**Abstract**

This is the abstract

# 1 Ordinary Least Squares on the Franke Function

In this section, we will generate our target values by sampling the Franke Function on the unit square. This function, which is widely used when testing interpolation and fitting algorithms, is given by

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \\ &+ \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)^2}{10}\right) \\ &+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \\ &- \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right) \end{aligned}$$

It should be noted right away that even if the Franke function is exponential in $x$ and $y$, Taylor's theorem should guarantee that we will be able to get quite close to the original Franke data by using fifth-degree polynomials. In addition to the "pure" Franke values given by the function, we will generate a couple of perturbed variations by adding normally distributed noise. This will give us the opportunity to study how it will be more difficult to create a good fit to the data as it grows more complex.

## 1.1 Generating and visualizing data

The Franke data are generated in the notebook `20190920-Generating-Franke-data.ipynb`, which relies on `src/data/generate_data.py`. By varying the noise term, we get three sets of target values, which are stored in separate files:

- `no_noise.csv` with no noise added to the Franke data
- `some_noise.csv`, where normally distributed noise with mean 0 and standard deviation 0.1 is added to the Franke data
- `noisy.csv`, where normally distributed noise with mean 0 and standard deviation 0.9 is added to the Franke data

The data are stored in `data/generated/`. Of course, the data could easily have been generated on the fly as needed, but by storing it at this point, we facilitate reproducibility, as we do not run the risk of generating lots of datasets that may

look similar, but actually are different from each other. From this point on, we will work with the data read from the files, and nothing else.

In the same notebook (`20190920-Generating-Franke-data.ipynb`) we also generate a feature matrix $X$. In addition to the original grid points $\{(x = 0.05i, y = 0.05j) : i, j = 0, ..., 19\}$ we include features of the form $x^m y^n$, where $m$ and $n$ are non-negative integers and $m + n \leq 5$. This is in order to perform polynomial regression analysis, which is just another name for ordinary linear regression performed on a feature matrix augmented by polynomial combinations of the original features. The heavy lifting in constructing the polynomial features is performed by the class `PolynomialFeatures` in `src/features/polynomial.py`. Again, we choose to store the generated feature matrix for subsequent use (`data/generated/X.csv`).

We now have one common feature matrix and three different sets of target values. In the notebook `20190905-visualizing-franke.ipynb`, we perform some simple exploratory data analysis on those datasets. The table below shows the output of the `describe` function from `pandas`[1].

| | No noise | Some noise (sigma 0.1) | Noisy (sigma 0.9) |
|---|---|---|---|
| count | 400 | 400 | 400 |
| mean | 0.43 | 0.43 | 0.42 |
| std | 0.28 | 0.3 | 0.94 |
| min | $4.5 \cdot 10^{-2}$ | $-0.13$ | $-2.37$ |
| 25 | 0.23 | 0.21 | $-0.26$ |
| 50 | 0.35 | 0.37 | 0.45 |
| 75 | 0.57 | 0.59 | 1 |
| max | 1.22 | 1.37 | 3.09 |

The table gives us some insight into the difference between the three sets of target values, in particular when we look at the spread (compare the standard deviation and the difference between the max and min rows.)

We can also get a feel of the data by plotting them. The code for the following contour plots is in the notebook `20190905-visualizing-franke.ipynb`. We see that the original Franke function seems quite smooth, whereas the noisy version is full of "spikes" that will make fitting harder.

## 1.2 Ordinary Least Squares regression - theoretical recap

### 1.2.1 Function fitting

We now proceed to actually fitting functions to the data points. The OLS (Ordinary Least Squares) method is very well known, so the following description will be brief. Given data points $\{(x_i, y_i) : i = 1, ..., m\}$, where each $x_i$ is a vector in $\mathbb{R}^n$ and each $y_i$ is a real number, we want to obtain a parameter vector $\hat{\beta} \in \mathbb{R}$ so that for each input vector $x_i$, the predicted output $\hat{y_i} = {x_i}^T \hat{\beta}$ will be "close

