Chapter 8
Load Forecasting Model Training and Selection

Chapters 5 and 6 have shown how to define a time series forecast, how to prepare the data, and how to generate inputs for the models. Chapters 9 to 11 will show several methods for forecasting the demand. However, although Chap. 7 provided us the tools for measuring the accuracy of a forecast, the following questions remain largely unanswered: **How do we train and select a model which will consistently produce accurate forecasts?**

This chapter will investigate this question by looking at some of the most important aspects for creating a good forecast including proper utilisation of benchmarking, and how to use cross-validation to properly train your model. Underlying cross-validation is one of the most important aspects of a creating a good forecast, the so-called **bias-variance trade-off** principle, discussed in Sect. 8.1.2. This ensures that the model is not over (or under-) trained and allows the model to better generalise to new, unseen data. Next, in Sect. 8.2, methods for training the models are considered, including ways to select the best model from a selection of models. One important set of techniques covered in Sects. 8.2.4 and 8.2.5 is regularisation, which helps to reduce overfitting, but also how to find the appropriate hyperparameters within a family of models.

8.1 General Principles for Forecasts Trials

In the previous sections the general form of a forecasting problem was introduced as well as methods for scoring the forecast accuracy through error measures. This section introduces some general principles with the aim to aid the practitioner to properly design and develop a forecast trial. This includes considerations on why choosing appropriate benchmarks is important to better understand the accuracy of your model; why it is important to avoid over/under-fitting your model to the data; and how to split the data in order to properly train and test your models.

https://doi.org/10.1007/978-3-031-27852-5_8

8.1.1 Benchmarking

An error score (see Chap. 7) for a model is not very informative on its own. The accuracy of a forecast can only be understood in the context of other, well-designed forecasts. **Benchmark** models are a vital component for creating useful and accurate forecasts. They enable informative comparisons and help to better understand important (and unimportant) features and relationships in the data. Often simple benchmarks can be quite effective as their strong performance can suggest important features or drivers for the forecast accuracy. How much your model(s) improve compared to the benchmarks can also be used as performance indicators (See skill scores in Sect. 7.4).

Most benchmarks fit into the following categories:

1. **Simple or Naïve benchmarks**. These are very basic benchmarks models which have minimal features and parameters. They serve as the lowest bar for which your main forecast model should outperform. If they don't, then, due to their simple form, these benchmarks should be able to suggest improvements to the current model or indicate flaws in the chosen model. A selection of several of these simple benchmarks can also highlight some of the most important features in the underlying data. At least one of the benchmarks in a forecast trial should be simple.
2. **Common benchmarks**. Different applications will have some models which are commonly used as benchmarks. For example, this could be ARIMAX or simple linear regression models (Forecast models will be introduced in detail in Chap. 9). This can be helpful since it allows some degree of comparison between different models across different experiments even though the underlying data or situation is completely different.
3. **State-of-the-art benchmarks**. Often it will be desirable to compare to the current best methods available and implement a version of the state-of-the-art in the selection of different models. Even if the model doesn't quite outperform the best in the business, confidence can be given to a model which performs similarly to models which been tried-and-tested and shown to work well over several experiments and data sets. In many cases it may be difficult to identify any single model which performs well in general and instead at least one, well-known, competitive model should be chosen for comparison in your experiment.

In addition to choosing a naïve or common model, a simple way to choose a benchmark is to base them on at least one feature/relationship which appears to be important for the dependent variable of interest. In load forecasting, there is often weekly or daily seasonalities, and therefore it is common to pick a benchmark model which includes these features. Several common benchmark methods for load forecasting will be introduced in Sect. 9.1.

It should be highlighted that just because a model has the smallest error there is no guarantee it will achieve the best performance when used within the chosen application. However, it is often not computationally viable to assess the model by testing each forecast model in the chosen application (e.g. storage control as introduced in Sect. 15.1). That is why it is important to carefully select the forecast error metric which reflects the aims of the forecast (see Chap. 7).

8.1.2 Bias-Variance Tradeoff

The **bias-variance** tradeoff is one of the single most important concepts in creating an accurate forecast. As seen in Sect. 5.2 and Eq. (5.27), a time series forecast is essentially a function which takes various inputs to give the desired outputs. The nature of the function is determined by a number of parameters which must be trained on historical data. How to properly choose and train the parameters can have a large impact on the overall accuracy of the forecast.

As introduced in Chap. 4 machine learning was defined as algorithms that learn from data to improve prediction performance. However, there is no practical value if a machine learning model is only capable of predicting accurately based on instances from the data it was trained on. Here, a model that simply memorised all the training data can, in theory, achieve perfect performance. However, this is meaningless for all practical problems, as it is typically infeasible that all possible inputs can be measured (e.g. if the variables are real-valued). Therefore, the central challenge is to train a machine learning model that performs well on new, previously unseen inputs. The ability to perform well on previously unobserved inputs is called **generalisation**.

At the one extreme it may be desirable to choose a model with a large number of parameters and train it so it fits very closely to the training data. However, the more parameters, the more likely the model is to fit to spurious noise in the time series signal and hence cannot be extrapolated very well to new data. This is often called **overfitting** the model to the data. In this case, small changes in the input to the model will produce large errors and hence the model is said to have **high variance**. A high variance model does not generalise well to new data. In contrast a model with very few parameters will miss some of the core features of the time series and **underfit** the data. It means that on average the errors will be quite large and the model is said to have **high bias**.

The bias and variance can be expressed in more precise mathematical terms. Consider a model which relates the true relationship between a dependent value L (e.g. Load), and an independent variable Z (say temperature), via a function f

$$\hat{L} = f(Z) + \epsilon, \tag{8.1}$$

with noise ϵ (with assumed zero mean). The aim is to develop a model $\hat{f}(Z, \boldsymbol{\beta})$ that estimates the true function $f(Z)$, by learning the parameters $\boldsymbol{\beta}$ over some training data of observed values. Now suppose this estimate is produced by minimising the mean squared error, a common error measure for time series forecasts (see Chap. 7),

$$MSE(f(Z), \hat{f}(Z, \boldsymbol{\beta})) = \mathbb{E}[(f(Z) - \hat{f}(Z, \boldsymbol{\beta}))^2]. \tag{8.2}$$

