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

[1] I have saved the output into the file `description.csv` and load it into LaTeXusing the `pgfplotstabletypeset` package, but as my LaTeXskillset is somewhat limited, it shows up strangely - I am not able to display the percentage signs in the row names of the percentile rows.

in R^n and each y_i is a real number, we want to obtain a parameter vector $\hat{\beta} \in R$ so that for each input vector x_i, the predicted output $\hat{y_i} = {x_i}^T \hat{\beta}$ will be "close 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

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

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 β, 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 β, 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 β. 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 ϵ 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 σ^2 is the variance of ϵ. 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, σ^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 $\hat{\beta_j}$.

(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.)

2 Resampling techniques, adding more complexity

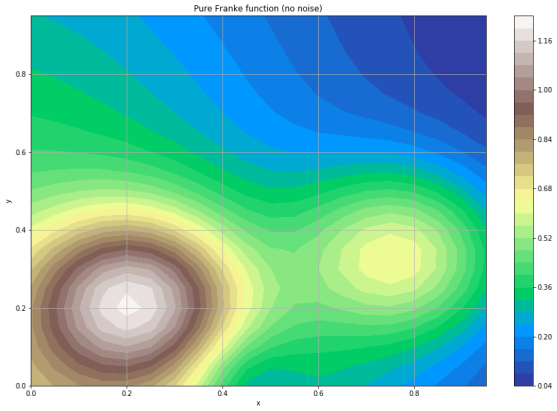
3 Bias-variance tradeoff

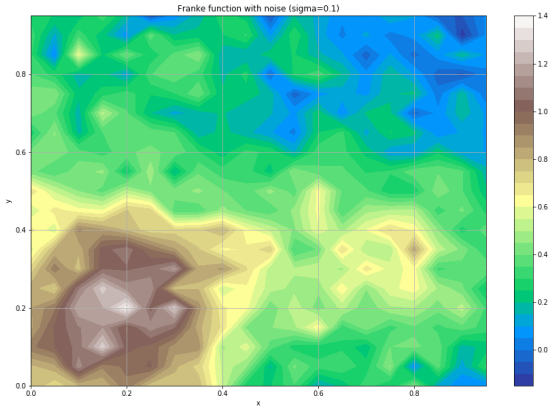
4 Ridge regression on the Franke function with resampling

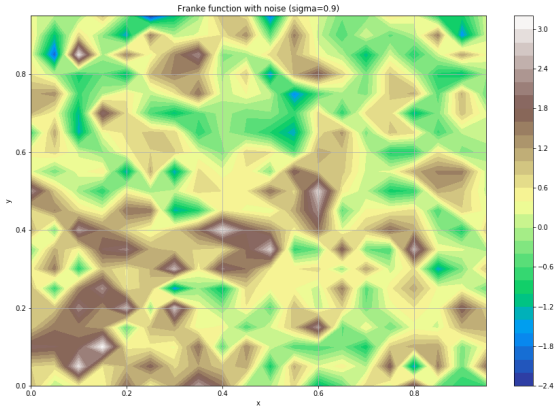
5 Lasso regression on the Franke function with resampling

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