[1] I have saved the output into the file `1_description_table.csv` and loaded it into LaTeX using the `pgfplotstabletypeset` package, but as my I am not able to display the percentage signs in the row names of the percentile rows.

to" the target value $y_i$. One possible way to do this is to choose $\hat{\beta}$ so as to minimize the sum of squared errors, that is, to set

$$\hat{\beta} = \arg\min_{\beta} \sum_{i=1}^{m} (y_i - \hat{y_i})^2$$

If we collect the $(x_i)$ in a feature matrix

$$\mathbf{X} = \begin{bmatrix} {x_1}^T \\ {x_2}^T \\ \dots \\ {x_m}^T \end{bmatrix}$$

we easily see that the above minimization problem can be formulated as

$$\hat{\beta} = \arg\min_{\beta} \|\mathbf{y} - \mathbf{X}\beta\|^2$$

The minimization problem turns out to have a closed form solution, which can be found using matrix differentiation. The solution is

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

Of course, this solution only exists if $\mathbf{X}^T\mathbf{X}$ is nonsingular, but this is rarely a problem in real-life situations. There are more effective ways of computing the least-squares estimate for $\beta$, but we stick to matrix inversion here, as it is very easy to implement. The implementation is in the `fit` method of the `OLS` class in `src/models/models.py`.

### 1.2.2 Estimates of error and variability

Once we have obtained our estimates for $\beta$, we can get predictions $\hat{\mathbf{y}}$ by putting $\hat{\mathbf{y}} = \mathbf{X}\hat{\beta}$. We then can assess the quality of our estimates in various ways. The MSE (Mean Square Error) is one such measure. Given target values $\mathbf{y}$ and predictions $\hat{\mathbf{y}}$, the MSE is given by $\frac{1}{m}\|\mathbf{y} - \hat{\mathbf{y}}\|^2$. The implementation of the MSE is in `src/evaluation/evaluation.py`, along with a function for evalutating the $R^2$ score. The $R^2$ score measures how much the error is reduced by using our linear model as opposed to a pure mean model, that is, a model where we predict $\hat{y_i} = \frac{1}{m}\sum_{i=1}^{m} y_i$ for $i = 1, ..., m$. The ratio between the mean square errors of the two models is

$$\frac{\frac{1}{m}\sum_{i=1}^{m} (y_i - \hat{y_i})^2}{\frac{1}{m}\sum_{i=1}^{m} (y_i - \bar{\mathbf{y}})^2}$$

where $\bar{\mathbf{y}} = \frac{1}{m}\sum_{i=1}^{m} y_i$. We want this ratio to be low, that is, we want our model predictions to be much better than the values predicted by the pure mean model. This leads us to defining an measure

$$R^2 = 1 - \frac{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y_i})^2}{\frac{1}{m}\sum_{i=1}^{m}(y_i - \bar{\mathbf{y}})^2}$$

which will hopefully be close to 1.

The last measure we are going to discuss is the variance of our estimates for $\beta$. This is an important issue, because when doing linear regression, we assume a "ground truth" model $y_i = {x_i}^T\beta + \epsilon$, where the $\epsilon$ is a normally distributed random variable. This means that $\hat{\beta}$ is a random variable too, and we are interested in its variance, which will allow us to construct condifence intervals around our point estimates. It can be shown that the covariance matrix of $\hat{\beta}$ is $\sigma^2(\mathbf{X}^T\mathbf{X})^{-1}$, where $\sigma^2$ is the variance of $\epsilon$. Because we are mainly interested in the variance of the individual components of $\hat{\beta}$, we focus on the diagonal of this matrix, and find that $\mathrm{Var}(\hat{\beta_j}) = \sigma^2((\mathbf{X}^T\mathbf{X})^{-1})_{j,j}$. In general, $\sigma^2$ will be unknown to us, but it can be estimated by the MSE we calculated previously. This leads to the expression

$$\hat{\sigma}^2(\hat{\beta_j}) = \mathrm{MSE} \times ((\mathbf{X}^T\mathbf{X})^{-1})_{j,j}$$

as our estimate of the variance of ${\hat{\beta_j}}^2$ Having obtained variance estimates, we can take their square root $\hat{\sigma}_{\hat{\beta_j}}$ and construct 95 % confidence intervals as

$$\left[\hat{\beta_j} - 1.96\hat{\sigma}_{\hat{\beta_j}}, \hat{\beta_j} + 1.96\hat{\sigma}_{\hat{\beta_j}}\right]$$