It turns out that this can be broken down as follows

$$\mathbb{E}[(f(Z)-\hat{f}(Z,\boldsymbol{\beta}))^2)] = (\mathbb{E}[\hat{f}(Z,\boldsymbol{\beta})]-f(Z))^2 + \mathbb{E}[(\hat{f}(Z,\boldsymbol{\beta})-\mathbb{E}[\hat{f}(Z,\boldsymbol{\beta})])^2] + \sigma^2, \tag{8.3}$$

where σ^2 is the variance of ϵ. The first term is the square of the bias $(\mathbb{E}[\hat{f}(Z,\boldsymbol{\beta})]-f(Z))^2$ and describes the difference between the model output and the output from the true function. The bias term will be large if the model is too simple to capture the pattern in the data. The second term $\mathbb{E}[(\hat{f}(Z,\boldsymbol{\beta})-\mathbb{E}[\hat{f}(Z,\boldsymbol{\beta})])^2]$ is the variance, and describe the dispersion of the model outputs around the mean. In practical terms this measures how spread out the errors are around the mean. Finally there is the **irreducible error** defined by σ^2. This is the error that can never be reduced which limits how much the MSE can be reduced. The key to producing consistently accurate forecasts is to get low bias and low variance, i.e. a model which captures the main features in the data but also generalises well to new data.

As an example consider a simple model $y = x^3 - 15x^2 + 66x - 60 + \epsilon$, with irreducible errors, ϵ, which is chosen to be Gaussian with mean zero and variance $\sigma^2 = 20$. This function takes as input x, and gives the observed outputs, y. Points are generated from the true model $x^3 - 15x^2 + 66x - 60$ to give pairs of input-outputs (x_k, y_k) for $k = 1, \ldots, 50$. To replicate a real system, random error samples from the Gaussian distribution, ϵ_k, are added to each true dependent variable, y_k, to give observed points $\hat{y}_k = y_k + \epsilon_k$, for $k = 1, \ldots, 50$. Hence the true outputs are unknown to the modeller who only sees the inputs with the noisy outputs, i.e. $(x_k, \hat{y}_k)$. This means it will be impossible to create a perfect match between any model and the original observations.

Now consider fitting polynomials of different orders to these noisy points. The first model is a simple linear one of the form, $a_1x + a_0$, this is an underparameterised model and is expected to have high bias but low variance. The second model is a cubic polynomial of the form $a_3x^3 + a_2x^2 + a_1x + a_0$, and should be a good balance between matching the general shape of the data without overfitting the noise. The final model is a polynomial of the order 20, i.e. of the form $a_{20}x^{20} + a_{19}x^{19} + \cdots a_2x^2 + a_1x + a_0$ which would be expected to overfit to the data and thus have high variance. In each case the coefficients (The $a_i's$) are trained to find the best fit to the points for that model (how to train the fit will be covered in Sect. 8.2).

The results are shown in Fig. 8.1 for the three different models. As expected the best fitting model is the cubic model which is very close to the original curve and the noisy observations (red circles). The highly parameterised polynomial of degree 20 clearly overfits to the noise and will not generalise (i.e. will not estimate the correct output) very well to new inputs, it has high variance. In contrast the linear polynomial on the left does not fit the data very well but gets the general level. It has high bias and clearly underfits the data.

There are several strategies for avoiding over- or under-fitting the data. Several techniques will be introduced in this chapter, including cross-validation in the following section, regularisation methods (Sect. 8.2.4) and information criterion (Sect. 8.2.2).

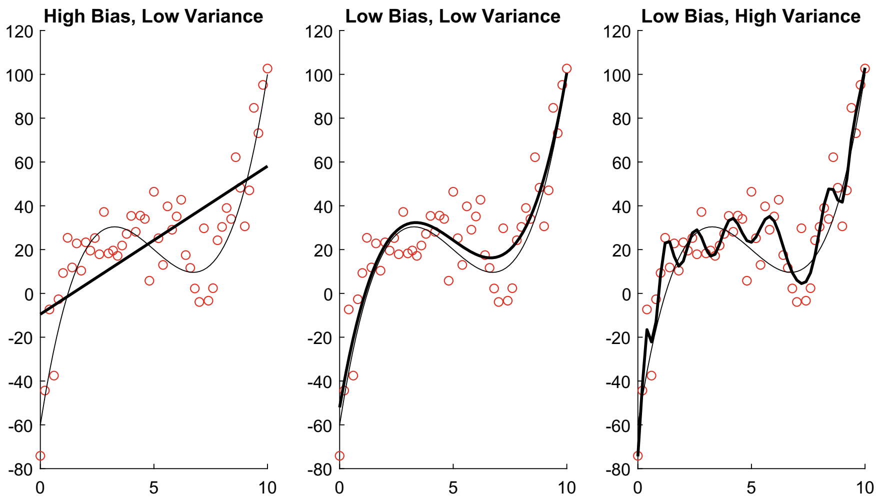

Fig. 8.1 Example of bias-variance trade-off by fitting different polynomials (thick black curve) to the observations (red circles). The true polynomial which generated the data is also shown as a thin black line. The left hand plot shows the fitting using a polynomial of degree 1, the middle plot using a polynomial model of degree 3 and the right plot using a polynomial of degree 20. Full details are in the main text

8.1.3 Cross-Validation Methods

In order to understand how a forecast model will perform in practice the available training data must be split into appropriate components. Time series data, the focus of this book, adds an extra potential restriction due to the chronological order of the data. This section will discuss some of the motivations and principles of cross-validation.

Forecast models must be tested on unseen data to ensure that the forecaster is not unrealistically tailoring (subconsciously or otherwise) the model to score higher than would be possible in practice. In real applications the future data is not available and forecasters would not have the advantages of knowing the actual values in advance. Hence designing a forecasting trial is very much like designing a blind experiment in medicine in order to test a particular hypothesis for whether a treatment is effective or not.

Another, related, reason for splitting the data is to choose a model with a good bias-variance trade-off (see Sect. 8.1.2). **Cross-validation**, the topic of this section, is one way to select a model so that it is not over- or under-fitted to the data, i.e. that it generalises well to unseen data.

For these reasons the data in machine learning trials is split into a unseen part, called the **test set**, or **hold-out set**, and a part for training the parameters and hyper-parameters of your model, often called the **training set**. For time series, the ordering of the data is often relevant and hence the test set typically follows chronologically from the training set. We will call this a **time-series split** (other approaches will be discussed shortly). For example, consider a time series $L_1, L_2, \ldots L_{N+k}$, for $N, k > 1$ and suppose the aim is to produce 1-step ahead forecasts for the test set consisting of data at the time steps $N+1, N+2, \ldots, N+k$. Any data (including any other explanatory variables, see Sect. 5.2) prior to the start of the test set is part of the training data. The following steps are then implemented:

1. The first forecast value $\hat{L}_{N+1}$ is produced by training a model (see Eq. (5.27)) on the current training data $L_1, L_2, \ldots L_N$ (as well as any other explanatory data).
2. The next step ahead forecast $\hat{L}_{N+2}$ is then produced by retraining[1] the data on $L_1, L_2, \ldots L_{N+1}$ (i.e. the last observation at $N+1$ is now included in the new training data).
3. This continues until the kth time step of the test period has been reached.

