Actuaries Institute.

Fellowship Program

Data Science Applications

Chapter 5: Classification and Neural Networks

Table of Contents

5. Classification and Neural Networks

This chapter covers the following learning objectives:

Item	Learning Objectives
2.	**Develop data science solutions to business problems using a variety of tools and techniques**
2.1.	**Develop solutions to classification problems**
2.1.1.	Explain what a neural network is
2.1.2.	Explain how a neural network can be used as an alternative to GLMs or tree-based models to solve classification problems
2.1.3.	Develop solutions to a range of classification problems using GLMs, tree-based models, ensembling and neural networks
2.1.4.	Evaluate solutions produced by classification models

5.1. Introduction

Regression and classification modelling were previously introduced in the Risk Modelling and Survival Analysis, and Data Science Principles subjects. To recap, regression and classification are both types of supervised learning methods, where data is used to train models to make predictions of chosen output variables. These output variables are sometimes called ‘responses’, ‘labels’, ‘targets’, or ‘dependent variables’. They are generally referred to as ‘responses’ in this subject.

A response variable will be at the unit of analysis, which may be at a different level to the data observations, a situation that frequently occurs when your data is multiple tables in a relational database rather than a single data table. The unit of analysis will always be at the level of an entity (occasionally more than one entity) and a point in time. An entity is a real-world object or concept that is represented by fields in the source tables, e.g. a customer. When choosing the unit of analysis and response variable, consider the following requirements:

- The selected entity must align with the business problem being modelled. For example, customer churn is analysed at the customer entity level, but monthly sales will be analysed at sales territory level.
- A response variable may be an attribute at a point of time, e.g. whether they have an active subscription, or an aggregation over a future time period, e.g. total purchases over the next month.
- Response variables should not include past known data, e.g. use future purchases, not lifetime cumulative purchases.
- To separate age from cohort effects, you will need observations of the same entity at multiple points of time.
- If your response variable is an aggregation of future events, leave that period out when partitioning data into training and validation, e.g. if your response variable is purchases over the next month, then there must be a one-month gap between the end of the training period and the start of the validation period.

The term ‘supervised’ refers to the fact that the training data contains responses that have been observed and these are used to train the model. A model’s predicted responses are compared to the actual responses from the training data, which allows the model to learn from the data in a supervised way.

Regression models are used to predict response variables that are continuous. For example, a regression model might try to predict:

- the demand for electricity in an area in the coming month, based on features such as the expected population size, weather forecasts and the price of electricity;
- the cost of insurance claims in a year, based on features such as insurance policyholder characteristics, past levels of claims and expected inflation rates; or
- a customer's propensity to buy a product, based on features such as the product price, the customer's past spending patterns, and other characteristics of the customer.

In contrast, classification models are used to predict discrete or categorical responses. For a predictive modelling task, classification involves using a vector of input variables to predict the likelihood that a response variable belongs to one of two or more classes. For example, a classification model might be used to predict:

- whether a patient has cancer, based on the colour of pixels contained in that patient's medical scans;
- whether a person is expected to develop diabetes, based on their demographic and health characteristics; or
- which plan a person will choose in a telecommunication company's product range, based on various characteristics that the company has recorded about that person.

A special case of classification is when the number of categories of interest is two. For example, there are two categories of interest when trying to predict whether a person will renew their insurance offer ('will renew' and 'will not renew'). This 'two-class' or 'binary' classification situation is common and allows certain algorithms to be used, such as logistic regression. The use of logistic regression also optionally allows the response variable to be treated as a continuous variable within the range of 0 and 1, and the use of loss functions or techniques that are usually reserved for regression. This concept of loss functions will be explored further in Section 5.2.

While regression and classification modelling have been described above as two separate types of modelling, many regression problems can be recast as classification problems. For example, rather than estimating a person's income (a regression problem), a model could instead estimate what income band they fall into (a classification problem). Therefore, regression and classification modelling can sometimes be used to solve similar problems.