### 1.3 Ordinary Least Squares regression - results

In the notebook `20190906-OLS-Franke.ipynb`, we perform OLS regression on our generated datasets and evaluate our results using the metrics described above. This notebook depends upon `src/models/models.py` for the actual data fitting and `src/evaluation/evaluation.py` for the metrics. We begin by comparing the output from the regression method with the target values of the three datasets, and obtain the following results:

| | MSE | $R^2$ |
|---|---|---|
| No noise | $2.1434 \cdot 10^{-3}$ | 0.99993 |
| Some noise (sigma 0.1) | 0.01064 | 0.9997 |
| Noisy (sigma 0.9) | 0.77315 | 0.99783 |

As we see, the $R^2$ scores are very close to one, meaning that our model explains most of the variability in the data. The MSE, as expected, grows significally with added noise. A fifth-degree polynomial can be fitted very well to a pure Franke function, whereas with added noise, the fitting becomes much harder.

[2]Have I misunderstood anything here? According to the lecture note, the square root of $((\mathbf{X}^T\mathbf{X})^{-1})_{j,j}$ should be taken in this formula, but this does not seem right.

We now proceed to the actual parameter estimates and their variances. Because we have three datasets and there are 21 parameters, we get quite a lot of data here. In the notebook `20190906-OLS-Franke.ipynb` we construct three csv files corresponding to the three datasets. For the no-noise case, we get the following data:

| Feature | Estimate | Variance | Lower bound | Upper bound |
|---|---|---|---|---|
| 1 | 0.54154 | $7.55532 \cdot 10^{-4}$ | 0.48766 | 0.59541 |
| x | 6.07027 | 0.12238 | 5.3846 | 6.75593 |
| y | 3.16585 | 0.12238 | 2.48018 | 3.85151 |
| $(x^2)$ | −27.6196 | 3.4886 | −31.28044 | −23.95875 |
| (x)(y) | −11.6201 | 2.01247 | −14.40059 | −8.83961 |
| $(y^2)$ | −6.30394 | 3.4886 | −9.96479 | −2.6431 |
| $(x^3)$ | 32.72699 | 20.48258 | 23.85648 | 41.5975 |
| $(x^2)(y)$ | 41.72654 | 10.53252 | 35.36559 | 48.08749 |
| $(x)(y^2)$ | 18.58331 | 10.53252 | 12.22236 | 24.94427 |
| $(y^3)$ | −15.79604 | 20.48258 | −24.66655 | −6.92553 |
| $(x^4)$ | −3.94401 | 25.24997 | −13.79288 | 5.90486 |
| $(x^3)(y)$ | −60.43109 | 13.81827 | −67.71699 | −53.1452 |
| $(x^2)(y^2)$ | 0.22793 | 11.68392 | −6.47169 | 6.92756 |
| $(x)(y^3)$ | −33.43817 | 13.81827 | −40.72407 | −26.15228 |
| $(y^4)$ | 41.04946 | 25.24997 | 31.20058 | 50.89833 |
| $(x^5)$ | −7.93131 | 4.30853 | −11.99968 | −3.86293 |
| $(x^4)(y)$ | 25.99715 | 3.06771 | 22.56423 | 29.43006 |
| $(x^3)(y^2)$ | 5.77911 | 2.83309 | 2.48008 | 9.07814 |
| $(x^2)(y^3)$ | −5.34619 | 2.83309 | −8.64522 | −2.04716 |
| $(x)(y^4)$ | 19.13743 | 3.06771 | 15.70451 | 22.57034 |
| $(y^5)$ | −22.59142 | 4.30853 | −26.6598 | −18.52305 |

This table shows the $\beta$ estimates for each individual feature, along with the estimates for the variance and the lower and upper bound for the corresponding confidence intervals.