Note if a multistep ahead point forecast is being produced from a one-step ahead forecast then instead forecasts from the previous time steps are used as inputs to the model rather than the actual observations, e.g. for the mth step ahead the forecast would use as inputs $L_1, L_2, \ldots, L_N, \hat{L}_{N+1}, \ldots, \hat{L}_{N+m-1}$.

To find the most accurate forecast, a large number of models could be trained on the training set and the the errors on their predictions could be compared. However, this is often computationally infeasible. Further, the trialing of a large number of models increases the possibility that one particular forecast will have high performance by chance alone rather than due to its particular suitability for predicting the data behaviour. Hence, it is more practical and reliable to test a relatively small number of models. Recall in Sect. 8.1.2, that a core goal when creating a forecast is to balance the bias and variance, and find a model which accurately generalises to new data. This means not overfitting to the training data set by using a very complicated model, but also not using a very simple model which under-fits the data.

One of the most common ways to do this is to split another set, called the **validation set** off the end of the training set, and use this to help select a well-trained model and to select appropriate **hyperparameters** (Sect. 8.2.3). Hyperparameters are parameters of the algorithm which have to be chosen before the remaining parameters such as weights are determined in training and influence the training. Examples are the regularisation parameter used in regularisation methods in Sect. 8.2.4, and the number of layers and nodes in artificial neural networks (Sect. 10.4). The original shortened data set is now simply renamed the training data.

To use the validation data set for model selection, a family of models, are fit on the training data with different hyperparameter values (for example for neural networks, the number of nodes in each layer, number of layers, see Sect. 10.4) are used to generate a forecast for the data in the validation dataset (i.e. in a similar way as they are generated for a test set). The performance of the models are then compared, e.g. using the error measures introduced in Chap. 7, and the best performing models are selected to produce forecasts on the test set, often after being retrained on the combined training and validation set. The additional testing of the models introduced

[1] The model may not have to be retrained at every new time step, especially if it retains the same accuracy and if it is too expensive to retrain.

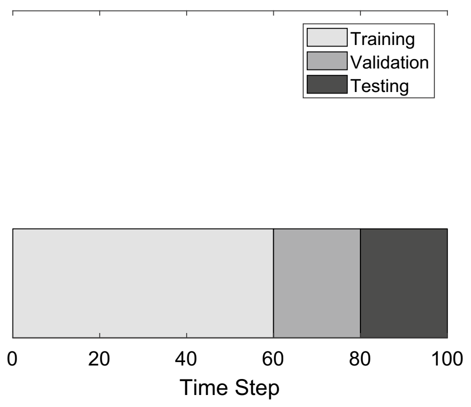

Fig. 8.2 Example illustrating a timeseries split into Training, Validation and Test set in a 3:1:1 ratio

by using a validation set improves the bias-variance tradeoff by eliminating models which over- or under-train. In addition, good performance on both the validation and test set increases confidence with using these models. Further hyperparameter selection methods are considered in Sect. 8.2.3.

A common split of the data is about 60%, 20%, 20% for the training, validation and testing set respectively, although this can be adjusted depending on the problem. As mentioned, more testing data increases the confidence in the performance of a model but sufficient training data is required to improve the chances that the models generalise as much as possible. An illustration of splitting the data into a training, validation and testing set is shown in Fig. 8.2.[2]

Given the above split of the data the following is a typical procedure to generate forecasts:

1. Given a family (or families) of forecast models, and suitable benchmarks (Sect. 8.1.1), train the model parameters on the training data.
2. Produce a forecasts over the validation set and compare the models (using the appropriate error measures as will be introduced in Chap. 7) within the same family to select a set of optimal values for the hyperparameters.
3. On the selected models (which may include one or two choices of hyperparameters for each family of models), re-train the models on the combined training and validation set.
4. Produce a forecast for the test set.

To illustrate the process consider a simple example. Take the model $y = \sin(x) + 0.2\sin(2x) + 0.4\sin(4x)$ which is used to generate values y_k at $x_k = 0.2(k-1)$ for $k = 1, 2, \ldots, 100$ to give 100 points (x_k, y_k). To make the data more realistic add random samples from a Gaussian distribution with mean 0 and variance 0.4 to the y

[2] Note there are ways to train models without using a validation set, for example, automatic model selection using information criteria (see Sect. 8.2.2) and this can be preferable if the amount of data available for training is quite low.

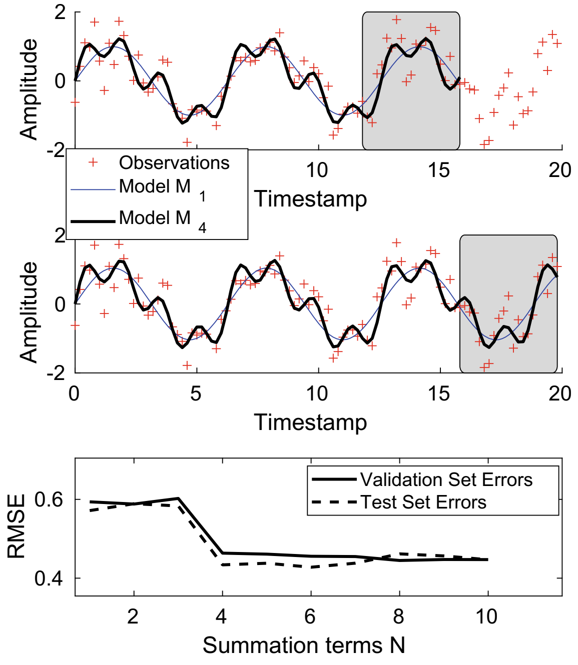

Fig. 8.3 Illustration of validation and testing for a simple example as described in the text. **a** Shows the forecasts of model M_1 and M_4 for the validation set (shaded), **b** Shows the same models for the test set. **c** Summarises the RMSE errors for the models M_N for $N = 1, \ldots, 10$ (Eq. 8.4) for both the validation and the test set

values to produce the updated input-output pairs $(x_k, \hat{y}_k)$. Next, consider fitting a set of models to the noisy observations, of the form

$$M_N(x) = \sum_{n=1}^{N} a_n \sin(nx), \tag{8.4}$$

for $N = 1, 2, \ldots, 10$ which for larger values includes more higher frequency terms to the model. Note that N is a hyperparameter for this family of models. Let the first 60% of the points be the training set (defined at $x = 0, 0.2, \ldots, 11.8$), the next 20% as the validation set ($x = 12, 12.2, \ldots, 15.8$) and the final 20% as the test set (points $x = 16, 16.2, \ldots, 19.8$). The models are trained for each N on the training data set.

