritised this.

Implementing this strategy is done in three steps:

1. Select training set from a grid with some gap $x_{sparsity}$
2. Select validation set from a grid with the same gap, but offset so that

these points come in the middle between the training set datapoints

3. Use the rest of the data as a testing set

Files used for data creation and storing are `/python/data_handling.py`, `/python/franke.py` and `/python/real_terrain.py`, with `/python/test_data_handling.py` for testing.

4.2 Models and training

To implement the models, we made two classes, one for ordinary linear regression, and one for regularized versions, with the latter inheriting the predict-function from the first, and taking in a function for beta. The classes are quite simple, with the two main methods being fit and predict.

The files used for the models classes is `/python/linear_models.py` and `/python/test_linear_models.py`.

4.3 Resampling

To resample, we have implemented two functions; one for bootstrap and one for kfold cross-validation.

The bootstrap function is not vectorised, and works by making an array of equal length to X_{train}, but with random indeces from them, and then using those indeces to sample X_{train} and y_{train}.

```
1     for r in range(R):
2         boot_mask = np.random.randint(0, n, n)
3         X_boot = X_train[boot_mask,:]
```

Our kfold cross-validation algorithm is also very simple. However, it starts by selecting the data it will use in its folds from the training and validation datasets, and selects only as much data as it needs to make the total size of the training folds equalto the size of the original training set. This was in an attempt to make it and bootstrap perform more similarly, to get MSEs of the

same scale, but was not very successful. Likely because bootstrap on picks only a subset of the training set every time, its MSE is consistently higher. The variance of our kfold techniques MSE estimate also seems heightened by picking a different set to train and validate everytime, but we also expect its bias to go down because any possible data bias is lowered due to it ot using the same data every time.

The file used for resampling is `/python/resampling.py`.

4.4 Visualisation

For visualisation, we have made two functions. One for plotting n 3-dimensional plots in a grid, and the other for making different kinds of plots showing the relations between validation errors, maximum polynomial degree, and λ.

The first one is mainly useful for comparing predicted and actual terrain visually, and in addition to creating still images it can also make an animation spinning the plot around.

The second one is used for understanding the rich validation errors better, and understand what hyperparameters works best. It can make three different kinds of plots:

1. An animation with lineplots showing the bootstrap estimated bias, variance and MSE of a model against λ, and with different maximum polynomials as time.
2. An animation with lineplots showing the bootstrap estimated MSE and kfold estimated MSE of a model against λ, with different maximum polynomials as time
3. Two still grid plots with λ against polynomials, one for kfold estimated MSE, and one for bootstrap estimated MSE.

The file used for visualisation is `/python/plotting.py`.

4.5 Validating, tuning and testing

In order to make our design matrixes we have a function that we think works particularly well. Because we want to emphasize hyperparameter optimalisation, but don't have access to any computing power other than the one our laptops can give us, we need to make efficient functions. The creation of a new design matrix is done once for every polynomial, and it's expensive for both large datasets and high maximum polynomials. To make an efficient function, we vectorised it, mitigated unnecessary recomputations, and in the process got very close to making it work for arbitrary input dimensions.

Algorithm 1 Pseudocode for polynomial design matrix function

procedure DESIGN MATRIX(maxDegree, x)
  $powerCombinations \leftarrow \{(p_1, p_2) \,|\, p_1 + p_2 \leq maxDegree \wedge p_1, p_2 \geq 0\}$
  $xPowers \leftarrow \{(x^p) \,|\, p \leq maxDegree \wedge p \geq 0\}$
  for $(p_1, p_2) \in powerCombinations, n$ **do**
    $X_n \leftarrow xPowers^{p_1} \cdot xPowers^{p_2}$
  end for
  return X
end procedure

The arguably most important part of this project is the tuning class, which iterates through different hyperparameters, uses the resampling techniques to estimate our error well, and stores all the data. It also computes optimal parameters as decided by the different resampling techniques, and tests it on the plentiful test data. The results this class produces, as well as the class itself and ways to improve it, is the main focus of the results and conclusion parts.