Exercise 5.1

Consider each of the following data science tasks:

- using the intensity of gene expressions to predict the type of cancer a person is suffering;
- using suburb and house characteristics to predict the market value of a home;
- using customer characteristics to predict whether they will click on a particular internet advertisement; and
- using car sensor data to identify whether a pedestrian is walking across the road ahead.

Identify the response variable for each of the tasks above.

State whether each task requires a classification or regression predictive model.

Regression techniques are covered in detail in the Data Science Principles subject. Therefore, this chapter concentrates on classification modelling and, in particular, the use of neural networks to solve classification problems.

First, the chapter will provide a refresher on classification concepts and the following methods that you have learned in your previous studies:

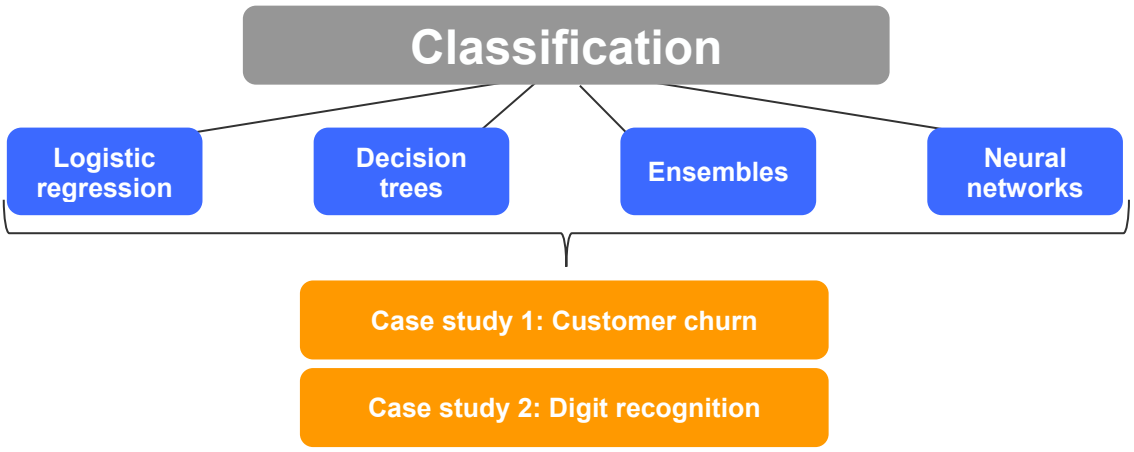
- logistic regression—a subset of generalised linear modelling;
- decision trees; and
- ensembles—bagging and boosting.

The chapter will then explore the use of neural networks as an alternative to the above classification techniques. The theory behind neural networks will be explained. Two case studies will then be presented to:

- demonstrate how neural networks can be applied in practice; and
- compare the effectiveness of different classification techniques in solving different types of problems.

These key components of the chapter are summarised in Figure 5.1.

Figure 5.1 – Chapter mind map



For each of the classification techniques covered in this chapter, mathematical formulae are provided to explain the theory behind the technique. You should not get too caught up in these formulae. They are there to help explain the theory, so you can better:

- understand what the relevant package in Python is doing;
- interpret its results; and
- work out what is happening if the model does not perform as expected.

You should note that while neural networks are explained in this chapter in a classification setting, they are equally amenable to solving regression problems. The adjustments that are needed when using neural networks in a regression setting are outlined in Section 5.5.6.

The exercises and case studies in this chapter use various Python libraries to apply the techniques that are taught. For example, Keras is used to build neural networks in Python. Alternatives to these libraries are available.

For interested students, a fuller treatment of the statistical aspects of classification models, including neural networks, can be found in the following resources:

- James, Witten, Hastie, and Tibshirani (2017), *An Introduction to Statistical Learning with Applications in R*, chapter 4;[1]
- Hastie, Tibshirani and Friedman (2008), *The Elements of Statistical Learning*, chapter 11;[2] and
- Goodfellow, Bengio, and Courville (2016), *Deep Learning*.[3]

[1] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2017). *An Introduction to Statistical Learning*. Springer.
[2] Hastie, T., Tibshirani, R., and Friedman, J. (2008). *The Elements of Statistical Learning*, Springer.
[3] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

5.1.1. Notation

This section outlines the notation that is used in this chapter and the following chapters for this subject.

