Report for project 1

Adam P W Sørensen

October 9, 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

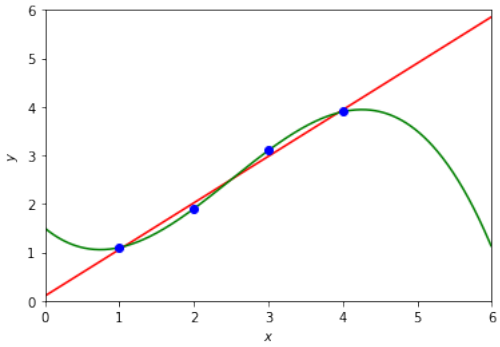
This project is concerned with the following problem: Suppose we know the values of a function f on a set of points $\{x_1, x_2, \ldots, x_n\}$, can we then predict $f(x)$ for any x? In full generality this is plainly impossible (though see [Hardin and Taylor, 2008]). But if we allow ourselves to believe 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`. A print of the jupyter notebook is also attached at the end of this report.

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

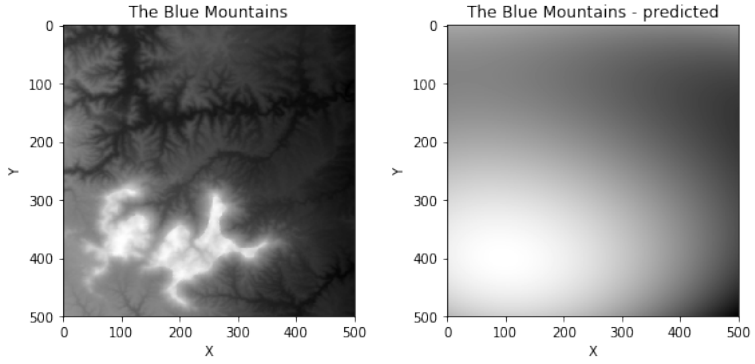
For the terrain data I chose a picturesque mountain range just outside of Sydney called the Blue Mountains. I again used cross validation to select the model with the smallest predicted error (code block 68), and I got Ridge regression for a third degree polynomial with a λ parameter of 0.5. Running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

Code for project 1

October 9, 2018

1 Project 1 - code

It is all meant to be run in order

1.1 A whole heap of setup

1.1.1 Imports

These appear to be the standard imports

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        from matplotlib import cm
        from matplotlib.ticker import LinearLocator, FormatStrFormatter
        from random import random, seed
        from sklearn.linear_model import LinearRegression, Ridge, Lasso
        from sklearn.metrics import mean_squared_error, r2_score
        from sklearn.preprocessing import PolynomialFeatures
        from sklearn.model_selection import cross_val_score
        from imageio import imread
```

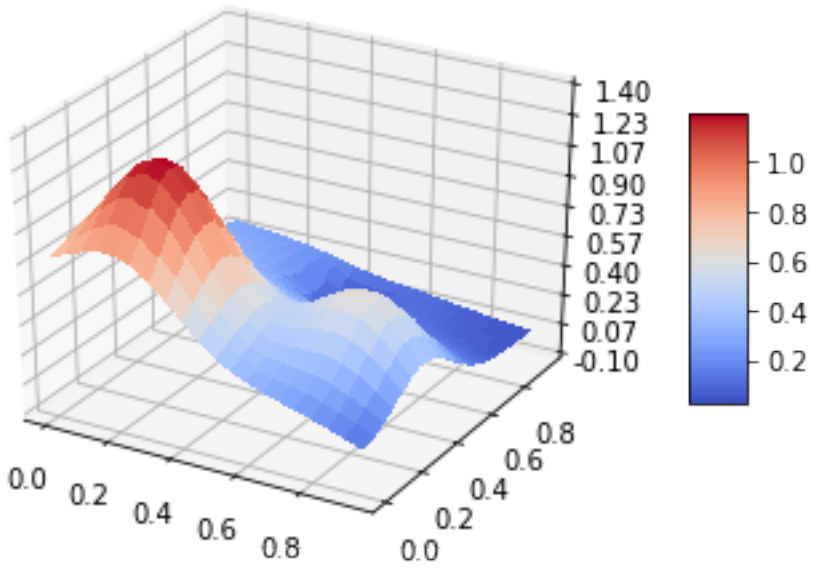
1.1.2 The Franke Function

We define the Franke Function, and do it plot of it. All of that is from the project description.

The only non-standard thing we do, is define a function Franke_on_row, which is just a wrapper to apply the Franke Function to a list with two elements.

```
In [2]: fig = plt.figure()
        ax = fig.gca(projection='3d')

        # Make data.
        fx = np.arange(0, 1, 0.05)
        fy = np.arange(0, 1, 0.05)
        fx, fy = np.meshgrid(fx,fy)


        def FrankeFunction(x,y):
```

```
    term1 = 0.75*np.exp(-(0.25*(9*x-2)**2) - 0.25*((9*y-2)**2))
    term2 = 0.75*np.exp(-((9*x+1)**2)/49.0 - 0.1*(9*y+1))
    term3 = 0.5*np.exp(-(9*x-7)**2/4.0 - 0.25*((9*y-3)**2))
    term4 = -0.2*np.exp(-(9*x-4)**2 - (9*y-7)**2)
    return term1 + term2 + term3 + term4

# A wrapper to let us apply FrankeFunction to a list of the form row = [x, y]
def Franke_on_row(row):
    return FrankeFunction(row[0], row[1])

fz = FrankeFunction(fx, fy)

# Plot the surface.
surf = ax.plot_surface(fx, fy, fz, cmap=cm.coolwarm, linewidth=0, antialiased=False)

# Customize the z axis.
ax.set_zlim(-0.10, 1.40)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```



1.1.3 Regression methods

We define the classes polynomialOLS, polynomialRidge, and polynomialLasso. Each will take a degree, and the later two also a parameter lamb. They each implement train and predict functions. The idea is that one initiates a polynomial method pm with a given degree, then we can simply pass a two-column matrix XY with x and y values to train and predict, the classes themselves will then call polymatrix to get a matrix with all the terms needed to get a polynomial of the given degree. The Ridge regression class also takes care to center the input. That is all just to make calling the functions easier.

Each class also has plotfunc, coefficients and info methods.

The plotfunc is simply a wrapper used to predict on a single pair of points (x,y). I use it because I had problems with applying predict when plotting.

The coefficients methods returns a column vector consisting of the beta functions.