Chapter 5

Verifying our implementation on the Franke function

Before we apply our methods to real terrain data, let's verify that it works as intended on the Franke function data. The code for this is in `/python/franke_verifying.py`.

5.1 Small dataset

Firstly, we want to check that we can induce overfitting, by using a small dataset, and allowing some complexity. In fig 5.1 it is easy to see that the model continues fit better to the training set for higher complexity, but quickly starts overfitting, and doing worse on the test set. The model actually does slightly better on the test set than the training set for low polynomial degrees, but this is just a coincidence that isn't that unlikely since the sets are so small.

5.2 β parameters

Next, we want to look at the β parameters made by the last model whose error is plotted above, namely OLS with $p = 5$. It has 21 parameters, and

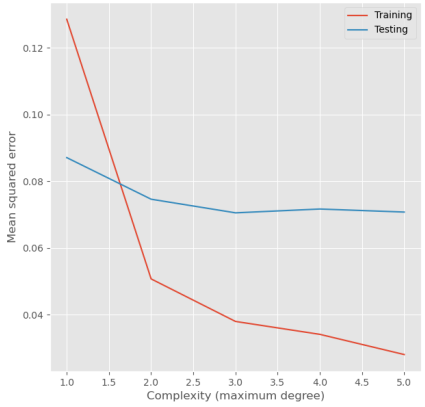

Figure 5.1: OLS models trained on only 49 datapoints, and tested on twice as much. Data from Franke function with noise.

their standard deviation σ can be computed by $\sigma_\beta = \sigma_y^2(X^TX)^{-1}$, where σ_y^2 is the variance of the noise in y. This can be estimated by the mean squared error of y. It's not a good estimator, not even unbiased, but for this all we need is a rough idea of σ_β^2. Combining this, we can get the confidence intervals by assuming a normal distribution of β values. In fig 5.2 we have plotted the interval for the different β parameters with 99% confidence.

5.3 Resampling

Next on the list of processes we want to verify works, is resampling. The data is scarce here, so resampling should give us better estimates of MSE, and also allow us to inspect the previously discussed bias-variance trade-off.

5.3.1 Bias-variance trade-off

Let us see if we can visualise thetrade-off with the data from the Franke function, and the bootstrap resampling technique. Our bootstrap function returns model variance and bias, as well as total mean squared error. We plot that data against different complexity scores, and look at it for ordinary

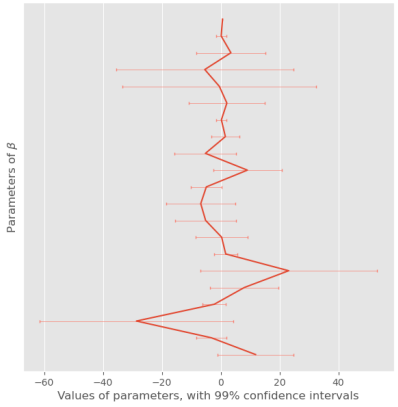

Figure 5.2: 99% confidence interval for each β parameter.

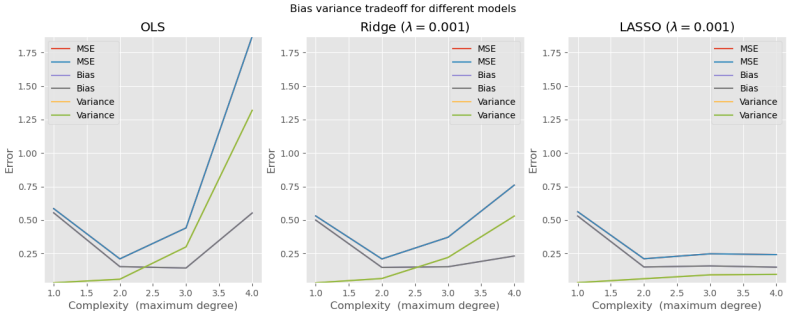

Figure 5.3: Bias, variance, and MSE estimated by bootstrap, for different models trained on the Franke dataset. There is a very strange issue with our code that makes the labels repeated, we're sorry for that.

linear regression, as well as for regularised Ridge and LASSO regression with a medium regularisation parameter λ. The results can be seen in fig 5.3.

