4 Polynomial Regression

4.1 Polynomial Models

So far, we have focussed on data where there is a linear relation between input and output. In fact, this was one of the basic assumtions for linear regression from Section 2.5. But how can we deal with data where the relation is not linear, for instance as shown in Figure 4.1? In this case, we can try to fit a **polynomial model** to the data, i.e. instead of a linear hypothesis, we use a polynomial hypothesis. For instance, for the data in the figure we might use a polynomial of degree 3, i.e. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$.

▸ Code

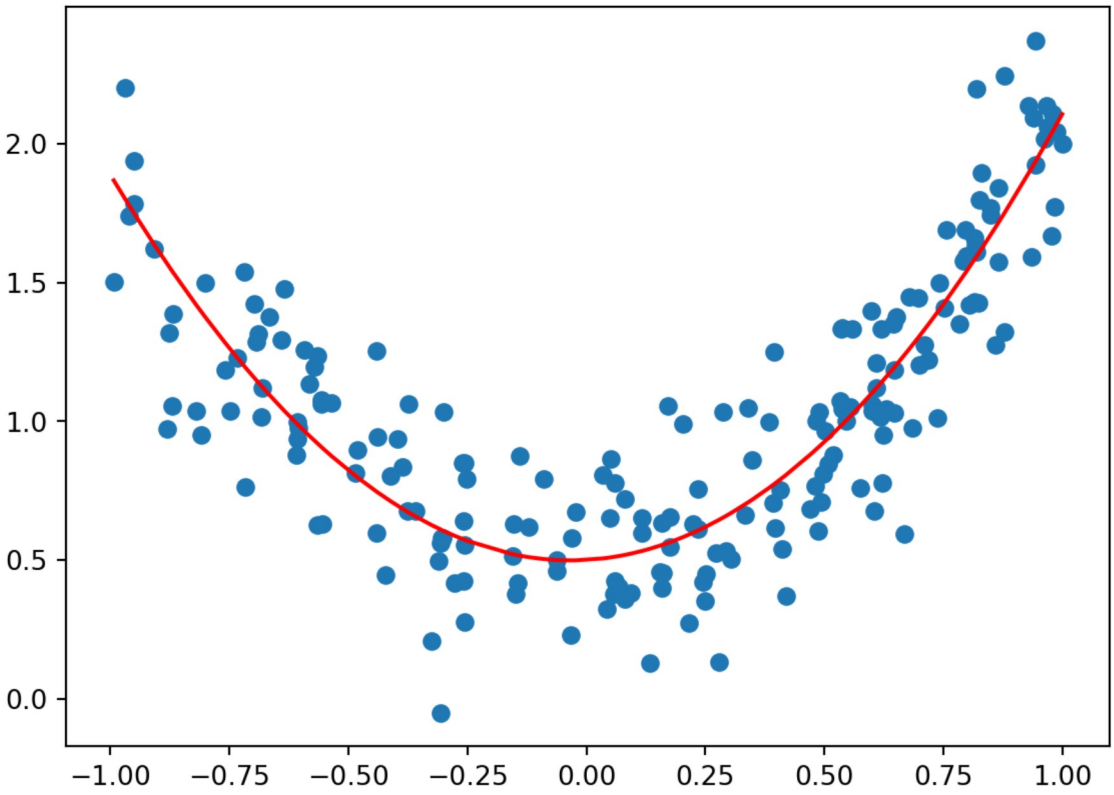

Figure 4.1: Datapoints with non-linear relation between input and output.

To formalize this, we first define *artificial variables* $z_1 = x$, $z_2 = x^2$, and $z_3 = x^3$. Using these, we can reformulate our hypothesis as $h_\theta(z) = \theta_0 z_0 + \theta_1 z_1 + \theta_2 z_2 + \theta_3 z_3$. Note that we are using now variable z as input. This is a multivariate linear regression model in z, with the restriction that the values of z_i are polynomials in x (the function is still linear in θ_0, θ_1 etc.). Hence, we can apply our methods from before - e.g. normal equation, gradient descent etc. - to solve these problems.

We can also use the same "trick" if we have multiple input variables $x_1, .. x_n$ (i.e., multivariate regression), and we can combine input variables in more complex polynomials. This might result in hypotheses such as $h(x_1, .. x_n) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 x_3 + \theta_3 x_2 x_3^3 + \theta_4 \sqrt{x_2 x_3} + \ldots$. Again, we can introduce corresponding artificial variables $z_1, z_2, \ldots$ to replace the polynomial terms and apply gradient descent etc.