The info method does some printing we use later to make nice output.

```
In [3]: # We are only going to work with input in two dimensions
        # and output in one.
        # Hence all our code will be writting with this assumption

        # Given a matrix XY with two columns x,y return an array with columns x, y, x^2, xy, y^2
        def polymatrix(XY, n):
            X = XY[:,[0]]
            Y = XY[:,[1]]
            ones = np.ones((len(X), 1))
            pl = np.concatenate([(np.power(X,k-i)*np.power(Y,i)) for k in range(1, n+1) for i in
            return np.hstack((ones, pl))


        # This is a class for doing the most basic OLS regression
        class myOLS :

            # We have one class virable beta which has the coefficients for the linear regressio
            def __init__(self):
                self.beta = 0

            # This method sets beta on the given training matrix xb with know values y.
            def train(self, xb, y):
                self.beta = np.linalg.inv(xb.T.dot(xb)).dot(xb.T).dot(y)

            # Use our beta to predict on a (correctly formatted) matrix xb.
            def predict(self, xb):
                return xb.dot(self.beta)

        # This class is simply a wrapper for the myOLS class
        # It simply takes care of transforming the raw input into proper polynomial input
        class polynomialOLS :

            def __init__(self, n):
```

```
        self.polyOLS = myOLS()
        self.degree = n

    # Train our polynomial OLS.
    # We take matrix XY with two columns x,y and given zs as output.
    # First we use polymatrix to transform XY to matrix with rows 1, x, y, x^2, xy, y^2
    # Then we feed that as the training data to an ordinary OLS
    def train(self, XY, z):
        xb = polymatrix(XY, self.degree)
        self.polyOLS.train(xb,z)

    # Predict our polynomial solution
    # As in train we take a matrix XY with two columns x,y.
    # First we use polymatrix to transform XY to matrix with rows 1, x, y, x^2, xy, y^2
    # Then use that as input to predict on an ordinart OLS
    def predict(self, XY):
        xb = polymatrix(XY, self.degree)
        return self.polyOLS.predict(xb)

    # This is only used for plotting.
    # It is just a wrapper function to predict the value a single (x,y) pair
    # Given two numbers x,y it just applies predict to an array with x and y as columns
    def plotfunc(self, x, y):
        return self.predict(np.array([[x, y]]))

    # Prints the coefficients beta
    def coefficients(self):
        return self.polyOLS.beta

    # A function to print some pretty information about this regression model
    def info(self):
        print("OLS regression, trying to fit a polynomial of degree %d " % self.degree)

class polynomialRidge :

    def __init__(self, n, lamb):
        self.degree = n
        self.lamb = lamb
        self.beta = 0
        self.intercept = 0

    # takes a column or row vecter x, and returns x - bar{x},
    # where bar{x} is the avarage of the entries in x
    # Also works on any size matrix, but that is not what we have in mind
    def center(self, x) :
        return x - np.mean(x)

    # Train our polynomial ridge regression.
```

```
        # We take matrix XY with two columns x,y and given zs as output.
        # First we transform XY to matrix with columns x, y, x^2, xy, y^2 ...
        # Then we center each row.
        # Finally wecompute beta according to formula from Hastings (3.44)
        def train(self, XY, z):
            xb = polymatrix(XY, self.degree)
            xb = np.delete(xb, 0, 1) # removes the column of 1s
            xb_centered = np.apply_along_axis(self.center, 0, xb) # center each column
            xbrows = len(xb[0,:]) # number of rows in xb
            b0 = self.center(z)
            self.beta = (np.linalg.inv(xb_centered.T.dot(xb_centered) + self.lamb*np.identit
            self.intercept = np.mean(z) - (np.mean(xb, axis = 0)).dot(self.beta)

        # Predict our polynomial solution
        # As in train we take a matrix XY with two columns x,y.
        # First we transform XY to matrix with columns x, y, x^2, xy, y^2 ...
        # Then use beta and intecept to predict the value
        def predict(self, XY):
            xb = polymatrix(XY, self.degree)
            xb = np.delete(xb, 0, 1) # removes the column of 1s
            return xb.dot(self.beta) + self.intercept

        # This is only used for plotting.
        # It is just a wrapper function to predict the value a single (x,y) pair
        # Given two numbers x,y it just applies predict to an array with x and y as columns
        def plotfunc(self, x, y):
            return self.predict(np.array([[x, y]]))

        # Returns the coefficients beta (a long with intercept) as a column vector
        # vstack puts the intercept ontop of the other betas
        def coefficients(self):
            return np.vstack(([self.intercept], self.beta))

        # A function to print some pretty information about this regression model
        def info(self):
            print("Ridge regression, trying to fit a polynomial of degree %d " % self.degree
            print("Lambda = %.4f" % self.lamb)

    class polynomialLasso :

        def __init__(self, n, lamb):
            self.lasso = Lasso(alpha = lamb)
            self.degree = n
            self.lamb = lamb

        # Train our polynomial lasso.
        # We take matrix XY with two columns x,y and given zs as output.
        # First we transform XY to matrix with columns x, y, x^2, xy, y^2 ...
```

```
        # Then we center each row.
        # Finally wecompute beta according to formula from Hastings (3.44)
        def train(self, XY, z):
            xb = polymatrix(XY, self.degree)
            xb = np.delete(xb, 0, 1) # removes the column of 1s
            self.lasso.fit(xb, z)

        # Predict our polynomial solution
        # As in train we take a matrix XY with two columns x,y.
        # First we transform XY to matrix with columns x, y, x^2, xy, y^2 ...
        # Then use beta and intecept to predict the value
        def predict(self, XY):
            xb = polymatrix(XY, self.degree)
            xb = np.delete(xb, 0, 1) # removes the column of 1s
            return np.c_[self.lasso.predict(xb)]

        # This is only used for plotting.
        # It is just a wrapper function to predict the value a single (x,y) pair
        # Given two numbers x,y it just applies predict to an array with x and y as columns
        def plotfunc(self, x, y):
            return self.predict(np.array([[x, y]]))

        # Returns the coefficients (including intercept) as a column vector
        # reshape(-1,1) makes the array of coef_ into a column vector
        # we put intercept_ on top of it.
        def coefficients(self):
            # resha
            return np.vstack((self.lasso.intercept_, self.lasso.coef_.reshape(-1,1)))

        def info(self):
            print("Lasso regression, trying to fit a polynomial of degree %d " % self.degree
            print("Lambda = %.4f" % self.lamb)
```