If we start by examining the OLS plot, it shows that for low complexity, the bias is high but the variance is small. This makes sense because low degree polynomials have few degrees of freedom, so they can't really vary that much. Bias is big because they probably can't fit to the data that well either though. With some more complexity the bias and variance meet in the middle, before they both shoot up, with the variance growing quicker than the bias. Here, the models have a lot of options as to how to bend,

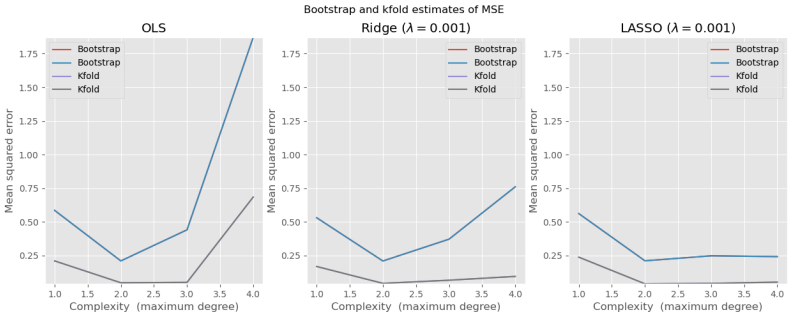

Figure 5.4: MSE estimated by bootstrap and k-fold resampling techinques, for different models trained on the Franke dataset.

and the way they bend is therefore highly dependent on which data it gets.

For the regularised plots, the tendencies is similar, but less extreme. The regularisation parameters keep the amount of overfitting down, and acts a lot more skeptical about believing the wild claims about the complex nature of the data than OLS does. LASSO does better than Ridge here, but this isn't enough data to conclude on anything, and we also haven't tested for different λ values yet.

5.3.2 Cross-validation

Lastly, let us check that our k-fold cross-validation technique performs similarly to our already tested bootstrap method. If they don't, that's a good indication that something is wrong with one of them. We have therefore plotted errors for the same models as above, just with bootstrap and k-fold estimates of MSE.

5.4 Results from tuning

Now that all the individual parts have been tested, we are ready to try out the big Tune class, and see how well it is able to fit to the data. We won't be analysing it much, only explaining the methodology. In the next two chapters we'll look at the same results as the ones we're about to present, just for real terrain data, and we'll do that much more in depth.

Figure 5.5: Hyperparameters tried out when tuning the models on the Franke function.

$p \in \{1, 2, \ldots, 15\}$
$\lambda_{Ridge} \in \{10^i \mid i \in \{-8, -7, -6, \ldots, 1\}\}$
$\lambda_{LASSO} \in \{10^i \mid i \in \{-24/16, -21/16, -18/16, -15/16, \ldots, 0\}\}$

To explain our key plots, and discuss our methodology, but not use too much time on it, we are narrowing down this section to just cover the results from the bootstrap technique on the Ridge model. In fig 5.6 is a visualisation of the errors k-fold estimated during the validation phase, for different set of hyperparameters. In table 5.7 are the optimal parameters for each model estimated by the different techniques, with average just being the average of k-fold and bootstrap. Which model is optimal is just selected by the single lowest estimated MSE. One of the weaknesses with this is that the variance for estimation is now an important contributor to deciding which model is best. As seen in the grid search figure, there are many parameter-sets with very similar error metrics, so which one of them is the lowest is highly dependent on chance. A better way to do this would be to test out the candidates with the lowest errors some more times, and also test out the neighbouring region with a finer selection of parameter values, perhaps even using gradient decent methods. This would reduce variance, and also allow us to fine-tune the model the places where we are likely to find a good one, and not use the same parameter resolution everywhere. Perhaps a meta-reinforcement learning algorithm could do this well.

In fig 5.8 is the model the bootstrap deemed optimal, tested on the testing set. Visually, the model looks like a good fit to the underlying structure, even though it seems as if a polynomial of higher degree should be able to capture the smaller peak in the middle of the plot as well.