Each data table has:

- n distinct data points ('observations'); and
- p input variables ('features') available for making predictions.

Note that the input variables are referred to as 'features' in this subject. Outside this subject, they are sometimes also referred to as 'explanatory' or 'independent' variables, or 'covariates'.

For convenience, assume that all features are numeric variables. Standard techniques exist for converting categorical variables to numeric variables, such as one-hot encoding, which is discussed in Section 5.5.5.

- x_{ij} represents the value of the j^{th} feature for the i^{th} observation, with $i = 1,2, \dots, n$ and $j = 1,2, \dots, p$;
- y_i represents the value of the response variable for the i^{th} observation;
- X is the $n \times p$ design matrix, where x_{ij} denotes element (i, j);
- $X_{i.} = (x_{i1}, \dots, x_{ip})$ represents the i^{th} observation vector (row);
- $X_{.j} = (x_{1j}, \dots, x_{nj})^T$ represents the j^{th} feature vector (column); and
- Y is the n-vector, where y_i denotes the i^{th} element.

Rows in X represent observations and columns in X are features.

For example, consider the dataset shown in Table 5.1.

Table 5.1 – A dataset called 'spam'

Observation	word_freq_all	char_freq_$	capital_run_length_total	spam_flag
1	0.64	0.000	278	1
2	0.50	0.180	1,028	1
3	0.71	0.184	2,259	1
4	0.30	0.000	118	0
5	0.00	0.000	78	0
6	0.65	0.000	40	0

In Table 5.1:

- each observation represents an email;
- 'word_freq_all' is the percentage of words in each email that are 'all';
- 'char_freq_$' is the percentage of characters in each email that are '$';
- 'capital_run_length_total' is the total number of capital letters in each email; and
- 'spam_flag' is 1 if the email has been classified as spam and 0 if it has been classified as not spam.

The dataset shown in Table 5.1 is an excerpt from the full spam dataset described in Section 5.1.2.

For the dataset shown in Table 5.1, the response variable is 'spam_flag'. Here:

- $n = 6$ (the number of observations);
- $p = 3$ (the number of features: 'word_freq_all', 'char_freq_$' and 'capital_run_length_total');
- x_{23} = 1,028 (the 3rd feature, 'capital_run_length_total', for the 2nd observation); and
- y_5 = 0 (the response for the 5th observation).

In general classification problems:

- $G = \{g_1, g_2, \ldots, g_K\}$ denotes the set of K possible class outcomes; and
- $Y = (y_1, y_2, \ldots y_n)^T$ is the response vector of classes observed for each observation, where y_i.is an element of G

For example, in Table 5.1, the response variable 'spam_flag' can take one of two values: 0 or 1. Therefore:

- $G = \{non - spam, spam\}$;
- $K = 2$; and
- $Y = (1, 1, 1, 0, 0, 0)^T$.

A classification model ('classifier') is a function $G(X_{i.})$ that outputs a value in G.

Many classifiers produce an underlying probability model that is used to classify each observation into one of the K classes. Let $P(y_i = g_k|X_{i.})$ denote the probability that an observation with features $X_{i.}$ will have the response g_k. A classification can then be made based on the estimated values of $P(y_i = g_k|X_{i.})$ as outlined below.

In this chapter:

- g_i is the class to which observation i belongs;
- $\hat{p}_{i,g_k}$ is the estimated value, for observation i, of $P(y_i = g_k|X_{i.})$;
- $\hat{P}_{g_k} = \left(\hat{p}_{1,g_k}, \hat{p}_{2,g_k}, \ldots, \hat{p}_{n,g_k}\right)^T$ is the vector of all $\hat{p}_{i,g_k}$ for the vector of observations X;
- $\hat{y}_{i,g_k}$ is the raw output of a classification model for observation i and class g_k; and
- $\hat{Y}_{g_k} = \left(\hat{y}_{1,g_k}, \hat{y}_{2,g_k}, \ldots, \hat{y}_{n,g_k}\right)^T$ is the vector of all $\hat{y}_{i,g_k}$ for the vector of observations X.