1.1.4 Test of regression functions

We verify that the classes we build yield the same outcome as the build-in functions of sci-kit learn.

```
In [4]: print("Test of regression methods and polymatrix")
        print("-------------------")
        print("")
        print("Testing polymatrix. Making first a random (4,2) array")
        test_XY = np.random.rand(100,2)
        test_z = np.c_[np.apply_along_axis(Franke_on_row, 1, test_XY)]
        #print(test_XY)
        print("")

        for test_degree in range(1,6) :
```

```
        print("For degree %d the mean diffrence between polymatrix and PolynomialFeatures ar
        poly = PolynomialFeatures(test_degree)
        print(np.mean(polymatrix(test_XY, test_degree) - poly.fit_transform(test_XY)))
        print("")

    print("--------------------")
    print("")
    print("Testing how my ols and ridge methods differ from the sci-kit learn ones")
    print("")
    for test_degree in range(1,6) :

        print("We look at a polynomial of degree %d" % test_degree)
        poly = PolynomialFeatures(test_degree)
        poly_XY = poly.fit_transform(test_XY)
        # We remove the leading column of 1s
        # This is done since we call teh fit functions without setting fit_intercept to fals
        # So in essence, the fit of OLS and more importantly Ridge makes sure to find the in
        poly_XY = np.delete(poly_XY, 0, 1)


        # Setup "my" OLS regression
        myols = polynomialOLS(test_degree)
        myols.train(test_XY, test_z)
        # Setup sci-kit learns OLS regression
        scikit_ols = LinearRegression()
        scikit_ols.fit(poly_XY, test_z)
        # make a column vector of betas for scikit_ols
        skols_betas = np.vstack((scikit_ols.intercept_, scikit_ols.coef_.reshape(-1,1)))
        print("Mean difference of hand computed intercept and coefs for OLS")
        print(np.mean(myols.coefficients() - skols_betas))

        # Setup "my" ridge regression
        myridge = polynomialRidge(test_degree, 0.5)
        myridge.train(test_XY, test_z)
        # Setup sci-kit learns ridge regression
        scikit_ridge = Ridge(alpha = 0.5)
        scikit_ridge.fit(poly_XY, test_z)
        skridge_betas = np.vstack((scikit_ridge.intercept_, scikit_ridge.coef_.reshape(-1,1)
        print("Mean difference of hand computed intercept and coefs for Ridge")
        print(np.mean(myridge.coefficients() - skridge_betas))
```

```
Test of regression methods and polymatrix
--------------------

Testing polymatrix. Making first a random (4,2) array

For degree 1 the mean diffrence between polymatrix and PolynomialFeatures are
```

```
0.0

For degree 2 the mean diffrence between polymatrix and PolynomialFeatures are
0.0

For degree 3 the mean diffrence between polymatrix and PolynomialFeatures are
2.524344530066211e-19

For degree 4 the mean diffrence between polymatrix and PolynomialFeatures are
3.493005673427315e-19

For degree 5 the mean diffrence between polymatrix and PolynomialFeatures are
2.3215042611641024e-19

-------------------

Testing how my ols and ridge methods differ from the sci-kit learn ones

We look at a polynomial of degree 1
Mean difference of hand computed intercept and coefs for OLS
5.181040781584064e-16
Mean difference of hand computed intercept and coefs for Ridge
3.700743415417188e-17
We look at a polynomial of degree 2
Mean difference of hand computed intercept and coefs for OLS
-5.782411586589357e-16
Mean difference of hand computed intercept and coefs for Ridge
7.517135062566164e-17
We look at a polynomial of degree 3
Mean difference of hand computed intercept and coefs for OLS
5.004885395010206e-14
Mean difference of hand computed intercept and coefs for Ridge
-4.163336342344337e-18
We look at a polynomial of degree 4
Mean difference of hand computed intercept and coefs for OLS
-1.777851939740079e-12
Mean difference of hand computed intercept and coefs for Ridge
-3.7007434154171884e-18
We look at a polynomial of degree 5
Mean difference of hand computed intercept and coefs for OLS
3.660379605239345e-11
Mean difference of hand computed intercept and coefs for Ridge
-5.683284530819253e-17
```

1.1.5 Cross validation

```
In [5]: # This function does k-fold cross validation
        # It takes a k, a matrix XY with two columns of inputs, and corresponding true values z
        # It also takes a model which we assume have train, predict, and coefficients functions.
        # Both should work on these matrices
        # We return the expected error, and a list of computed coefficients for our model
        def cross_validation(k, XY, z, model):

            sqdiffs = []
            clength = len(XY[:,0]) # the length of the first column in XY
            permutation = np.random.permutation(clength) # a permutation of the indexes of rows
            partitions = np.array_split(permutation, k) # The permutation is devided in to k alm

            for i in range(0,k) :
                # create a mask to pick everything but the elements in the i'th partition
                mask = np.ones(clength,dtype=bool)
                mask[partitions[i]] = 0
                # now train on everything but the i'th partition
                model.train(XY[mask, :], z[mask, :])
                # make the mask the picks only the elements of the i'th partition
                notmask = np.invert(mask)
                # update the sqdiffs of predicted values and the true values in the i'th partiti
                zpredict = model.predict(XY[notmask, :])
                sqdiffs.append(np.power(zpredict - z[notmask, :],2))

            pred_mse = np.mean(sqdiffs)
            return pred_mse
```

1.1.6 Statitistics functions

Just a small collection of statistics functions we will use to evaluate the preformance of our models

```
In [6]: def MSE(x, y):
            return np.mean(np.power(x-y, 2))

        def r2d2score(ytrue, ypredict):
            truemean = np.mean(ytrue)
            return 1 - (np.sum(np.power(ytrue-ypredict, 2))/np.sum(np.power(ytrue-truemean, 2)))

        # variance of a column of row vector x
        def variance(x):
            return np.mean(np.power(x - np.mean(x), 2))

        # Given input data XY, z, and a model.
        # We first train the model on XY, z.
        # Then we compute the MSE and R2 score of the prediction vs the true values
        # After thinking it over, this just seems boring, so it wont be used
        def basic_tests(XY, z, model) :
```

```
        model.train(XY, z)
        zpredict = model.predict(XY)
        mse = MSE(z, zpredict)
        r2 = r2d2score(z, zpredict)
        model.info()
        print("MSE = %.4f" % mse)
        print("R2 scrore = %.4f" %r2)
        print("----------------")



    # Given a model already trained on some data, and some new data XY, z
    # Compute MSE and R2 scores of the models prediction of the newdata against it true new
    def test_against_new(newXY, newz, model):
        zpredict = model.predict(newXY)
        mse = MSE(newz, zpredict)
        r2 = r2d2score(newz, zpredict)
        model.info()
        print("MSE = %f" % mse)
        print("R2 scrore = %f" % r2)
        print("----------------")
```