For the case where normally distributed noise with $\sigma = 0.1$ is added to the output from the Franke function, we get the following parameter estimates and confidence intervals:

Finally, for the case where normally distributed noise with $\sigma = 0.9$ is added, we get this table:

# 2 Resampling techniques, adding more complexity

Hitherto, we have evaluated our results using the same data we used for model fitting. In most real-life scenarios, this is considered bad practice, as the models tend to "learn" noise in addition to the underlying distribution of the data. Thus, we easily end up with models that are able to reproduce the original data

| Feature | Estimate | Variance | Lower bound | Upper bound |
|---|---|---|---|---|
| 1 | 0.57132 | $3.74901 \cdot 10^{-3}$ | 0.45131 | 0.69133 |
| x | 6.28238 | 0.60726 | 4.75501 | 7.80975 |
| y | 2.8003 | 0.60726 | 1.27293 | 4.32768 |
| $(x^2)$ | −28.66944 | 17.31069 | −36.82423 | −20.51464 |
| (x)(y) | −13.88187 | 9.98605 | −20.07561 | −7.68814 |
| $(y^2)$ | −3.36449 | 17.31069 | −11.51929 | 4.79031 |
| $(x^3)$ | 33.98078 | 101.6362 | 14.22109 | 53.74048 |
| $(x^2)(y)$ | 49.92817 | 52.26317 | 35.75869 | 64.09765 |
| $(x)(y^2)$ | 21.24688 | 52.26317 | 7.0774 | 35.41636 |
| $(y^3)$ | −24.48217 | 101.6362 | −44.24187 | −4.72247 |
| $(x^4)$ | −4.49262 | 125.29237 | −26.4317 | 17.44645 |
| $(x^3)(y)$ | −69.94827 | 68.56736 | −86.17813 | −53.71841 |
| $(x^2)(y^2)$ | −4.23323 | 57.97655 | −19.15712 | 10.69067 |
| $(x)(y^3)$ | −35.41406 | 68.56736 | −51.64392 | −19.1842 |
| $(y^4)$ | 51.63963 | 125.29237 | 29.70055 | 73.57871 |
| $(x^5)$ | −7.76393 | 21.37928 | −16.82652 | 1.29867 |
| $(x^4)(y)$ | 29.27725 | 15.22221 | 21.63018 | 36.92432 |
| $(x^3)(y^2)$ | 8.55686 | 14.05803 | 1.20803 | 15.90569 |
| $(x^2)(y^3)$ | −4.82208 | 14.05803 | −12.17091 | 2.52675 |
| $(x)(y^4)$ | 19.95001 | 15.22221 | 12.30295 | 27.59708 |
| $(y^5)$ | −27.15941 | 21.37928 | −36.22201 | −18.09682 |

quite well, but do not necessarily deal equally well with "unknown" data. This is known as the overfitting problem.

In order to get a better picture of how well our models generalize to unseen data, we use different resampling techniques. We can for instance split our data into two parts, known as the test set and the training set. We use the training set for model fitting and the test set for evaluation. As the test set consists entirely of unseen data, this gives us a much better understanding of the generalizing ability of the model.

In the notebook `20190918-Resampling-OLS-Franke.ipynb` we use the function `train-test-split` from `sklearn.model_selection` in order to achieve such a split. When evaluating our model's ability to reproduce the test data after having been trained on the training data, we get the following MSE and $R^2$:

If we compare this table to the corresponding table we made earlier, we see that the MSE is higher, in particular in the noisy case (1.068 versus 0.773). This is of course to be expected, as we now test our model on unseen data. The $R^2$ scores still are fairly high, although a little lower than before.