As shown above, strictly speaking, the values that G takes $(g_1, g_2, \ldots, g_K)$ are the actual possible class outcomes, such as 'spam' or 'non-spam'. However, in practice, it is often convenient to represent these actual possible class outcomes with digits. For example, rather than expressing G as taking values of 'non-spam' or 'spam', G might instead take values of 0 (representing non-spam) or 1 (representing spam). In this case, $g_i = y_i$. This latter approach is the one taken in the remainder of this chapter.

It is important to note that the raw outputs of a classification model, $\hat{Y}_{g_k}$, can represent different things, depending on the classifier used. $\hat{Y}_{g_k}$might represent:

- $\hat{P}_{g_k}$, the estimated probabilities that each observation in X will have the response g_k;
- something close to $\hat{P}_{g_k}$, but for which the individual $\hat{p}_{i,g_k}$s are not strictly probability estimates as the values of $\hat{y}_{i,g_k}$, for each observation i, do not add to 1 across all K classes; or
- values in G (i.e. actual class outcomes taking values of g_1, g_2, g_3, etc.).

These different types of raw outputs from a classifier will make more sense as you study the different classification models discussed throughout this chapter.

If $\hat{Y}_{g_k}$represents values in G, then no further transformations are required and the classifier's job is done. An example of this type of classifier output is provided in the 'XNOR Gate' example in Section 5.5.3.

If $\hat{Y}_{g_k}$represents $\hat{P}_{g_k}$or something close to $\hat{P}_{g_k}$, then a transformation of the raw output from the model is required. In this case, for each observation i, a classification, $G(X_{i.})$, can be made by selecting the class, g_k, that maximises $\hat{y}_{i,g_k}$. This can be expressed mathematically as follows:

Equation 5.1 $$G(X_{i.}) = \boldsymbol{arg\ max}_{g_k}\ \hat{y}_{i,g_k}$$

where the max' of a function is the value of the domain (in this instance, g_k) at which the function is maximised.

For example, assume that a classifier has three classes, g_1, g_2, and g_3, and the raw outputs from the model for observation i are as follows:

- $\hat{y}_{i,g_1} = 0.25$;
- $\hat{y}_{i,g_2} = 0.35$; and
- $\hat{y}_{i,g_3} = 0.40$.

In this example, the sum of $\hat{y}_{i,g_1}, \hat{y}_{i,g_2}$ and $\hat{y}_{i,g_3}$ is 1, and $\hat{y}_{i,g_k}$ represents $\hat{p}_{i,g_k}$, the estimated probability that observation i belongs to class g_k. In this example, $G(X_{i.})$ = g_3 because g_3 is the class for which $\hat{y}_{i,g_k}$ has the highest value (0.40).

As noted above, some classifiers' raw outputs, $\hat{Y}_{g_k}$, are not strictly estimates of $P(y_i = g_k|X_{i.})$ because the values of $\hat{y}_{i,g_k}$, for each observation i, do not add to 1 across all K classes. However, these $\hat{Y}_{g_k}$model outputs can, nonetheless, be used to arrive at a classification using Equation 5.1 by finding the value of g_k, for each observation i, that maximises $\hat{y}_{i,g_k}$. This concept is explained further in Section 5.5.2 in the context of neural networks.

5.1.2. Two Example Datasets

The following two datasets are used in examples provided throughout this chapter.

Toy Dataset

The toy dataset is a synthetic dataset that contains:

