

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 $\{(x = 0.05i, y = 0.05j) : i, j = 0, \dots, 19\}$ 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 $\{(x_i, y_i) : i = 1, \dots, m\}$, where each x_i is a vector 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

¹I have saved the output into the file `1_description_table.csv` and loaded it into \LaTeX using the `pgfplots` table typeset package, but as my \LaTeX is 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 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 evaluating 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 $i = 1, \dots, m$. 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

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

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 confidence 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 $\text{Var}(\hat{\beta}_j) = \sigma^2((\mathbf{X}^T \mathbf{X})^{-1})_{j,j}$. 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

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

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 significantly 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.

²Have 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 β 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 :

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

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