Resampling can be done in other ways as well. One possible approach is k-fold cross-validation, where we start by partitioning our data $\mathcal{D}$ into $k$ disjoint folds $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_k$. For $j = 1, ..., k$, we then use $\mathcal{D} \setminus \mathcal{D}_j$ as our training set and $\mathcal{D}_j$ as our test set. For each value of $j$, we can obtain a MSE score, and by averaging the MSE scores, we hopefully get a precise estimate of the true MSE.

| Feature | Estimate | Variance | Lower bound | Upper bound |
|---|---|---|---|---|
| 1 | 0.80961 | 0.27253 | −0.2136 | 1.83282 |
| x | 7.9793 | 44.14436 | −5.04318 | 21.00178 |
| y | −0.12404 | 44.14436 | −13.14652 | 12.89844 |
| $(x^2)$ | −37.06813 | 1,258.38098 | −106.59652 | 32.46025 |
| (x)(y) | −31.97609 | 725.92419 | −84.78433 | 20.83215 |
| $(y^2)$ | 20.1511 | 1,258.38098 | −49.37729 | 89.67948 |
| $(x^3)$ | 44.01111 | 7,388.32721 | −124.46143 | 212.48365 |
| $(x^2)(y)$ | 115.54122 | 3,799.21169 | −5.26876 | 236.3512 |
| $(x)(y^2)$ | 42.55543 | 3,799.21169 | −78.25455 | 163.36541 |
| $(y^3)$ | −93.97123 | 7,388.32722 | −262.44377 | 74.50132 |
| $(x^4)$ | −8.88154 | 9,107.98586 | −195.93565 | 178.17256 |
| $(x^3)(y)$ | −146.08565 | 4,984.42574 | −284.46256 | −7.70874 |
| $(x^2)(y^2)$ | −39.92251 | 4,214.53908 | −167.16469 | 87.31968 |
| $(x)(y^3)$ | −51.22112 | 4,984.42574 | −189.59803 | 87.15579 |
| $(y^4)$ | 136.36105 | 9,107.98588 | −50.69306 | 323.41516 |
| $(x^5)$ | −6.42488 | 1,554.14253 | −83.6932 | 70.84344 |
| $(x^4)(y)$ | 55.51806 | 1,106.56081 | −9.68136 | 120.71748 |
| $(x^3)(y^2)$ | 30.77886 | 1,021.93206 | −31.87778 | 93.4355 |
| $(x^2)(y^3)$ | −0.62923 | 1,021.93206 | −63.28587 | 62.02741 |
| $(x)(y^4)$ | 26.45069 | 1,106.56081 | −38.74873 | 91.65011 |
| $(y^5)$ | −63.70335 | 1,554.14253 | −140.97167 | 13.56498 |

| | MSE | $R^2$ |
|---|---|---|
| No noise | $3.42806 \cdot 10^{-3}$ | 0.99963 |
| Some noise (sigma 0.1) | 0.01482 | 0.99852 |
| Noisy (sigma 0.9) | 1.0677 | 0.99055 |

In the same notebook (`20190918-Resampling-OLS-Franke.ipynb`) we use the class `K_fold_splitter` from `src/resampling/resampling.py` to estimate the test MSE using cross-validation. We obtain the following results on our three datasets:

| | MSE |
|---|---|
| No noise | $2.64309 \cdot 10^{-3}$ |
| Some noise (sigma 0.1) | 0.01235 |
| Noisy (sigma 0.9) | 0.87061 |

# 3 Bias-variance tradeoff

Above we mentioned the importance of validating our models on unseen data in order to avoid overfitting. In this section, we will bring this somewhat "intuitive" idea onto a more precise theoretical footing. In particular, we will decompose

our test MSE into three separate terms, of which the first may be interpreted as error stemming from adopting too rigid a model, which may not be able to capture the relevant fluctuations in the training set properly. This is called the bias (or to be more precise, the square of the bias). An obvious way to reduce the bias is to adopt a more flexible model, but this may easily introduce a second kind of error, which is the one described above, i. e. that the model fits too closely to the training set, picking up fluctuations due to sampling rather than to the true underlying distribution of the data. This second kind of error is known as variance.

As stated above, we take as our point of departure a dataset $\{(x_i, y_i) : i = 1, ..., m\}$ and a "ground truth" model $\mathbf{y} = f(x) + \epsilon$. Hitherto, the function $f$ has been of the linear form $f(x) = x^T\beta$, but in the following discussion we will consider general models of this type. What is important is that the noise term $\epsilon$ is normally distributed with mean 0 and variance $\sigma^2$.