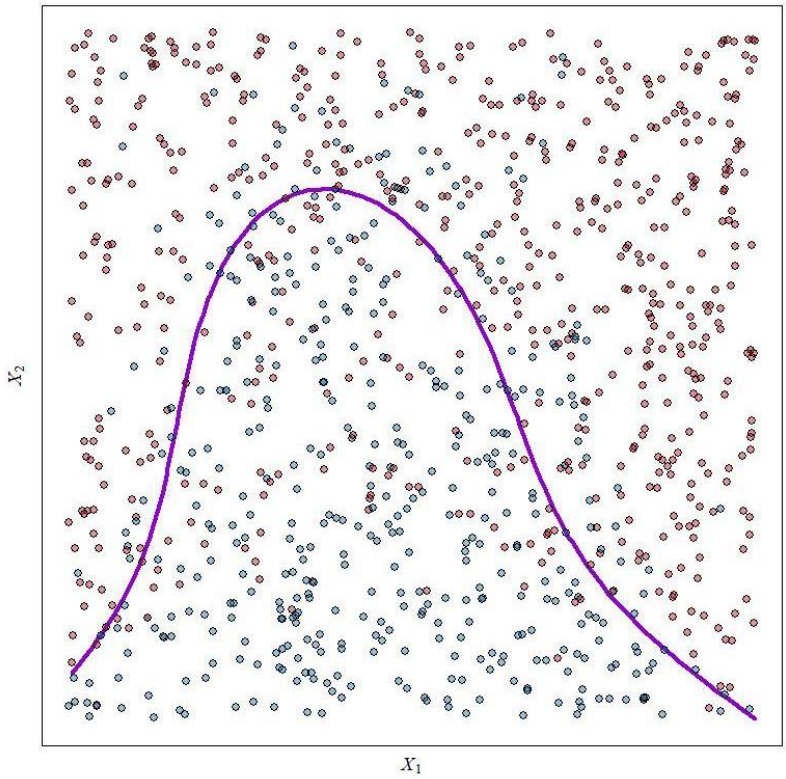
- 1,000 observations that were produced via simulation;
- two features, $X_{.1}$ and $X_{.2}$, that were sampled from a random standard uniform distribution (with range 0 to 1);
- a response variable for each observation that indicates whether the observation is 'blue' (response = 0) or 'red' (response = 1).

Each observation was assigned to the 'blue' or 'red' class by:

1. applying a nonlinear function, $f(X_{.1}, X_{.2})$, to $X_{.1}$ and $X_{.2}$ where $f(X_{.1}, X_{.2}) = 10 \times \left(-0.2 + \left(0.4{X_{.1}}^3 + 0.3{X_{.1}}^2 X_{.2} - 0.1 X_{.1} {X_{.2}}^2 + 0.3{X_{.2}}^3 + 0.3{X_{.1}}^6 + 0.5(1 - X_{.1})^6 - 0.5(1 - X_{.2})^8\right)\right)$;
2. applying the sigmoid function, $S(x) = \frac{1}{1+e^{-x}}$, to $f(X_{.1}, X_{.2})$ (i.e. $S(f(X_{.1}, X_{.2})) = \frac{1}{1+e^{-f(X_{.1}, X_{.2})}}$) to produce probabilities of belonging to the 'red' class for each observation; and
3. using the probabilities from the previous step to sample from a random Bernoulli distribution.

Figure 5.2 shows the 1,000 observations from this dataset and the true boundary line, which is shown in purple. The true boundary line represents points for which an observation is equally likely to be blue or red ($S(f(X_{.1}, X_{.2})) = 0.5$).

Figure 5.2 – Observations from the toy dataset



Spam Dataset

The spam dataset is sourced from the University of California, Irvine Machine Learning Repository.[4] This dataset contains the following:

- 4,601 observations, each representing an email originally collected from a Hewlett-Packard email server, of which 1,813 (39%) were identified as spam;
- 57 continuous features:
  - 48 features of type 'word_freq_WORD' that represent the percentage (0 to 100%) of words in the email that match 'WORD';
  - 6 features of type 'char_freq_CHAR' that represent the percentage (0 to 100%) of characters in the email that match 'CHAR';
  - 1 feature, 'capital_run_length_average', that is the average length of uninterrupted sequences of capital letters in the email;
  - 1 feature, 'capital_run_length_longest', that is the length of the longest uninterrupted sequence of capital letters in the email; and
  - 1 feature, 'capital_run_length_total', that is the total number of capital letters in the email; and
  - a binary response variable that takes on a value 0 if the email is not spam and 1 if the email is spam.