Figure 8.3a shows the models M_1 and M_4 trained on the training set including its prediction on the validation set (shaded box). The noisy observations are shown as red circles. It's clear from this plot that the simplest model M_1 captures the main periodic behaviour but misses the higher frequency oscillations. In contrast the model M_4 (which matches the order of the true model) is much more accurate, as would be expected. Similarly Fig. 8.3b shows the prediction on the test set for the same models. This prediction has now been formed by training on the original training data set together with the validation set.

The RMSE errors (Chap. 7) for the ten models are shown in Fig. 8.3c for both the validation and test set. The accuracy drops down for $N = 4$ for both validation and

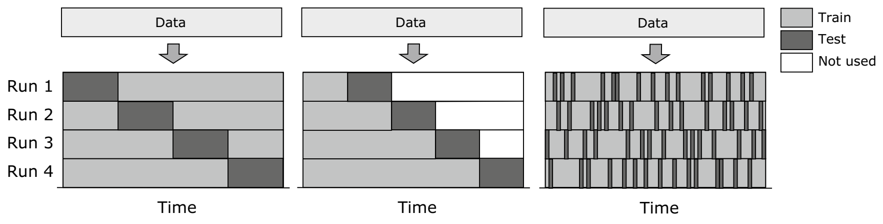

Fig. 8.4 Comparison of various cross-validation schemes. Figure from From Tutorial 1: Building Load Forecasting with ML licensed under CC BY 4.0

test set. This is because the complexity of the models required a periodic term of at least $\sin(4x)$. In addition, for these models the a_1, a_2, a_4 coefficients have the largest magnitude, as would be expected since they coincide with the terms in the actual underlying model, $y = \sin(x) + 0.2\sin(2x) + 0.4\sin(4x)$. Notice that the errors on the test set are smaller than on the validation set which could be because there is more training data available to better refine the coefficients. Although all models have been applied to the validation and test set, in practice only the best performing models may be carried forward to forecast on the test set, especially if the models are computationally expensive to train.

There is several other cross-validation methods which split the data in different ways. These are illustrated in Fig. 8.4. In the middle is the Time-Series split as illustrated in detail above but also shown are the **blocked** split (left) and **shuffled** split (right). The blocked approach splits the data into test blocks (typically of the same size) however, unlike the Time-Series split, this split uses data before and after the test block to train the data. The shuffled split, uses random samples to generate the test and training sets.

The blocked and shuffled splits have the advantage of utilising more data with which to train the models but for time series problems this may be less realistic since data is not typically available after the period of interest. Hence these approaches may be more appropriate when considering cross-sectional models or for a time-series model when data availability is quite limited and it is difficult to properly train using the time-series split.

Finally it is worth noting, that if there is insufficient data available, it may be that no choice of splits will be appropriate to create an accurate forecast. For example consider day or week ahead forecasts for hourly time series data which is known to have annual seasonality (e.g. as is usually the case with electricity demand). If there is only one year of data available it will be unlikely you could train a model which will be able to accurately capture the annual seasonality. Even if there is two years of data this could still be difficult since one of those years could have been a particularly unusual year and may not be representative of a typical year which the forecaster is trying to model (for example, consider the 'Beast from the East' an unusual cold wave occurring in Great Britain in 2018). However, it obviously may not be known ahead of time whether there is insufficient data for an accurate forecast and may only become apparent as more testing is performed and more data becomes available.

8.2 Training and Selecting Models

Once the data has been pre-processed (Sect. 6.1), features to include have been decided (Sect. 6.2.2) and the initial forecast model(s) are chosen (See Chap. 9), the parameters/coefficients of the models must be trained and the final models selected. This is often done via cross-validation (see Sect. 8.1.3). This section dives deeper into how to train a forecast model as well as further techniques for selecting accurate models.

The performance of the predictions on the instances in the training set are called the training error. Reducing this training error is typically one part of the optimisation task usually solved by gradient descent-type methods (Sect. 4.3). However, what separates machine learning from simple optimisation is that a **generalisation error** should be minimised as well. The generalisation error is defined as the expected value of the error on *new input*, ideally from the distribution of inputs we expect the algorithm to encounter in practice. To simulate this, in the process of training a model and tuning its hyperparameters, the generalisation error is estimated by assessing its performance on a test set of examples that were collected separately from the training set. When creating a test set (or several test sets as in cross validation as discussed in more detail in Sect. 8.1.3) the following so called i.i.d. assumptions are made:

- the examples in each dataset are **i**ndependent from each other,
- the training set and test set are **i**dentically **d**istributed.

Under these assumptions one can expect, that the expected training error is equal to the expected test error for a *random* model. However, in practice since the parameters of a model are chosen to minimise the training set error, the expected test error is larger than (or equal to) the expected value of the training error. Thus in order to achieve an accurate model that generalises well, a model should minimise the training error, but at the same time minimise the gap between the training and test set error. Figure 8.5 illustrates the typical relationship between training and generalisation error against model capacities (i.e. model complexity).

This suggests another way to understand the bias-variance trade-off alluded to in Sect. 8.1.2. If a model is not able to sufficiently minimise the training error then this corresponds to a model which **underfits** the data/observations. Alternatively, if the gap between test and training error is too large, it is **overfitting** the data. These situations are illustrated in Fig. 8.5. Generally, one can control over- and underfitting by trading off variance and bias and this can be determined using cross-validation methods as described in Sect. 8.1.3 but also regularisation methods as described in Sect. 8.2.4.