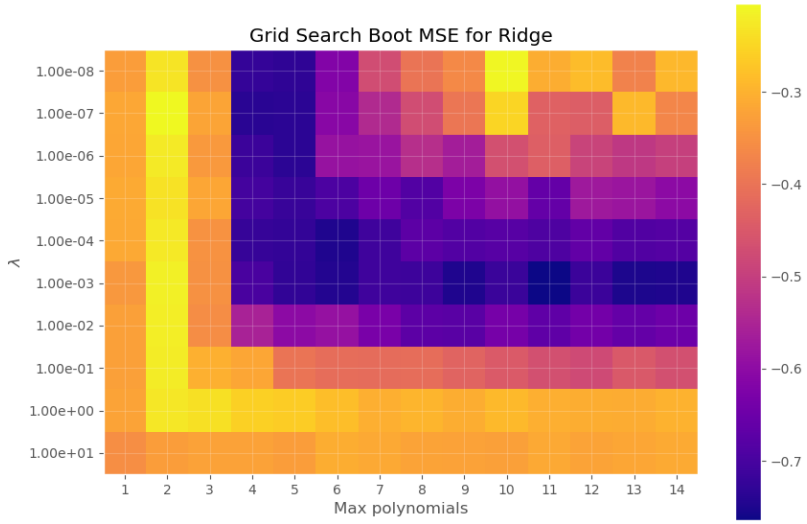

Figure 5.6: Bootstrap grid search for best λ and polynomial hyperparameters for the Ridge model, on the franke data. Complexity is lowest in the bottom left, and highest top right.

Figure 5.7: Optimal models as estimated by k-fold, bootstrap, and the average of them. In the rightmost column is the mean squared error of the model with those paramaters trained on the training set, and tested once on the testing set.

Model	Resampling technique	Lambda	Poly	Validation	Test
Ridge	Bootstrap	1.000e-03	11	1.721e-01	4.525e-02
	Kfold	1.000e-02	1	4.652e-02	6.325e-02
	Average	1.000e-03	11	1.135e-01	4.525e-02
Lasso	Bootstrap	1.389e-01	3	1.361e-01	1.131e-01
	Kfold	3.162e-02	10	4.944e-02	6.522e-02
	Average	8.483e-02	12	1.097e-01	8.316e-02
OLS	Bootstrap	0.000e+00	4	1.826e-01	4.675e-02
	Kfold	0.000e+00	1	4.842e-02	6.325e-02
	Average	0.000e+00	4	1.168e-01	4.675e-02

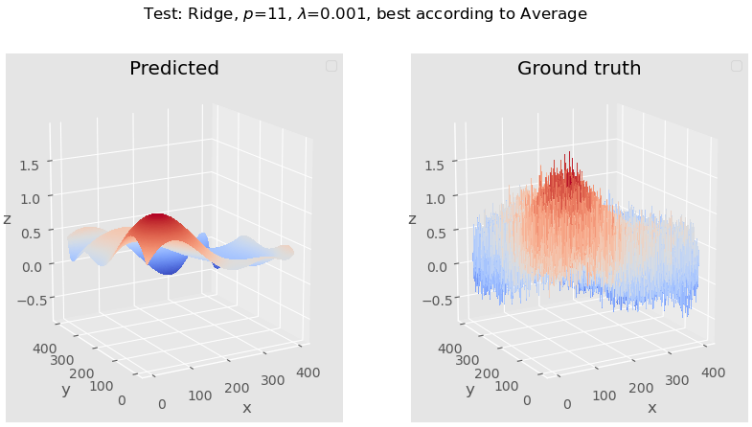

Figure 5.8: Terrain predicted on franke function testing data by Ridge with $\lambda = 10^{-8}$ and with maximum degree of 3. These parameters are suggested by bootstrap as the best model.

Chapter 6

Results on real terrain data

When using the tuning class discussed in the previous chapter, with the same parameters as we tried on the Franke function (table 5.5), these are the results that we get on our real terrain data.