When doing regression analysis, our goal is to obtain a model $\hat{f}$ to obtain predictions $\hat{\mathbf{y}} = \hat{f}(x)$. In this setting, both $\mathbf{y}$, $\hat{\mathbf{y}}$ and $\epsilon$ are random variables. We have already stated that $\mathbb{E}(\epsilon) = 0$ and $\mathrm{Var}(\epsilon) = \sigma^2$, and now proceed to compute the expectations and variances of $\mathbf{y}$ and $\hat{\mathbf{y}}$. To unclutter notation, we put $f = f(x)$, and note that since $f$ is deterministic, its expected value is $f$ and its variance is zero. We have

$$\begin{aligned}\mathbb{E}(\mathbf{y}) &= \mathbb{E}(f + \epsilon) \\ &= \mathbb{E}(f) + \mathbb{E}(\epsilon) \\ &= f\end{aligned}$$

and

$$\begin{aligned}\mathrm{Var}(\mathbf{y}) &= \mathbb{E}[(\mathbf{y} - \mathbb{E}(\mathbf{y}))^2] \\ &= \mathbb{E}[(f + \epsilon - f)^2] \\ &= \mathbb{E}[(\epsilon - 0)^2] \\ &= \mathrm{Var}(\epsilon) \\ &= \sigma^2\end{aligned}$$

When doing regression analysis, we try to minimize the test MSE, which can be written as $\mathbb{E}[(\mathbf{y} - \hat{\mathbf{y}})^2]$. We can rewrite this in the following way. We set $u = \mathbb{E}(\hat{\mathbf{y}})$, again in an attempt to unclutter notation somewhat.

$$
\begin{aligned}
\mathbb{E}[(\mathbf{y}-\hat{\mathbf{y}})^2] &= \mathbb{E}[(f+\epsilon-\hat{\mathbf{y}})^2] \\
&= \mathbb{E}[(f+\epsilon-\hat{\mathbf{y}}+u-u)^2] \\
&= \mathbb{E}[((f-u)+\epsilon+(u-\hat{\mathbf{y}}))^2] \\
&= \mathbb{E}[(f-u)^2+\epsilon^2+(u-\hat{\mathbf{y}})^2+2\epsilon(f-u)+2\epsilon(u-\hat{\mathbf{y}})+2(f-u)(u-\hat{\mathbf{y}})] \\
&= \mathbb{E}[(f-u)^2]+\mathbb{E}[\epsilon^2]+\mathbb{E}[(u-\hat{\mathbf{y}})^2]+\mathbb{E}[2\epsilon(f-u)]+\mathbb{E}[2\epsilon(u-\hat{\mathbf{y}})]+\mathbb{E}[2(f-u)(u-\hat{\mathbf{y}})] \\
&= (f-u)^2+\sigma^2+\mathrm{Var}(\hat{\mathbf{y}})+0+0+0 \\
&= (f-\mathbb{E}(\hat{\mathbf{y}}))^2+\mathrm{Var}(\hat{\mathbf{y}})+\sigma^2 \\
&= \frac{1}{m}\sum_1^m (f_i-\mathbb{E}(\hat{\mathbf{y}}))^2+\frac{1}{m}\sum_1^m (\hat{y}_i-\mathbb{E}(\hat{\mathbf{y}}))^2+\sigma^2
\end{aligned}
$$

The first of these terms is the bias, which can be explained as the error resulting from erroneous assumptions in the learning algorithm, for instance that we assume a linear model when the actual underlying distribution is non-linear. The second term is the variance, which is the error resulting from our model picking up and incorporating randomness in the training set due to sampling. The last term is the irreducible error, stemming from the fact that the true underlying model $\mathbf{y} = f(x) + \epsilon$ has a random noise term.