1.1.7 Statistics functions test

```
In [7]: print("Testing the basic statistics functions")
        print("")
        # make two lists of random numbers
        # we need the ravel or the build in r2_score behaves in an odd way
        xs = np.random.rand(1,100).ravel()
        ys = np.random.rand(1,100).ravel()
        mse_diff = MSE(xs,ys) - mean_squared_error(xs,ys)
        print("MSE difference between mine and sci-kit learns: %f " % mse_diff)
        r2_diff = r2d2score(xs,ys) - r2_score(xs,ys)
        print("R2 difference between mine and sci-kit learns: %f " % r2_diff)
```

```
Testing the basic statistics functions

MSE difference between mine and sci-kit learns: 0.000000
R2 difference between mine and sci-kit learns: 0.000000
```

1.1.8 BV estimate and confidence intervals

Functions to estimate the bias and variance and confidence intervals of a given model. They are just minor modifications of the confidence interval code, and really one should probably do both (or all 3) in one go.

```
In [8]: # This function tries to estimate the variance and bias of our model
        # It takes a k, a matrix XY with two columns of inputs, and corresponding true values z
```

```
# It also takes a model which we assume have train, predict, and coefficients functions.
# We return the variance and bias of computing the model on k subsets of XY
# when compared to a single "true" set
def BV_estimate(k, XY, z, model):

    clength = len(XY[:,0]) # the length of the first column in XY
    permutation = np.random.permutation(clength) # a permutation of the indexes of rows
    partitions = np.array_split(permutation, k+1) # The permutation is devided in to k+1
    preds_list = [] # this will be a list of the predicted out comes

    # create a mask to pick everything in the k'th partition
    # this is the partition we will compare aginst
    fixed_mask = np.zeros(clength,dtype=bool)
    fixed_mask[partitions[k]] = 1

    for i in range(0,k) :
        # create a mask to pick everything but the i'th partition
        mask = np.ones(clength,dtype=bool)
        mask[partitions[i]] = 0
        mask[partitions[k]] = 0
        # now train on the i'th partition
        model.train(XY[mask, :], z[mask, :])
        # now predict what will happen on the fixed set
        zpredict = model.predict(XY[fixed_mask, :])
        preds_list.append(zpredict)

    # transform from list of column vectors to a single matrix
    # each row contains the prediction of a single xy point
    preds_matrix = np.hstack(preds_list)

    vari = np.mean(np.apply_along_axis(variance, 1, preds_matrix))
    bias = np.mean(np.power(np.mean(preds_matrix - z[fixed_mask, :], axis=1), 2))
    #mse = np.mean(np.power(preds_matrix - z[fixed_mask, :], 2))

    return bias, vari

##usage example:
#pRidge = polynomialOLS(4)
#print("Using 9 folds to predict variance and bias for a polynomial of degree %i." % 2)
#print(BV_estimate(9, XY, z, pRidge))

# This function estimates the variance of the betas
# It takes a k, a matrix XY with two columns of inputs, and corresponding true values z
# It also takes a model which we assume have train, predict, and coefficients functions.
# We return the variance the the betas when computing the model on k subsets of XY
def beta_variance(k, XY, z, model):

    clength = len(XY[:,0]) # the length of the first column in XY
```

```
        permutation = np.random.permutation(clength) # a permutation of the indexes of rows
        partitions = np.array_split(permutation, k) # The permutation is devided in to k alm
        beta_list = [] # this will be a list of the betas for each training session

        for i in range(0,k) :
            # create a mask to pick everything but the elements in the i'th partition
            mask = np.ones(clength,dtype=bool)
            mask[partitions[i]] = 0
            # now train on everything but the i'th partition
            model.train(XY[mask, :], z[mask, :])
            beta_list.append(model.coefficients())

        # transform from list of column vectors to a single matrix
        # each row contains the "same beta" computed from different training session
        beta_matrix = np.hstack(beta_list)

        variances = np.apply_along_axis(variance, 1, beta_matrix)

        return variances

    ##usage example
    #pRidge = polynomialOLS(4)
    #print("Using 10 folds to compute variances of betas for a polynomial of degree %i." % 2
    #print(beta_variance(10, XY, z, pRidge))
```

1.2 Acutally working with the models or solving parts a,b, c

1.2.1 Data for all our test

In [53]:
```
# Setup input data.
# We make numberofpoints points
# the x's and y's are chosen randomly
# the z's according to the fomula and with some noise
number_of_points = 1000
noise = 0.2
XY = np.random.rand(number_of_points, 2) # a matrix of random numbers with 2 columns of

# we have to flip the normal dist array to get the right dimensions
z = np.c_[np.apply_along_axis(Franke_on_row, 1, XY)] + np.c_[noise*np.random.normal(0,1

# Now we make some "clean" data
# Models will never be trained on this, only tested against it
cleanXY = np.random.rand(100, 2)
cleanz = np.c_[np.apply_along_axis(Franke_on_row, 1, cleanXY)] + np.c_[noise*np.random.
```

1.2.2 A, OLS

In [54]:
```
print("Testing OLS methods for polynomial of degrees 1,2,3,4, and 5.")
print("")
print("First we use 10 fold cross validation to decide which model is best.")
print("")

for n in range(1,6) :
    pOLS = polynomialOLS(n)
    pOLS.info()
    cv = cross_validation(10, XY, z, pOLS)
    print("The predicted error is %f" % cv)
    print("")

print("-------------")
print("")
print("Testing 5 degree polynomial model against new data")

# When I ran this to lowest predicted error came from the polynomial of degree 5
# so we will go with that model

# We test the actual statistics against the unseen (clean) data
bestOLS = polynomialOLS(5)
bestOLS.train(XY, z)
print("The coefficients for our best OLS model is:")
print(bestOLS.coefficients())
test_against_new(cleanXY, cleanz, bestOLS)

# Now we compute the bias and variance of our model
# and esitamte the variance of the parameters beta

print("")
print("Using 9 folds to predict variance and bias")
b, v = BV_estimate(9, XY, z, bestOLS)
print("The bias is %f" % b)
print("The variance is %f" % v)
print("")
print("Using 10 folds to compute variances of betas")
print(beta_variance(10, XY, z, bestOLS))
```

```
Testing OLS methods for polynomial of degrees 1,2,3,4, and 5.

First we use 10 fold cross validation to decide which model is best.

OLS regression, trying to fit a polynomial of degree 1
The predicted error is 0.062884

OLS regression, trying to fit a polynomial of degree 2
The predicted error is 0.058462
```