As demonstrated by both fig 6.1, fig 6.2, and table 6.3, the real data encourages higher polynomial degrees, but roughly the same λ values as the Franke function generated data did. Something that is also even more visible here than on the Franke data, is how different the estimates for error are between bootstrap and k-fold. Most of the time, they mark out the same regions of parameter sets as relatively good or bad, but give estimates of how erronous models with those parameters are a magnitude apart. This can probably be attributed mostly to the different data they are getting trained on. While each bootstrap sample only contains a subset of the training set, out k-fold function makes it so that almost all the training and validation data is available for each model. The different amounts of data explains the k-fold estimated errors tendency to be lower. The fact that our k-fold estimate is based on data the bootstrap trained models are only evaluated on, likely explains the other discrepancies. See `/plots/animations/franke_tune_Lasso_bootstrapKfold.mp4` for an example of very different k-fold and bootstrap estimates.

Another difference between our resampling techniques is that k-fold seems

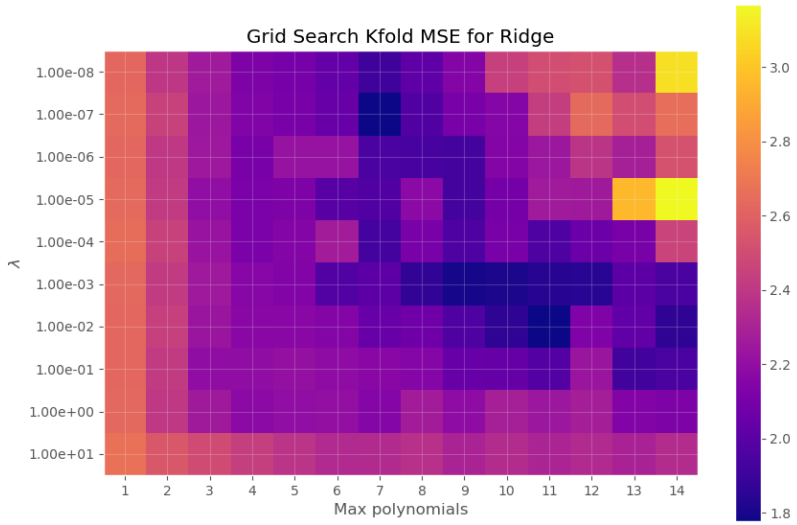

Figure 6.1: K-fold grid search for best λ and polynomial hyperparameters for the Ridge model, on the real terrain data.

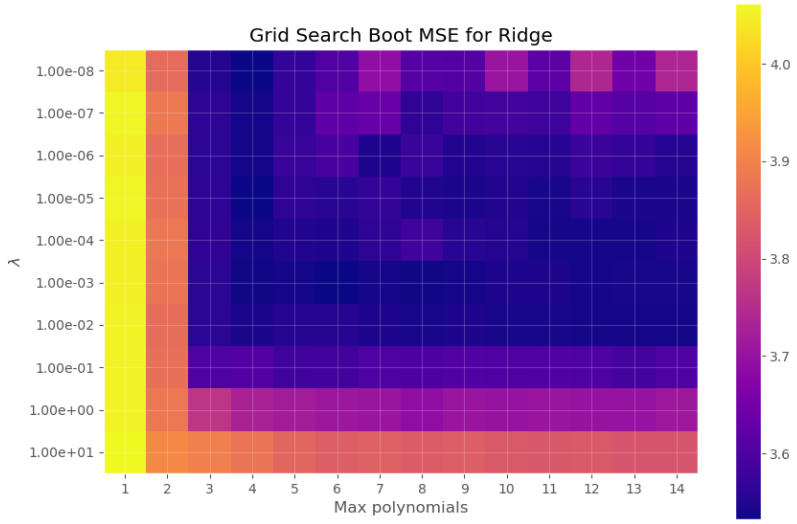

Figure 6.2: K-fold grid search for best λ and polynomial hyperparameters for the LASSO model, on the real terrain data.

Figure 6.3: Optimal models as estimated by k-fold, bootstrap, and the average of them. In the rightmost column is the mean squared error of the model with those paramaters trained on the training set, and tested once on the testing set.