[4] Hopkins, M., Reeber, E., Forman, G., and Suermondt, J. (1999). *Spambase Data Set* [Dataset]. https://archive.ics.uci.edu/ml/datasets/Spambase

5.2. Measures of Success and Diagnostics

5.2.1. Introduction

This section will review some statistical measures and model diagnostics that can be used to assess how well different classifiers perform in their predictions. When evaluating models, it is also important to consider the commercial outcomes together with these statistical measures of success.

A model may meet statistical significance tests or outperform random chance, but may still fall short of business requirements. For example, if false positives lead to significant customer harm, a low false positive rate would likely be required.

Areas where the model is inaccurate may also be important from a commercial perspective. For example, an image recognition model may perform well in training data overall but underperform in specific culturally-sensitive contexts.

Business processes can be also designed to mitigate risks that may arise from inaccurate predictions. For example, consider a model that recommends whether to accept or decline an insurance claim, where incorrectly declining a claim is considered to be a detrimental customer outcome. If the model is predictive but not sufficiently accurate, a process could be implemented that involves human review of decisions before a claim is declined. This may be sufficient to mitigate the downside outcomes from inaccurate predictions, while realising the automation benefits from the use of the predictive model.

These contextual points should be considered together with the statistical measures and diagnostics outlined in this section.

5.2.2. Loss Functions

A predictive model is made up of:

- data (features and a response variable);
- a model that uses the features to predict the response variable; and
- a loss function.

A loss function represents the price paid for the inaccuracy of a model's predictions. The loss function is the measure by which a model is judged and optimised. A better classifier will produce a lower loss function score than a poorer model.

Common loss functions that are used for classification models are shown in Table 5.2 and discussed further below.

Table 5.2 – Common classification loss functions

Loss function	Formula
Misclassification rate (0-1 loss)	$\frac{1}{n}\sum_{i=1}^{n} I\{g_i \neq G(X_{i.})\}$ where $I(\cdot)$ is the indicator function that takes the value 1 if the condition is true and 0 otherwise
Misclassification rate with class costs	$\frac{1}{n}\sum_{i=1}^{n} C_{g_i} I\{g_i \neq G(X_{i.})\}$ where C_{g_i} is the cost or weight penalty for misclassifying observation i
Logistic loss	$-\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} I\{g_i = g_k\} \log \hat{y}_{i,g_k}$
A special case of logistic loss for binary classification	$-\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y}_{i,1}) + (1-y_i)\log(1-\hat{y}_{i,1})$ where y_i can take values of 0 or 1
Mean squared error (for binary classification)	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_{i,1})^2$ where y_i can take values of 0 or 1

Misclassification Rate

The simplest classification loss function is misclassification rate. This is also known as 0-1 loss. The misclassification rate for each observation takes the value 1 for an incorrect prediction and 0 for a correct prediction.

The opposite of the misclassification rate is the accuracy of the model (i.e. 1 – misclassification rate).

An important idea is that even if the true probability function, $P(y_i = g_k|X_{i.})$, is known for every observation, there will still usually be a nonzero misclassification rate. For example, when flipping two fair coins the most likely outcome—occurring half the time—is one head and one tail. However, if this prediction is made, there is still an expected misclassification rate of 50%. This rate is better than if two heads or two tails was predicted (each with a 75% expected misclassification rate) but the expected misclassification rate is still nonzero.

The 'Bayes error' rate refers to the expected misclassification rate obtained when the most likely option is predicted, given the predictors $X_{i.}$. That is, the class for which $P(y_i = g_k|X_{i.})$ is highest is selected. Such a selection is referred to as the 'Bayes optimal classifier'. This is an important concept since it represents the best possible expected misclassification rate—no model can be expected to do better than this on a large set of unseen observations.

When the misclassification rate is used as a loss function, the aspiration is to build a model whose error approaches the Bayes error rate. However, in practice, the true value of $P(y_i = g_k|X_{i.})$ is not known, so it can only be estimated.

Exercise 5.2

A continuously running production line produces faulty widgets 30% of the time when Manager A manages from Monday to Friday, and 60% of the time when Manager B manages on the weekend.