There are several ways a particular model can be trained but some models typically use specific approaches. For example, linear regression models (Sect. 9.3) will often use least-squares estimation, whereas artificial neural networks (Sect. 10.4) will often be solved via back-propagation techniques. When implementing a particular forecast model in a standard programming package they will be trained based on those techniques which have been shown to be most suitable or typical for that method. To

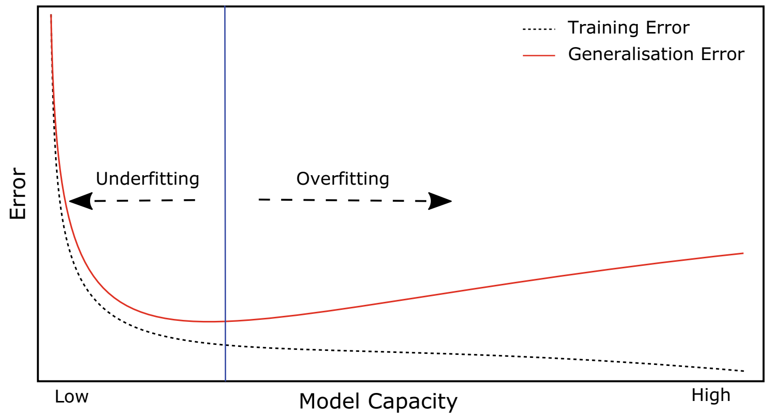

Fig. 8.5 Typical relationship between model capacity (e.g. complexity, or number of parameters) and train and generalisation error

illustrate some of the principles of training a forecast model the following sections consider some common techniques such as least-squares estimation and, maximum likelihood estimation, but also describes some more principles and strategies to control the generalisation error of machine learning models, in particular **regularisation** techniques. Regularisation refers to strategies and model modifications that intend to reduce the generalisation error (but not the training error). In practice multiple strategies are combined.

8.2.1 Least-Squares and Maximum Likelihood Model Fitting

The general aim of training will be to find a good fit between the model and the observations. However, as noted in the previous sections if too many parameters are chosen then the model may overfit on the training set and not generalise very well to the test set (or any other unseen data). A model with features that have been selected appropriately will have a good trade-off between bias and variance and perform better on the test set (see Sect. 8.1.2).

What is a deemed a 'good' fit is relatively subjective but requires a consistent measure of the difference between the observation and models. The best choice is often a balance between practical considerations and what is most appropriate for the application being considered (for example the control of storage devices, as in Sect. 15.1), and therefore models should be evaluated and tested accordingly. Good candidates for such measures are the p-norms introduced in Chap. 7.

As discussed in the corresponding evaluation section (Chap. 7), the choice of p can change the focus of the errors, with larger p values meaning the final error score is more representative of the larger errors compared to p-norms with smaller p.

Further, some norms, such as the 1-norm are not differentiable, which make it more difficult to train the optimal parameters compared to differentiable functions which can be optimised by gradient methods (see Sect. 4.3). The 2-norm is differentiable and is therefore often used as the basis of parameter estimation, in particular it is the core measure used in so-called **least-squares estimation**.

Least-squares estimation (LSE) is one of the most common methods for training parameters, especially for linear regression models. The aim is to minimise the square of the residuals. Recall the residuals are the difference between the real observation, say Y_k, and the model estimate, say $\hat{Y}_k = f_k(Z, \boldsymbol{\beta})$ where Z represents the input variables and $\boldsymbol{\beta}$ are the set of parameters for the model being estimated. The least squares problem can be written

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta} \in \mathcal{B}} \sum_{k=1}^{N} (Y_k - f_k(Z, \boldsymbol{\beta}))^2, \tag{8.5}$$

where $r_k = Y_k - f_k(Z, \boldsymbol{\beta})$ are the residuals. The term arg min simply means finding the *arguments* (parameters) over some feasible set of values (in this case represented by $\mathcal{B}$ and defines the space of reasonable values that $\boldsymbol{\beta}$ could take) which minimises the equation $\sum_{k=1}^{N}(Y_{n+k} - f_{n+k}(Z, \boldsymbol{\beta}))^2$. The process is illustrated in Fig. 8.6 which shows the best fitting line to a set of observations. The vertical distance between the observations and the line shown on the plot are the residuals for the model

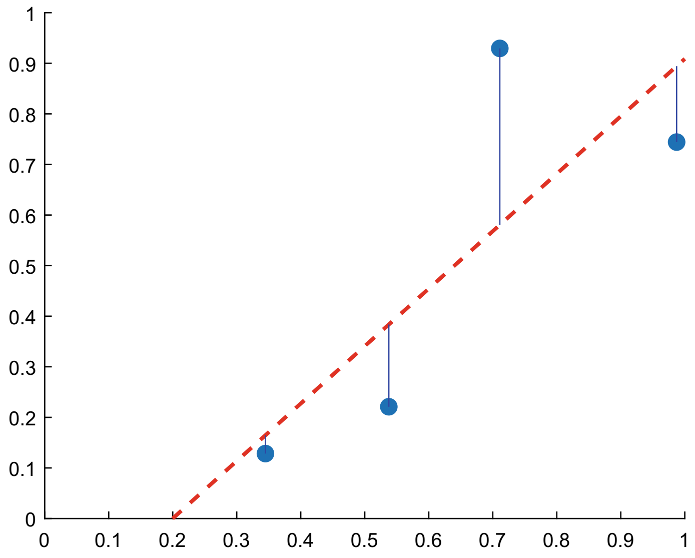

Fig. 8.6 Illustration of least squares estimation for fitting a one variable linear model (dotted red line) to four data points (blue dots). The least squares fit minimises the sum of the squares of the vertical residuals

and the aim in least squares estimation is to minimise the sum of the *square* of these residuals. An advantage of the least squares method is that it is relatively easy to solve, especially for linear models (i.e. those which are linear combinations of parameters—$\sum_{k=1}^{N} \alpha_k \phi_k(x)$ where the α_k's are constant coefficients/parameters to be found), this is because the least squares problem in Eq. (8.5) is differentiable and hence can be solved by differentiating with respect to each element in the set of parameters $\boldsymbol{\beta}$ and setting to zero.

One of the most important statistical methods for training model parameters is by **maximum likelihood estimation** (MLE). This involves setting the model errors within a probabilistic framework which allows further statistical analysis and application of further methods (for example, in Sect. 8.2.2 it will be shown how this enables a method for model selection). The aim of MLE is to create a likelihood function, based on a density function describing the distribution of errors of the model fit. Hence maximising this function finds the *most likely* parameters which minimises the error given the assumed distribution of values. The resultant parameters are called maximum likelihood estimates.