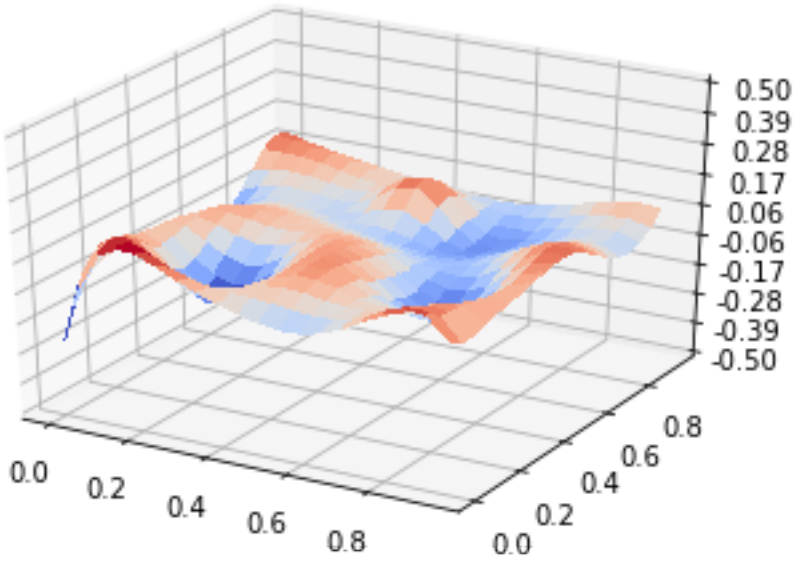
```
OLS regression, trying to fit a polynomial of degree 3
The predicted error is 0.049181

OLS regression, trying to fit a polynomial of degree 4
The predicted error is 0.046228

OLS regression, trying to fit a polynomial of degree 5
The predicted error is 0.044361

-------------

Testing 5 degree polynomial model against new data
The coefficients for our best OLS model is:
[[  0.50487456]
 [  6.84139196]
 [  3.40447963]
 [-28.83003238]
 [-13.51823062]
 [ -8.49877932]
 [ 34.9506793 ]
 [ 36.27449647]
 [ 26.08048924]
 [ -9.02317215]
 [ -8.29829456]
 [-47.10130772]
 [ -3.18784026]
 [-41.11659721]
 [ 32.0059086 ]
 [ -5.18824459]
 [ 19.65607878]
 [  3.94561365]
 [ -1.70731701]
 [ 21.06836119]
 [-18.13575605]]
OLS regression, trying to fit a polynomial of degree 5
MSE = 0.046539
R2 scrore = 0.636927
----------------

Using 9 folds to predict variance and bias
The bias is 0.043213
The variance is 0.000132

Using 10 folds to compute variances of betas
[1.33555253e-03 2.37302168e-01 1.69367710e-01 8.30320758e+00
 1.61226390e+00 3.36708078e+00 4.22657606e+01 4.71033533e+00
 1.22514595e+01 1.76142175e+01 4.01009946e+01 1.26645999e+01
```

```
 7.45089345e+00 1.43587585e+01 2.26653797e+01 5.09400433e+00
 3.50051075e+00 1.36496341e+00 2.68100217e+00 2.12393538e+00
 3.97723740e+00]
```

1.2.3 A - a plot

We do a plot of our predicted model - franke function. If this had been a great model we should have seen a very flat plot, we don't quite

In [55]:
```
# retrain model as it has been modified by BV and variance computations
bestOLS.train(XY, z)

# Do a plot of the best model - franke
fig = plt.figure()
ax = fig.gca(projection='3d')

# Make plot points .
xpoints = np.arange(0, 1, 0.05)
ypoints = np.arange(0, 1, 0.05)
xm, ym = np.meshgrid(xpoints,ypoints)

# Plot the real and predicted surfaces.
diffsurf = ax.plot_surface(xm, ym, np.vectorize(bestOLS.plotfunc)(xm,ym) - FrankeFuncti

# Customize the z axis.
ax.set_zlim(-0.50, 0.5)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

# Add a color bar which maps values to colors.
#fig.colorbar(realsurf, shrink=0.5, aspect=5)

plt.show()
```



1.2.4 B, ridge regression - model selection

```
In [56]: print("Testing Ridge methods for polynomial of degrees 1,2,3,4, and 5.")
         print("")
         print("First we use 10 fold cross validation to decide which model is best.")
         print("")

         #Since printing out all the results of cross validation with different lambdas get tedi
         # we make a list of cv values and corresponding degrees and lambdas.
         # Then we can just sort by the cv value to pick the best model
         cv_list = []

         for n in range(1,6) :
             for a in [0.1, 0.2, 0.5, 1] :
                 pRidge = polynomialRidge(n, a)
                 cv = cross_validation(10, XY, z, pRidge)
                 cv_list.append((n,a,cv))

         cv_list.sort(key=lambda tup: tup[2]) #sort the list by the cv value

         best_tup = cv_list[0]

         print("The best model had a degree %d polynomial and a lambda of %f " % (best_tup[0], b
         print("It had a predicted error of %f" % best_tup[2])

         # NOTE: Since this is not entirely deterministic, we some times get different answers.
         # Mostly I get a degree 4 polynomial with a lambda of 0.1
```

```
Testing Ridge methods for polynomial of degrees 1,2,3,4, and 5.

First we use 10 fold cross validation to decide which model is best.

The best model had a degree 5 polynomial and a lambda of 0.100000
It had a predicted error of 0.049839
```