# 4 Ridge regression on the Franke function with resampling

As described above, we have fitted our OLS model parameters by minimizing the sum of squared errors on the training set. That is, we have solved the minimization problem

$$\hat{\beta} = \arg\min_{\beta} \|\mathbf{y}-\mathbf{X}\beta\|^2$$

The idea behind ridge regression is to introduce a penalty term that will "dampen" the obtained parameters somewhat by bringing them closer to zero. The minimization problem now becomes

$$\hat{\beta} = \arg\min_{\beta} \|\mathbf{y}-\mathbf{X}\beta\|^2 + \lambda\|\beta\|^2$$

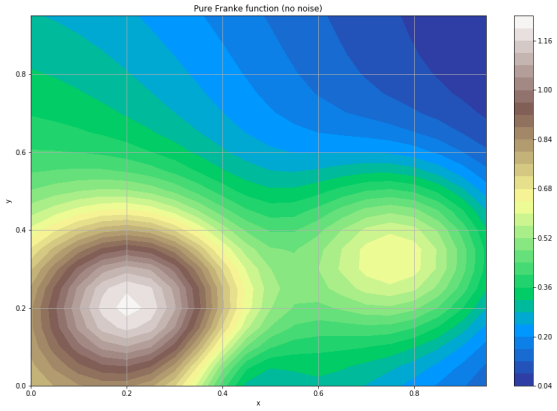
Here, $\lambda$ is a non-negative real number. If $\lambda = 0$, we get our usual OLS cost function. Higher values of $\lambda$ corresponds to more significant "damping" or regularization. As we have already stated that our original OLS approach has a closed-form solution that minimizes the sum of squared errors on the training set, we cannot expect our new approach to obtain a better result on the training set. At first glance, this may seem as a problem, but as explained above, what

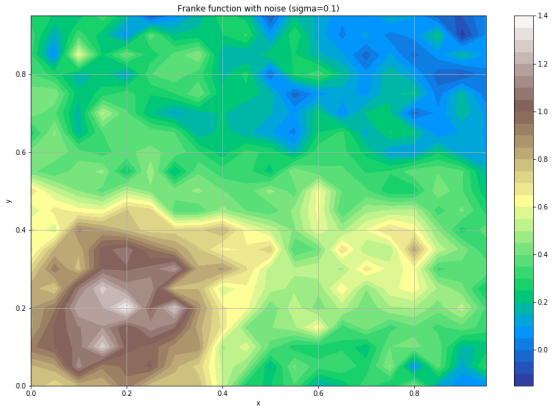
we really are interested in are models that perform well on unseen data, not on the same data as it was trained on. Our hope is that as we increase the bias slightly by forcing the $\beta$ parameters closer to zero, we reduce the variance so that the model will be less prone to overfitting. Hopefully, this will lead to a lower test MSE.

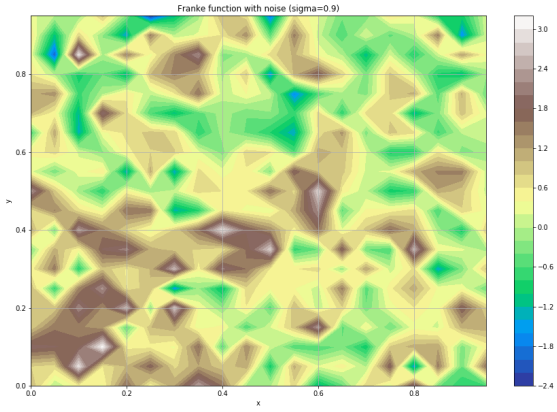
# 5 Lasso regression on the Franke function with resampling

# 6 Introducing real data

# 7 OLS, Ridge and Lasso with resampling


Pure Franke function (no noise)
x
y
0.0
0.2
0.4
0.6
0.8
1.16
1.00
0.84
0.68
0.52
0.36
0.20
0.04


Franke function with noise (sigma=0.1)
x
y
0.0
0.2
0.4
0.6
0.8
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0


Franke function with noise (sigma=0.9)
x
y
0.0
0.2
0.4
0.6
0.8
3.0
2.4
1.8
1.2
0.6
0.0
−0.6
−1.2
−1.8
−2.4