The following discussion will require some basics on univariate probability distributions. To illustrate MLE consider a basic case where the errors follow a normal distribution (See Sect. 3.1), with mean *zero* and fixed standard deviation σ. Assume finite observation data given by $Y_1, \ldots, Y_N$ and a model $f_k(Z, \boldsymbol{\beta})$ which aims to approximate these observations. The errors are given by the residuals $r_k = Y_k - f_k(Z, \boldsymbol{\beta})$ as before and the probability model can be written as

$$P(r_k, \boldsymbol{\beta}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(r_k - 0)^2}{\sigma^2}\right), \tag{8.6}$$

where the zero is written explicitly to illustrate the standard Gaussian distribution format. The $\boldsymbol{\beta}$ (which in this case simply is the standard deviation σ) is also included to show the dependence of the value on the parameters in the model. The likelihood function is in fact the probability of the model with all observations, in other words

$$L(\beta; Y_1, \ldots, Y_N) = \prod_{k=1}^{N} P(r_k, \boldsymbol{\beta}) = \frac{1}{(\sqrt{2\pi}\sigma)^N} \prod_{k=1}^{N} \exp\left(\frac{-(r_k)^2}{\sigma^2}\right). \tag{8.7}$$

Often it is difficult to solve the likelihood directly, so instead the loglikelihood is solved, which is just the log of the likelihood

$$log(L(\beta; Y_1, \ldots, Y_N)) = -\frac{N}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{k=1}^{N} r_k^2. \tag{8.8}$$

Since the log function is a strictly increasing function, maximising the likelihood is equivalent to maximising the loglikelihood. Further since in this case the first term $\frac{N}{2}\log(2\pi\sigma^2)$ is constant then this is equivalent to minimising the term $\sum_{k=1}^{N} r_k^2$, in other words, solving the least squares problem in Eq. (8.5). Of course this is just for a very specific situation where the distribution is assumed to be Gaussian but more complicated distributions can also be considered but they are more difficult to solve analytically.

Both the least squares and maximum likelihood function are examples of **cost functions** which provides a cost between the observed data Y_k and the model estimates $f_k(Z, \boldsymbol{\beta})$. Standard p-norms type errors such as RMSE and MAPE (See Chap. 7) can also be used to define basic cost functions. The cost functions can be very general with different weightings and/or structures which can force the model to fit to different features of the data. For example, as shown in Chap. 7 the pinball loss score can be used as a cost function to produce a quantile estimate whereas using the least squares can produce an estimate of the expected value. The aim, as with the least squares and MLE is to optimise the parameters $\boldsymbol{\beta}$ of the model to achieve the optimal value of the cost function. A common way to find the optimal fit is through gradient methods which were introduced in Sect. 4.3 and these are commonly deployed when fitting machine learning models.

As discussed in Sect. 8.1.2, overfitting the model to the data will produce large generalisation errors. One way to avoid this is to use cross-validation and use a validation set to choose models which don't overfit (Sect. 8.1.3). However, in some cases alternative techniques are often employed to reduce the effect of overfitting a forecast model. These are explored in the following sections.

8.2.2 Information Criterion

The likelihood framework introduced above facilitates the use of information criterion for model selection and is particularly useful for selecting the model orders for ARIMA methods (Sect. 9.4) and for selecting amongst linear regression models. Consider the likelihood function, L, e.g. such as that as in Eq. (8.7) when the residuals are assumed to be Gaussian distributed, evaluated at the maximum likelihood estimated parameters $\hat{\boldsymbol{\beta}}$ which in turn provides the maximum likelihood value $\hat{L}$. In this case the best fit can be framed in terms of the information they provide through the so-called Information Criterion, the most famous examples are the Bayesian Information Criterion (BIC) defined as

$$BIC = M.\ln(N) - 2\ln(\hat{L}) \tag{8.9}$$

and the Akaike Information Criterion (AIC) defined as

$$AIC = 2M - 2\ln(\hat{L}), \tag{8.10}$$

where M is the number of parameters in the model, and N is the number of observations. Both of these create a cost function which is a mix of the loglikelihood (a measure of the fit between the data and the model) and a penalty term which penalises the number of parameters in the model. Hence by optimising the information criterion a balance is made between a good fit and the number of inputs in the model, and reduces the chance of overfitting the model to the data. Such a model is called **parsimonious**. As can be seen from the equations the AIC penalises the likelihood more than the BIC and therefore typically optimises models with lower number of parameters. For predictive modelling the AIC has been suggested to be more appropriate than the BIC [1]. Note there are several other information criterions but the BIC and AIC are the most common.

For models which can be set within a likelihood framework, information criterion methods are often used instead of a validation set (Sect. 8.1.3), especially where there is insufficient training data available. However, if there is sufficient amount of data, and a test can be created which sufficiently represents a reasonable sample of real observations, then cross-validation may be preferable.

8.2.3 Hyper-Parameter Tuning

The main objective of optimising a machine learning model is to find an optimal set of parameters, like the weights of a neural network, or the coefficients in a linear regression. However, some parameters have to be chosen that influence the optimisation itself, like the different parameters that have been introduced in other sections like step size, batch size, activations functions and regularisation parameters (Sects. 8.2.4 and 8.2.5). Additionally, models may introduce even more parameters like the architecture of a neural network (number of layers and number of neurons per layer), or the order of the polynomials in a polynomial regression. Such parameters are called **hyperparameters.**

In Sect. 8.2.2 information criterion was shown to be a way to select the optimal order for linear models. For example, by selecting the model with the minimum AIC (or BIC) a more parsimonious model can be chosen which does not sacrifice the model fit. However, information criteria models are not applicable to other types of models such as neural networks or tree-based models.

For simpler models with few hyperparameters, a common approach is to exhaustively search the best configuration among a grid of sensible parameters within each dimension. Real-valued parameters are typically sampled linearly or logarithmically across the feasible values. However, parameters can also be categorical or binary. This approach is referred to as **grid search**. However, this approach is impractical when tuning many hyperparameters of a large neural network, where it may take several hours or even days and weeks to train. Here, one can randomly sample from each dimension of the hyperparameters. This approach is called **random search**, which has been shown to be superior to grid search, especially when only a small number of hyperparameters affect the model performance, i.e., the optimisation problem has a low intrinsic dimensionality. Random search has a further advantage that it is an **any-time** algorithm, as one can stop the algorithm after a specific calculation budget (i.e., a specific number of draws or computation time) is reached. The best solution

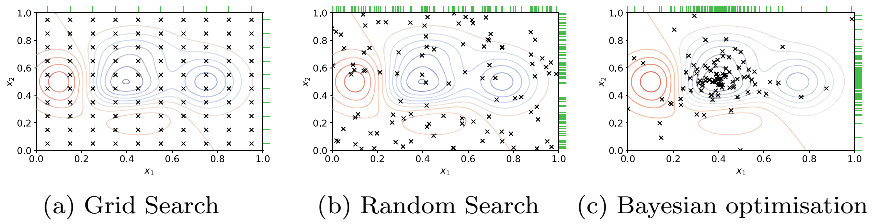

Fig. 8.7 The exemplary performance of a model based on two hyperparameters. For each hyperparameter, ten different values are evaluated and compared. Blue contours indicate regions with strong results, whereas red ones show poor results. *Source* Alexander Elvers, CC BY-SA 4.0

found during the search is then selected. It is also straightforward to parallelise. By choosing a specific distribution, one can also include prior knowledge to help focus the search.