1.2.5 B - working with chosen model

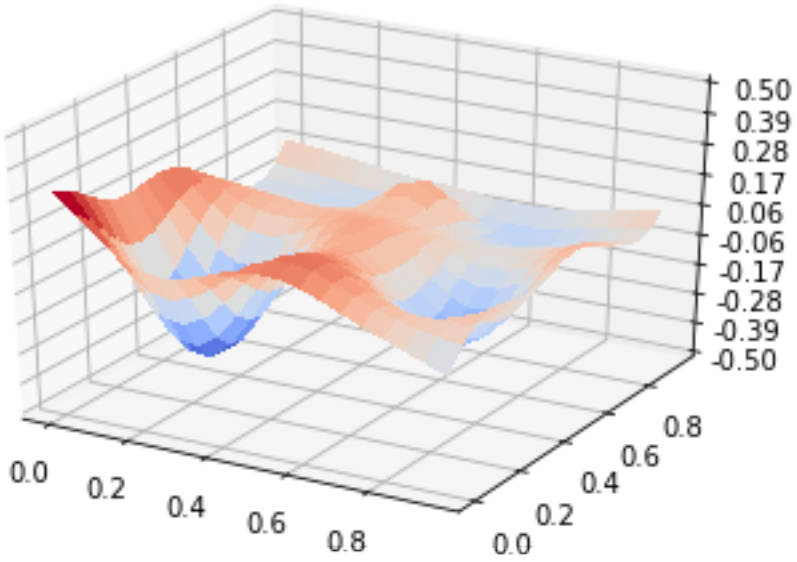
```
In [57]: # We will work with a degree 5 polynomial and a lambda of 0.1
         # We test the actual statistics against the unseen (clean) data
         bestRidge = polynomialRidge(5, 0.1)
         bestRidge.train(XY, z)
         print("The coefficients for our best Ridge model is:")
         print(bestRidge.coefficients())
         test_against_new(cleanXY, cleanz, bestRidge)

         # Now we compute the bias and variance of our model
         # and esitamte the variance of the parameters beta

         print("")
         print("Using 9 folds to predict variance and bias")
         b, v = BV_estimate(9, XY, z, bestRidge)
         print("The bias is %f" % b)
         print("The variance is %f" % v)
         print("")
         print("Using 10 folds to compute variances of betas")
         print(beta_variance(10, XY, z, bestRidge))
```

```
The coefficients for our best Ridge model is:
[[ 1.06612447]
 [-0.43069641]
 [ 0.10905763]
 [-1.5575589 ]
 [ 0.99764565]
 [-2.0988343 ]
 [ 0.70765274]
 [ 0.73417389]
 [-0.66134346]
 [-0.28112356]
 [ 0.89019451]
 [ 0.6225893 ]
 [-0.21970549]
 [-0.32754434]
 [ 0.87460214]
 [-0.6857009 ]
 [-0.22535477]
```

```
 [-0.26390879]
 [-0.08609822]
 [ 0.23636058]
 [ 0.76602987]]
Ridge regression, trying to fit a polynomial of degree 5
Lambda = 0.1000
MSE = 0.058986
R2 scrore = 0.539831
----------------

Using 9 folds to predict variance and bias
The bias is 0.054887
The variance is 0.000056

Using 10 folds to compute variances of betas
[0.00033641 0.00409355 0.00262426 0.00327892 0.00884094 0.00301148
 0.00212812 0.00403362 0.0044825  0.00108847 0.001235   0.00170429
 0.00141348 0.00185033 0.00048559 0.00107178 0.00369705 0.00535533
 0.00346365 0.00359964 0.00189097]
```

1.2.6 B - the plot

```
In [58]: # retrain model as it has been modified by BV and variance computations
         bestRidge.train(XY, z)

         # Do a plot of the best model - franke
         fig = plt.figure()
         ax = fig.gca(projection='3d')

         # Make plot points .
         xpoints = np.arange(0, 1, 0.05)
         ypoints = np.arange(0, 1, 0.05)
         xm, ym = np.meshgrid(xpoints,ypoints)

         # Plot the real and predicted surfaces.
         diffsurf = ax.plot_surface(xm, ym, np.vectorize(bestRidge.plotfunc)(xm,ym) - FrankeFunc

         # Customize the z axis.
         ax.set_zlim(-0.50, 0.5)
         ax.zaxis.set_major_locator(LinearLocator(10))
         ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

         # Add a color bar which maps values to colors.
         #fig.colorbar(realsurf, shrink=0.5, aspect=5)

         plt.show()
```



1.2.7 C, lasso regression - model selection

```
In [59]: print("Testing Lasso methods for polynomial of degrees 1,2,3,4, and 5.")
         print("")
         print("First we use 10 fold cross validation to decide which model is best.")
         print("")

         #Since printing out all the results of cross validation with different lambdas get tedi
         # we make a list of cv values and corresponding degrees and lambdas.
         # Then we can just sort by the cv value to pick the best model
         cv_list = []

         for n in range(1,6) :
             for a in [0.1, 0.2, 0.5, 1] :
                 pLasso = polynomialLasso(n, a)
                 cv = cross_validation(10, XY, z, pLasso)
                 cv_list.append((n,a,cv))

         cv_list.sort(key=lambda tup: tup[2]) #sort the list by the cv value

         best_tup = cv_list[0]

         print("The best model had a degree %d polynomial and a lambda of %f " % (best_tup[0], b
         print("It had a predicted error of %f" % best_tup[2])

         # NOTE: Since this is not entirely deterministic, we sometimes get different answers.
```

```
        # In fact it seems incredibly unstable.
        # I assume a big part of this is that the lasso models sets almost all betas to zero
        # When I did it, I got a degree 2 polynomial with a lambda of 0.2
```

```
Testing Lasso methods for polynomial of degrees 1,2,3,4, and 5.

First we use 10 fold cross validation to decide which model is best.

The best model had a degree 4 polynomial and a lambda of 0.100000
It had a predicted error of 0.124234
```

1.2.8 C - working with chosen model

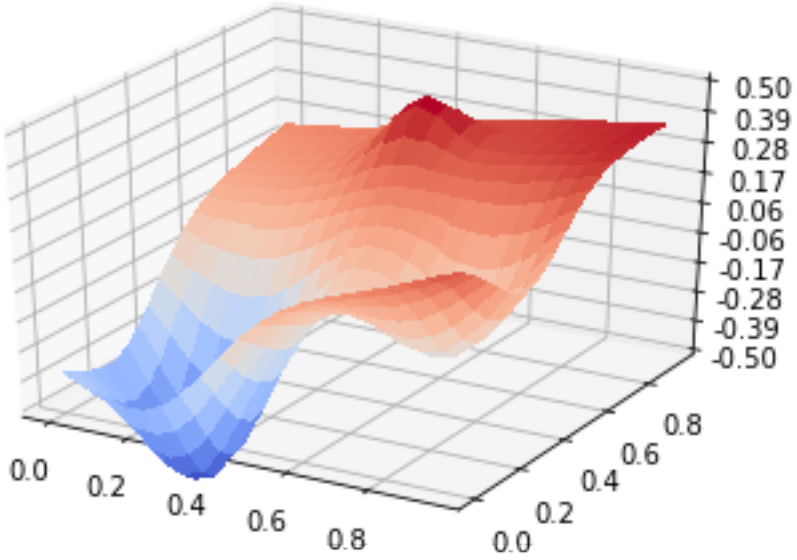
```
In [66]: # We will work with a degree 5 polynomial and a lambda of 0.1
         # We test the actual statistics against the unseen (clean) data
         bestLasso = polynomialLasso(4, 0.1)
         bestLasso.train(XY, z)
         print("The coefficients for our best Lasso model is:")
         print(bestLasso.coefficients())
         test_against_new(cleanXY, cleanz, bestLasso)

         # Now we compute the bias and variance of our model
         # and esitamte the variance of the parameters beta

         print("")
         print("Using 9 folds to predict variance and bias")
         b, v = BV_estimate(9, XY, z, bestLasso)
         print("The bias is %f" % b)
         print("The variance is %f" % v)
         print("")
         print("Using 10 folds to compute variances of betas")
         print(beta_variance(10, XY, z, bestLasso))
```

```
The coefficients for our best Lasso model is:
[[ 0.39662301]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
 [-0.        ]
```

```
 [-0.        ]
 [-0.        ]]
Lasso regression, trying to fit a polynomial of degree 4
Lambda = 0.1000
MSE = 0.130758
R2 scrore = -0.020091
----------------

Using 9 folds to predict variance and bias
The bias is 0.128723
The variance is 0.000007

Using 10 folds to compute variances of betas
[2.51423352e-05 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00]
```