4.2 Over- and Underfitting

One crucial question when applying polynomial regression is which polynomials to use. If we take very large degrees, our curve will "jump around" a lot. In extremum, we might be able to hit every point of the input almost perfectly, as shown in the blue curve in Figure 4.2 on the right side. While this model would have very good costs (close to zero), it is not really useful, since it "does not generalize" well: For a new datapoint whose value is between the first and second sample, the model's prediction is far too small, whereas a datapoint whose value is between fifth and sixth sample might be predicted too large, or too small, by the model. We call this **overfitting**, since the model fits (almost) perfectly to the given training data, but does not work well for new (unseen) data.

The opposite is **underfitting**, whereby the model does not have sufficient power or *complexity* to fit the data accurately. This is shown in Figure Figure 4.2 on the left side, where we try to fit a linear line with data that has a more complex relation.

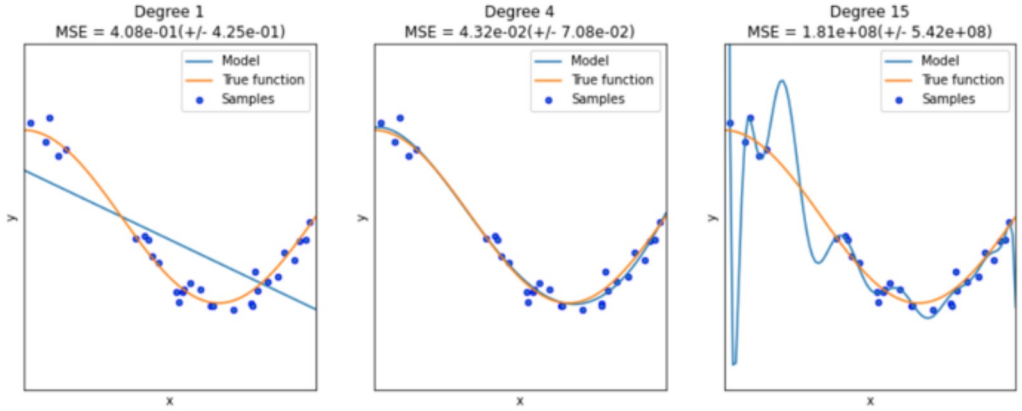

Figure 4.2: Datapoints with non-linear relation between input and output. Each orange line shows the optimal fit of a polynomial regression model of varying degree. Left: Underfitting, since degree of the polynomial (a linear model) is too low. Right: Overfitting, since the degree of polynomial is too high.

Note that, if one is working with a dataset that grows over time (e.g., more data are collected every day), then it is sometimes useful to increase the complexity of the model every now and then, e.g., by increasing the degree of the polynomial.

4.3 Regularization

In order to avoid overfitting, we can either decrease the degree of the polynomials - or we can, for a given degree, prevent the fitted curve from "jumping around" too much. Generally speaking, the smaller the absolute values of parameters θ, the less extreme the curve will be. However, we usually do not know how much "jumping around" is required to properly fit the data. Hence, we want to learn this from the data during training. This is called **regularization**, which aims to ensure that the *values* of the parameters θ do not become too large. For this, we introduce a new *regularization term* to the cost function J:

$$J(\theta) = \frac{1}{2M}\left[\sum_{m=1}^{M}(y^{(m)} - h_\theta(x^{(m)}))^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{4.1}$$