Since hyperparameter selection is an optimisation problem, different general optimisation methods, including meta-heuristic methods, such as evolutionary algorithms and other population-based algorithms, can be used. A popular and successful class of such algorithms are **Bayesian optimisation** approaches. Bayesian optimisation builds a probabilistic model of how hyperparameter values map to model performance determined over a validation set. The probabilistic assumptions are iteratively updated to include more and more information about the search space as it becomes available. Algorithms differ in how they balance the **exploration** of hyperparameters for which the outcome is uncertain and **exploitation** is increased by sampling hyperparameters that are expected to be close to the optimum. Figure 8.7 shows examples of how grid search, random search and Bayesian optimisation sample the search space. Here one can see that Bayesian optimisation does explore the search space more effectively. There is a denser cluster of observations close to the true optimum, while fewer points are sampled in regions that perform poorly. This can generally lead to better or similar results than grid and random search but in quicker time.

8.2.4 Weight Regularisation

Much like the information criterion presented in Sect. 8.2.2, regularisation also uses a penalty based on the number of parameters to try and reduce overfitting and, as will be seen with LASSO, can also be used for feature selection. The regularisation methods presented here are typically applied to linear regression and artificial neural networks (Sect. 10.4) models, although the principles can be generalised to any cost function.

The principle is best demonstrated with an example. Consider a linear model for $n \geq 1$ input/dependent variables $X_{1,t}, X_{2,t}, \ldots, X_{n,t}$ which are assumed to be linearly related to the load L_t at time t. This can be written

$$L_{N+1} = \sum_{k=1}^{n} \beta_k X_{k,N+1}. \tag{8.11}$$

For compactness, the linear model $\sum_{k=1}^{n} \beta_k X_{k,N+1}$ can be written as the matrix-vector multiplication $\mathbf{X}\boldsymbol{\beta}$ where $\mathbf{X} \in \mathbb{R}^{N \times n}$ is the matrix where the ith row and jth column corresponds to the ith time step for the jth variable, i.e. $X_{j,i}$. Finally, $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_n)^T \in \mathbb{R}^n$ is the vector of parameters. Given a vector of dependent variables $\mathbf{L} = (L_1, L_2, \ldots, L_n)^T$, a regularised least squares regression can be written as

$$\min_{\boldsymbol{\beta}}(||\mathbf{L} - \mathbf{X}\boldsymbol{\beta}||_2 + \lambda||\boldsymbol{\beta}||_p) \tag{8.12}$$

for some hyperparameter $\lambda \geq 0$ (also known as a Lagrangian). Recall from Sect. 7.1 that $||.||_p$ represents the p-norm. This hyperparameter must be found via the validation set as defined in Sect. 8.1.3 and controls the size of the penalty on the coefficients. If $\lambda = 0$ then the problem reduces to the standard least-squares estimation, Eq. (8.5). For large values of λ the parameters become small. The most common forms used are either $p = 1, 2$ the so-called **LASSO** (least absolute shrinkage and selection operator) or ridge regression respectively. LASSO is particularly popular at it can reduce the number of inputs as it will often set many of the coefficients in a linear regression to zero due to the 1-norm penalty. In this case a large number of unimportant inputs can be eliminated. Hence LASSO can be used for both training a model and feature selection.

To understand why LASSO can be used for feature selection consider the illustration in Fig. 8.8. The figure compares the ridge regression with the LASSO regression. Both plots show the contours (lines of the same value) of the least squares cost function within the parameter space $\boldsymbol{\beta} = (\beta_1, \beta_2)^T$ (assuming only 2 dimensional problem). In the centre of these contours is the least squares estimate $\hat{\boldsymbol{\beta}}$, i.e. the parameter values which gives the smallest values of the least squares cost function.

In LASSO or ridge regression, there is effectively a penalty on the size of the parameters and this penalty changes based on the size of the λ hyperparameter. The larger the λ the smaller the values of the parameters. Thus the parameters are constrained within a bounded area of size $0 < C \in \mathbb{R}$ given

$$||\boldsymbol{\beta}||_p = (|\beta_1|^p + |\beta_1|^p)^{1/p} \leq C, \tag{8.13}$$

where $p = 1$ or 2 depending on whether LASSO or Ridge regression is being considered. These constrained regions are shown as the shaded areas in Fig. 8.8. Notice that due to the different norm values used the shapes are very different. The constrained region in the 2-norm (Ridge regression) is a spherical ball, but with the

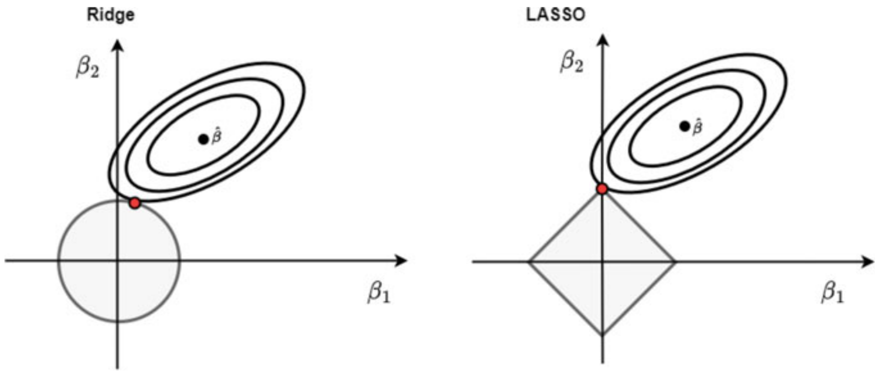

Fig. 8.8 Demonstration of how LASSO regularisation (right) can be used to select features. This is compared to ridge regression (left). The chosen parameters are represented by the red dot showing the smallest value of the least-squares cost function with respect to the feasible parameters (the shaded shape). The figure is explained in the main text

1-norm (LASSO) is a square. The minimum value for the regularised cost function is therefore given by the $\boldsymbol{\beta}$ within the shaded feasible region closest to the optimal least squares estimate $\hat{\boldsymbol{\beta}}$, shown as the red dot in the figure.