Model	Resampling technique	Lambda	Poly	Validation	Test
Ridge	Bootstrap	1.000e-08	4	3.406e+03	6.434e+02
	Kfold	1.000e-07	7	5.984e+01	4.358e+02
	Average	1.000e-03	8	1.753e+03	4.164e+02
Lasso	Bootstrap	3.162e-02	6	3.911e+03	5.562e+02
	Kfold	3.162e-02	14	8.845e+01	5.519e+02
	Average	3.162e-02	6	2.038e+03	5.562e+02
OLS	Bootstrap	0.000e+00	4	3.437e+03	6.434e+02
	Kfold	0.000e+00	7	5.935e+01	4.545e+02
	Average	0.000e+00	4	1.790e+03	6.434e+02

to have a higher variance. The values seem to follow much less of a continous function, with our heatmaps showing single parameter sets with completely different error estimates than the ones around. We have already hypothesised that (our implementation of) k-fold should have a higher variance than bootstrap with our implementation, so this strengthens that hypothesis.

Comparing the test scores for for different model architectures, with the hyperparameters selected with the average of the bootstrap and the k-fold validation data, Ridge does the best here. LASSO comes a bit further behind, followed by OLS. LASSO might be at a disadvantage her because it seems as if lower λ values benefitted it a lot, but we struggled with getting it to converge when we used $\lambda \in [\leftarrow, 10^-2]$.

6.1 Validation error vs test error

The hyperparameter optimimisation procedure established seems to works well. It is able to find hyperparameters λ and max degree for both Franke function generated data, and one example of real terrain data, that makes

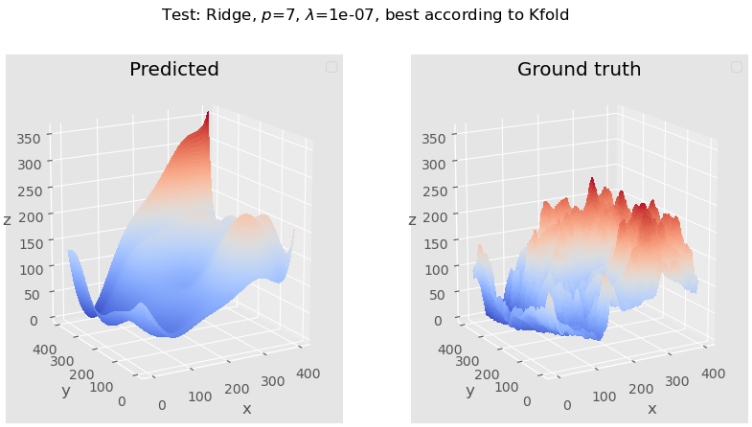

Figure 6.4: Terrain predicted on real terrain testing data by ridge with $\lambda = 10^{-3}$ and with maximum degree of 8. These parameters are suggested by k-fold as the best model.

polynomials fit the data. However, we have so far only looked at the models it deemed best, and have not looked at the ones it discarded. It very well might be that these models are just above average, and that random hyperparameters can do almost as well on unseen data. To discuss this, we need to look at the relation between the validation mean squared errors of our bootstrap and k-fold functions, and mean squared error on test data. Plotting this for the Franke function, we get fig 6.5 for the Franke function data, and fig 6.6 for the real terrain data. Please note that we have 160 000 datapoints in total, but that because we used $x_{sparsity} = 20$, only 800 of those were used in training and validation. The test errors are therefore expected to be a very good estimate of the actual error.