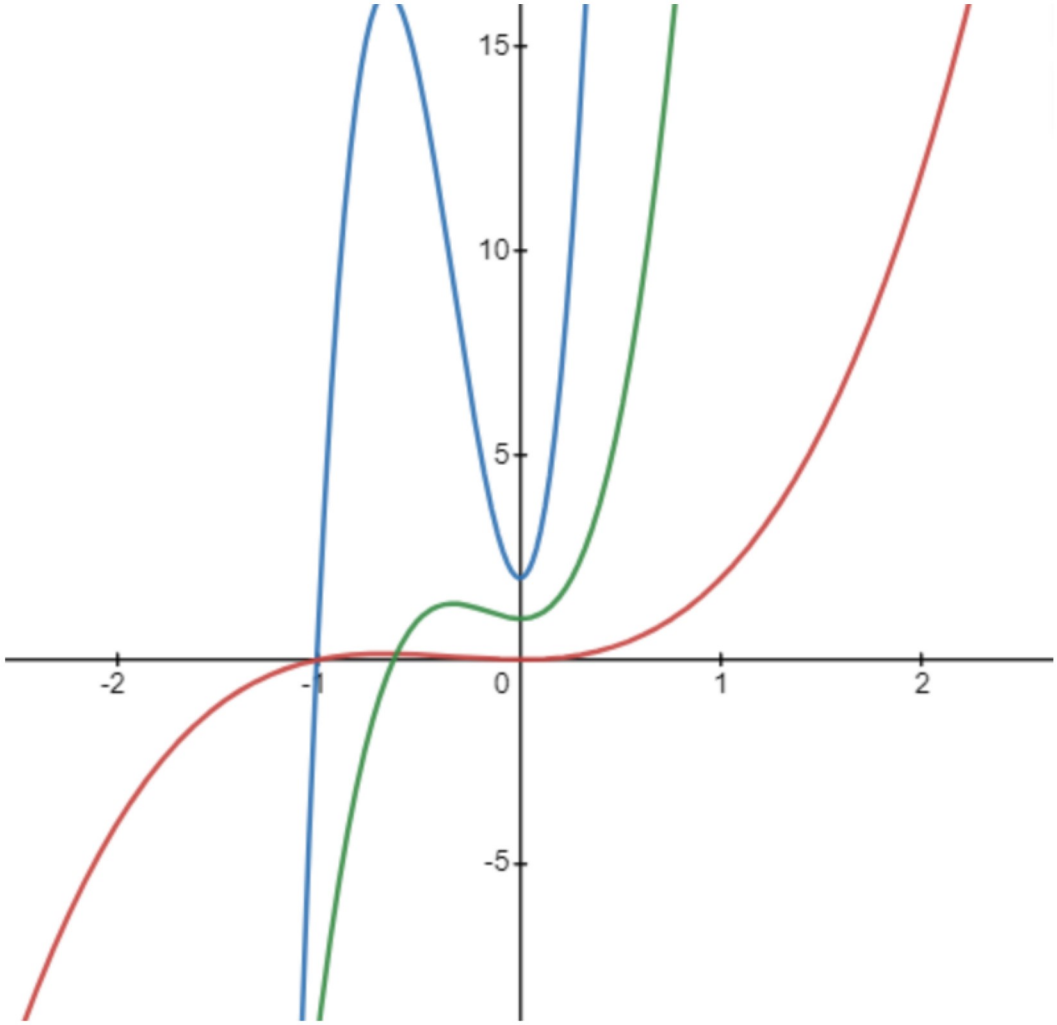

Figure 4.3: Influence of parameters on shape of polynomial functions of degree 3. Red: $y = x^2 + x^3$; Green: $y = 10x^2 + 20x^3 + 1$; Blue: $y = 99x^2 + 100x^3 + 2$

Note that the regularization term depends *only* on the parameters θ, not on the training samples. The hyperparameter λ determines "how much" regularization counts: The larger λ, the more the values of θ contribute to the cost. Since we want to minimize cost, large λ will encourage the values of θ to stay rather small, and the prediction curve will be rather smooth. In contrast, λ close to zero means that values in θ might become very large, and hence the courve might jump around a lot.

4.4 Hyperparameter Tuning

Which is the "best" value for regularization parameter λ? We cannot know beforehand, but we have to choose one to fit our regression model. One typical way to find a suitable value is **hyperparameter tuning**, where we try out different values for λ. For instance, we could use values $\lambda \in \{10, 1, 0.1, 0.01\}$, and train a model for each of these values. We then evaluate the quality of these four models (e.g. using R^2 from Section 2.6.1) to determine which λ works best.

For polynomial regression, we have two other hyperparameters: the degree of the polynomial, and the learning rate (assuming for a moment that we use a fixed learning rate, no adaptive method). Note that for different degrees of the polynomial, different values of λ and different learning rates α might work best. Hence, in principle we want to search for a good combination of all parameters - degree of the polynomial, regularization parameter λ and learning rate α - at the same time.

A straightforward approach for this would be to explore all combinations of all three hyperparameters

A straightforward approach for this would be to explore all combinations of all three hyperparameters, where we specify for each hyperparameter a range of values (e.g. degree of polynomial $\in \{3, 5, 8\}$, $\lambda \in \{10, 1, 0.1, 0.01\}$ and $\alpha \in \{1, 0.3, 0.1, 0.03, 0.001\}$) and train a model for each combination (in this case, $3 * 4 * 5 = 60$ models). This is called **grid search**, and works well as long as we do not have too many combinations of hyperparamers.

However, if we have more hyperparameters, then a complete grid search might not be possible, either because there are too many potential combinations of hyperparameters, or because training the model for each combination takes too long. In this case, search heuristics such as *local search* or *simulated annealing* are used to find good settings for the hyperparameters.