1.2.9 C - the plot

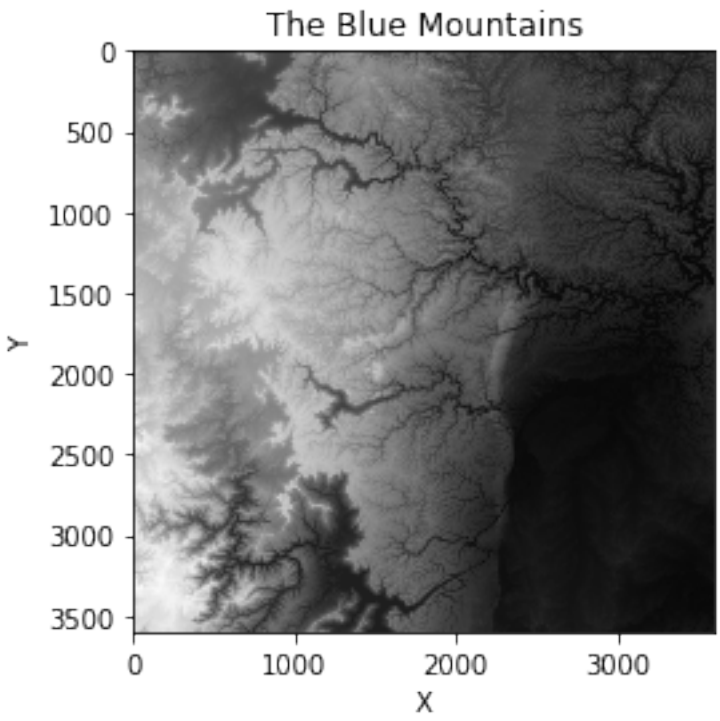
In [61]:
```
# retrain model as it has been modified by BV and variance computations
bestLasso.train(XY, z)

# Do a plot of the best model - franke
fig = plt.figure()
ax = fig.gca(projection='3d')

# Make plot points .
xpoints = np.arange(0, 1, 0.05)
ypoints = np.arange(0, 1, 0.05)
xm, ym = np.meshgrid(xpoints,ypoints)

# Plot the real and predicted surfaces.
diffsurf = ax.plot_surface(xm, ym, np.vectorize(bestLasso.plotfunc)(xm,ym) - FrankeFunc

# Customize the z axis.
ax.set_zlim(-0.50, 0.5)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

# Add a color bar which maps values to colors.
#fig.colorbar(realsurf, shrink=0.5, aspect=5)

plt.show()
```



1.3 Parts d and e

1.3.1 Loading Tarain data (part d)

In [67]:
```
################
# I got my tarrain data by searching for blue mountains on earthexplorer
# Used Blue Mountains, New South Wales, Australia -33.4100 150.3037
################

# Load the terrain
terrain1 = imread('s34_e150_1arc_v3.tif')
# Show the terrain
plt.figure()
plt.title('The Blue Mountains')
plt.imshow(terrain1, cmap='gray')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

terrain_array = np.array(terrain1)
print(terrain_array.shape)
```




```
(3601, 3601)
```

1.4 Terrain analysis (part e) - model selection

```
In [68]: def terrain_function(row) :
             return terrain_array[(row[0], row[1])]

         # Setup input data.
         # We pick a 1000 (x,y) integer points at random
         # the z's according to the fomula and with some noise
         tnumberofpoints = 10000
         tnoise = 0.5
         tXY = np.random.randint(3600, size = (tnumberofpoints, 2))

         tz = np.apply_along_axis(terrain_function, 1, tXY).reshape(-1,1)

         # Now we make some "clean" data
         # Models will never be trained on this, only tested against it
         ctXY = np.random.randint(3600, size = (1000, 2))
         ctz = np.apply_along_axis(terrain_function, 1, ctXY).reshape(-1,1)

         print("we do cross fold validation to pick the best model")
         cv_list = []
```

```
        #first we try the OLS models

        for n in range(1,6) :
            pOLS = polynomialOLS(n)
            cv = cross_validation(10, tXY, tz, pRidge)
            cv_list.append((cv,("OLS", n)))

        # then we try Ridge
        for n in range(1,6) :
            for a in [0.1, 0.2, 0.5, 1] :
                pRidge = polynomialRidge(n, a)
                cv = cross_validation(10, tXY, tz, pRidge)
                cv_list.append((cv,("Ridge", n, a)))

        # then we try Ridge
        for n in range(1,6) :
            for a in [0.1, 0.2, 0.5, 1] :
                pLasso = polynomialLasso(n, a)
                cv = cross_validation(10, tXY, tz, pLasso)
                cv_list.append((cv,("Lasso", n, a)))

        cv_list.sort(key=lambda tup: tup[0]) #sort the list by the cv value

        print(cv_list[0])

        # Again the results are not exactly fixed, but when I ran it, I got
        # Ridge with degree 3 with lambda 0.2
```

```
we do cross fold validation to pick the best model
(27589.21792591468, ('Ridge', 3, 0.5))
```

1.4.1 Working with chosen model

```
In [74]: # We will work with a degree 3 polynomial and a lambda of 0.5
         # We test the actual statistics against the unseen (clean) data
         tmodel = polynomialRidge(3, 0.5)
         tmodel.train(tXY, tz)
         print("The coefficients for our best model is:")
         print(tmodel.coefficients())
         test_against_new(ctXY, ctz, tmodel)

         # Now we compute the bias and variance of our model
         # and esitamte the variance of the parameters beta

         print("")
         print("Using 9 folds to predict variance and bias")
         b, v = BV_estimate(9, tXY, tz, tmodel)
```