The figures look quite different. While as the Franke function validation errors seem to poorly represent the test errors, the tuning for models on real terrain data fits pretty well. In the latter, most of the points are in the diagonal line going from little validation error with both resampling methods to high validation error with both, and in this series the test errors seem to verify that they were mostly correct. There are points outside of this main series, where either k-fold or bootstrap has seen a structure the other one hasn't. Trying to fit a linear regression model to this data has noe been

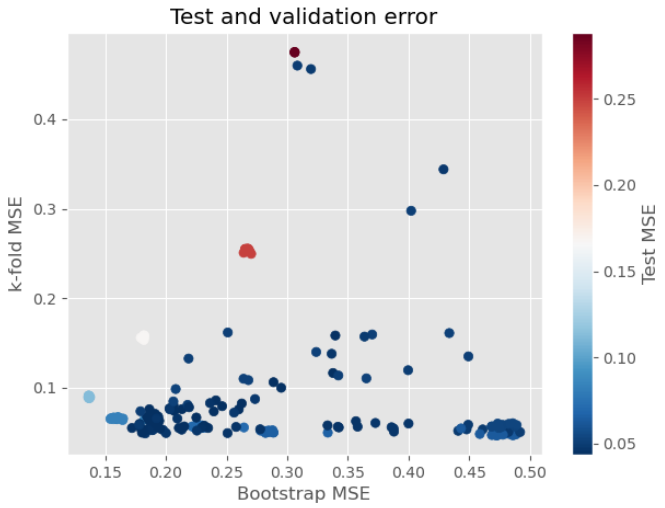

Figure 6.5: Franke terrain data: Validation errors calculated through resampling, with the colour of the points indicating their respective test error. Note that we have removed some outlier values.

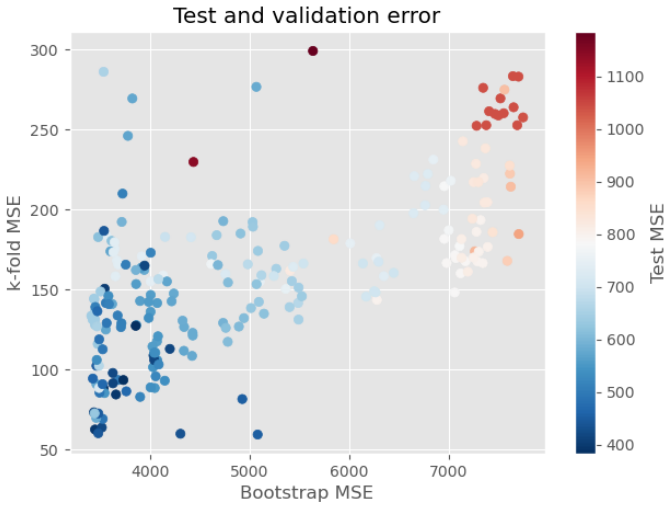

Figure 6.6: Real terrain data: Validation errors calculated through resampling, with the colour of the points indicating their respective test error. Note that we have removed some outlier values.

succesful, but we have little data.

The reason for the poorer performance on the Franke function data is likely because this data is very noisy, and therefore difficult to fit to. There is naturally a high variance between models.

All in all, it seems as if our validation is useful in predicting which models will do good, but far from perfect. After seeing this data, it is also hard to justify going back and changing the validation code, as that would make the test data less unseen, and risk having us fitting the models to it implicitly.

Chapter 7

Conclusions

The implementation done here of linear regression is able to fit to terrain data, both synthetic and actual, well. Resampling techniques allows a decent idea of how a model would perform on unseen data, without requiring much data itself. With this validation procedure, the tuning class is able to pick out models that perform well, that don't overfit, but that also have a sufficient complexity. This is also due to regularisation techniques that discourage fitting to weak structures, which the difference between OLS and regularised linear regression here shows often to be false structures. The best model architecture for the data we've tested here has consistently been Ridge, with LASSO performing second best on the real terrain data, and OLS on the Franke function data.

Further work could look into a better tuning class that didn't use the same resolution everywhere, but explored areas of low validation errors further, resampling them more and trying out neighbouring hyperparameters.

It would also be interesting looking further into the relationship between test and validation error. Both looking at how much is really gained by resampling, which we have not discussed here, and seeing how well this implementation can predict test error for other datasets.

Chapter 8

Referencing

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition.* Springer Series in Statistics. Springer New York, 2009. ISBN: 9780387848587.

Hjorth-Jensen, M. *Lecture notes, FYS-STK3155* 2020.