Due to the shape of the constrained region for LASSO regression this will often lie on the 'corners' of this region, which means that often many of the parameters will be equal to zero, effectively selecting the parameters in the process.

8.2.5 *Other Regularisation Methods*

Reducing Model Capacity For many algorithms one can control with hyperparameters whether a model is more likely to over- or underfit by altering its capacity, i.e., its complexity or more generally its available degrees of freedom. By choosing certain hyperparameters (see Sect. 8.2.3) the hypothesis space, i.e., the set of functions that the algorithm is capable of selecting as a solution, is affected. For instance, for neural network models, the number of trainable parameters determines the ability to fit a wide variety of functions. Similarly, in random forests, the number of trees determines its complexity. Models with low capacity may not be able to properly fit the training set (they have high bias). Models with high capacity will overtrain on the training data set and don't generalise to the actual underlying process (hence are not represented in the test set).

Data Augmentation Overfitting can generally be avoided by training a model on more data. In practice the amount of data may be small or data collection can be expensive. Instead, one way to improve model performance is to create synthetic data samples in the training set. For image data this can often easily be achieved by adding transformations to images like mirroring, rotating, shifting, as problems

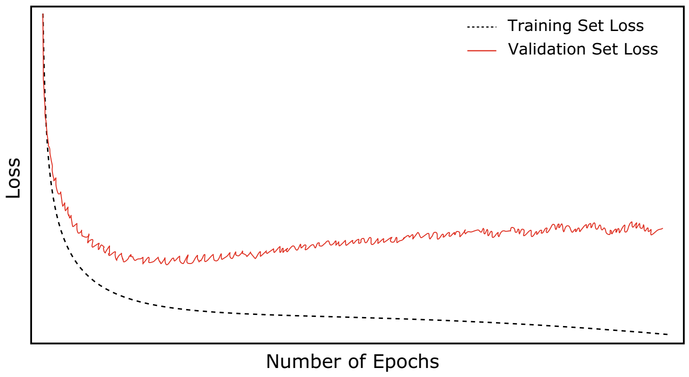

Fig. 8.9 Exemplary learning curve to show relationship of training and evaluation loss over time in training process for a model with high capacity

such as object detection are expected to be invariant to such operations. This is not trivial for many machine learning tasks, in particular time series forecast where the real patterns and interdependencies in the data need to be preserved. However, where possible, data augmenting strategies should be explored.

Early Stopping Training neural network models with large capacity on tasks which are too simple can lead to overfitting. One popular diagnostic tool to prevent this are **learning curves**, i.e., the calculation of the error on the training set at regular intervals in the training process (e.g., after each training epoch). If the hyperparameters are chosen reasonably well the training error should decrease. To monitor generalisation in the training process, one can create a validation dataset and calculate the errors. If the validation error increases, while the training error decreases this is an indication of the model starting to overfit. Figure 8.9 gives an exemplary learning curve of a model with high capacity. Thus one popular regularisation strategy is to stop training if the validation error does not improve beyond a specific number of iterations. At the end of the process, the model that has the smallest validation error is returned rather than the final model configuration. This requires the algorithm to store checkpoints of the model configurations during the training process (e.g., the values of the weights in a neural networks).

Batch Normalisation Another popular improvement to the training process of neural networks is batch normalisation, or batch norm. It was introduced as a method to speed up the training of neural networks and make it more stable by normalising each of the layers' inputs by re-centering and re-scaling (standardising). However, besides providing faster and more stable training, batch normalisation also has a regularising effect. Further, the training becomes more robust to different initialisation schemes and the choice of the learning rates (i.e., a larger learning rate can be chosen).

Dropout In dropout a certain share of artificial neurons and its weights are randomly omitted during the training process of a neural network (Sect. 10.4). This process effectively creates an ensemble of simpler neural network architectures. This is related to ensemble methods such as random forests that combine the predictions of simple decision trees (see Sect. 10.3.2 on random forests). Dropout has the effect of adding noise to the training process. It has been shown that a reasonable default for a wide range of tasks is to use a dropout of 0.5 for each layer. Dropout can be used and configured for each layer of the neural network, and works with different kinds of layers such as dense fully connected layers, but also convolutional and recurrent layers (see Sects. 10.5 and 10.4). However, it should not be used in the output layer. When adding dropout only to the input layer, this is related to the idea of adding noise as it has been used in denoising autoencoders. It is computationally cheap and an effective regularisation method to reduce overfitting and improve the generalisation error in many kinds of deep neural networks.

8.3 Questions

1. Create your own bias variance experiment. You could repeat the polynomial fit in Fig. 8.1. Alternatively choose another polynomial of a different degree. Generate 100 samples from the polynomial and add noise (say with a Gaussian distribution). Now fit polynomials of a variety of degrees, say from 1 to 20. Calculate the training errors. Plot the training errors for each polynomial as a function of degree. How does the error change? Is the smallest error at the correct polynomial degree? For higher degrees does the error increase? Now resample the polynomial (and add noise with the same distribution as before). Measure the error between the fitted polynomials and this new data? This is the generalisation error. Plot the errors against degree again. What is the optimal degree? Compare this plot with the original one with the training errors. What is the difference between them?
2. Repeat the above experiment but sample just 15 points this time. How do the training and generalisation errors change? What about reducing the number of sampled points to 5?
3. Perform your own grid search. Generate points from a simple model, say a line $y = ax + b$, with known coefficients a, b. Add a small amount of noise to the points. Select a rectangle around the coefficients, i.e. $(a, b) \in [A_1, A_2] \times [B_1, B_2]$. Generate a grid of $N_1 \times N_2$ points (say $N_1 = N_2 = 10$) within this rectangle by simply choosing uniformly spaced values on each side of the rectangle, i.e. the kth a value $a_k = A_1 + (k-1)\frac{(A_2-A_1)}{N_1-1}$, similarly $b_l = A_2 + (l-1)\frac{(A_2-B_2)}{N_2-1}$ for $k = 1, \dots, N_1$, and $l = 1, \dots, N_2$. For each pair of coefficients in the rectangle calculate the errors between the sampled data and the associated line. Which pair of coefficients give the lowest errors? How close are they to the true values? In addition, sample random pairs from within the rectangle. How many samples did you need to produce smaller errors than the grid search.
4. Show that the mean squared error for a model can be broken down in bias, variance and irreducible error as in Eq. 8.3.

Reference

1. G. Shmueli, To explain or to predict? Stat. Sci. **25**, 289–310 (2010)