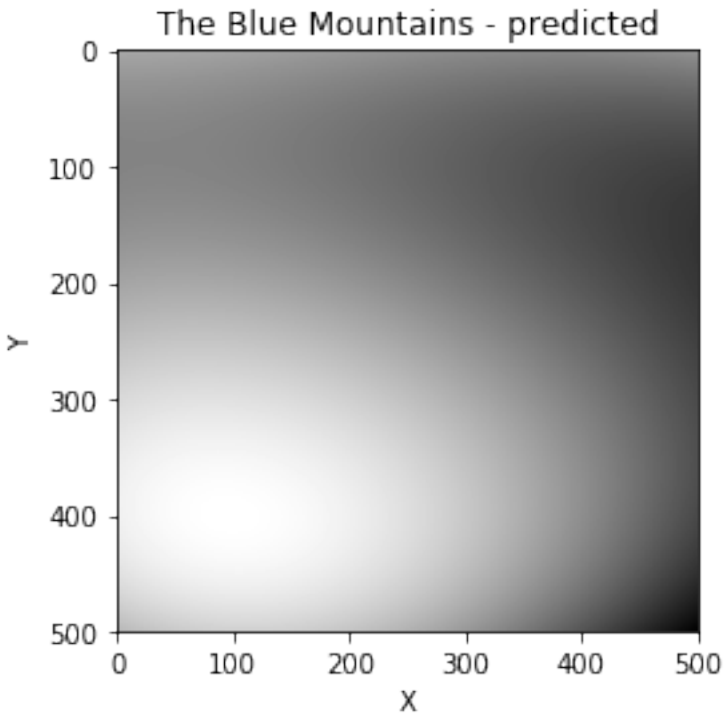
```
        print("The bias is %f" % b)
        print("The variance is %f" % v)
        print("")
        print("Using 10 folds to compute variances of betas")
        print(beta_variance(10, tXY, tz, tmodel))
```

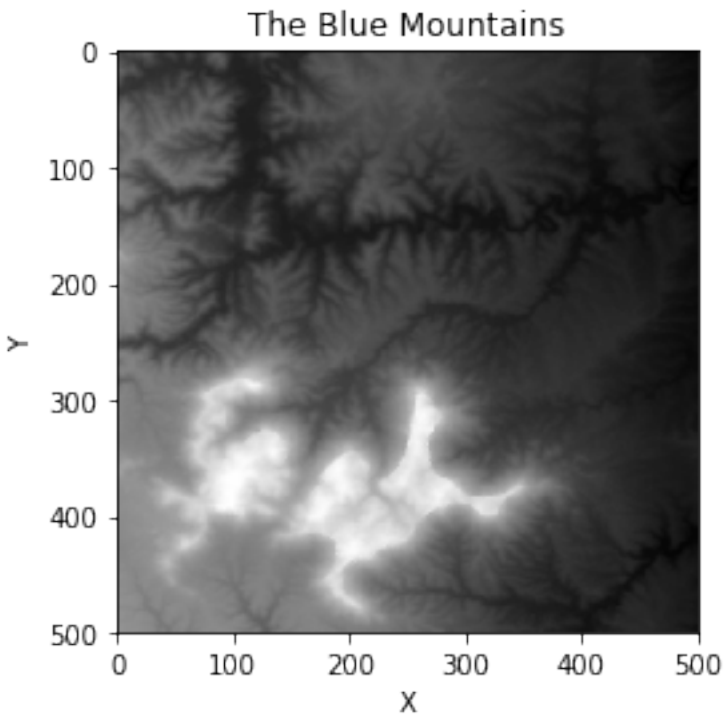
```
The coefficients for our best model is:
[[ 7.74423356e+02]
 [ 2.63623576e-01]
 [-2.41755778e-01]
 [-6.09620512e-05]
 [-4.45552817e-05]
 [ 1.35688615e-05]
 [ 6.49809862e-09]
 [ 1.08423133e-08]
 [ 4.29164295e-09]
 [ 1.75566896e-10]]
Ridge regression, trying to fit a polynomial of degree 3
Lambda = 0.5000
MSE = 27383.656382
R2 scrore = 0.767798
----------------

Using 9 folds to predict variance and bias
The bias is 24636.566560
The variance is 5.116623

Using 10 folds to compute variances of betas
[1.42955179e+01 3.32090863e-06 1.19403923e-05 2.00656719e-13
 2.76789538e-13 5.22608739e-13 3.33498950e-19 8.82435959e-20
 2.44105857e-19 8.89518016e-20]
```

1.4.2 ploting terrain

```
In [76]: # Because of limited computing power of my laptop,
         # we will restrict our selves to the 500 by 500 top corner of the map
         # We train our model on some data from there,
         # Then plot it and the real map

         restricted_XY = np.random.randint(499, size = (10000, 2))
         restricted_z = np.apply_along_axis(terrain_function, 1, restricted_XY).reshape(-1,1)

         rc_XY = np.random.randint(499, size = (1000, 2))
         rc_z = np.apply_along_axis(terrain_function, 1, rc_XY).reshape(-1,1)

         tmodel = polynomialRidge(3, 0.5)
         tmodel.train(restricted_XY, restricted_z)
```

```
# plot a patch of our predicted landscape
my_terrain = np.rint(np.vstack([np.hstack([ tmodel.plotfunc(tx,ty) for ty in range(500)
plt.figure()
plt.title('The Blue Mountains - predicted')
plt.imshow(my_terrain, cmap='gray')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()


# Show the terrain
plt.figure()
plt.title('The Blue Mountains')
plt.imshow(terrain_array[:500, :500], cmap='gray')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

print("The coefficients for our best model is:")
print(tmodel.coefficients())
test_against_new(rc_XY, rc_z, tmodel)
```






```
The coefficients for our best model is:
[[ 6.03684600e+02]
 [-2.08347169e+00]
 [-7.31907064e-03]
 [ 1.40890570e-02]
 [ 1.25217434e-03]
 [-1.45402890e-03]
 [-1.95642716e-05]
 [ 1.83078371e-06]
 [-7.47418512e-06]
 [ 2.55957023e-06]]
Ridge regression, trying to fit a polynomial of degree 3
Lambda = 0.5000
MSE = 6684.622435
R2 scrore = 0.713436
----------------
```