Calculate the error rate when a Bayes optimal classifier is used to predict whether a widget is faulty or not.

Misclassification Rate with Class Costs

An important practical consideration is the impact of incorrect predictions. A variation of misclassification rate is misclassification rate **with class costs.** This loss function imposes different costs, C_{g_k}, on misclassifying an observation that belongs to class g_k. For example, when using an email spam classifier, an individual might be more concerned about missing genuine emails than receiving too much spam. In this case, a higher cost might be imposed on the classifier when it incorrectly classifies a legitimate email as spam than when it incorrectly classifies spam as a genuine email.

Therefore, the misclassification rate with class costs loss function allows the model's context to be explicitly allowed for when deciding on which classifier provides the 'best' results. For example, analysis of the costs of treating patients might indicate that the cost of misclassifying a sick person as healthy might be five times the cost of misclassifying a healthy person as sick.

When estimating the cost penalties to apply for each class, consideration should be given to factors such as the financial cost of an incorrect classification or the time cost for a person to remediate errors in the classifications. This is an important example of how business context can, and should, impact the model building process (see Chapter 2, Domain knowledge).

Assigning a higher cost, such as by teaching the model that a false negative is five times worse than a false positive, will tilt the classifier towards predicting certain classes.

Then, the best classifier is whichever class has the highest estimated probability times cost.

Equation 5.2 $$\boldsymbol{G(X_i) = arg\ max_{g_k}\ C_{g_k}\hat{p}_{i,g_k}}$$

Equation 5.2 may not, at first glance, seem intuitive, as why would you want to select a class with a high cost? The answer is that C_{g_k} is the cost of *misclassifying* an observation that belongs to class g_k. Therefore, if C_{g_k} is very high, the classifier should be 'reluctant' to assign an observation to a class other than g_k as doing so incorrectly will impose a high cost. Therefore, Equation 5.2 has the impact of selecting a class with a high cost unless the estimated probability that an observation belongs to a different class is very high. This is illustrated through the following example.

Imagine that you are building a classifier to identify spam, where 0 indicates that an email is not spam and 1 indicates spam. Assume that missing a genuine email has eight times the cost of receiving a spam email, such that the chosen costs might be $C_0 = 8$ and $C_1 = 1$. These costs might, for example, reflect an estimated eight times greater loss in a workplace's productivity if important emails are not read and actioned compared to a small estimated loss of one person's time in having to skim-read and delete a spam email. Table 5.3 sets out different scenarios for this example and the classification that will be made under each.

Table 5.3 – Misclassification rate with cost example

Estimated probability that the email is not spam $\hat{p}_{i,0}$	Estimated probability that the email is spam $\hat{p}_{i,1}$	$C_0\hat{p}_{i,0}$	$C_1\hat{p}_{i,1}$	$G(X_i)$
0.9	0.1	7.2 (8 x 0.9)	0.1 (1 x 0.1)	0 (not spam)
0.5	0.5	4.0 (8 x 0.5)	0.5 (1 x 0.5)	0 (not spam)
0.2	0.8	1.6 (8 x 0.2)	0.8 (1 x 0.8)	0 (not spam)
0.1	0.9	0.8 (8 x 0.1)	0.9 (1 x 0.9)	1 (spam)

In this example, the classifier will assign an observation as spam only if the estimated probability that the email is spam is higher than 88.89%.

Exercise 5.3

Describe some other examples of classification where having different costs across classes might be appropriate.

Exercise 5.4

Imagine that a hospital has built two classifiers to predict whether current hospital patients have an infection based on measures such as their temperature, blood pressure and reported symptoms. The results of the two classifiers for 1,000 patients are shown below:

		Predicted	
	Model A	No Infection	Infection
Actual	No Infection	630	50
	Infection	170	150

		Predicted	
	Model B	No Infection	Infection
Actual	No Infection	480	200
	Infection	70	250

Determine which model is 'better', based only on the information shown above.

Now imagine that the hospital has told you about the following financial costs related to their patients testing for and being found to have an infection:

Exercise 5.4

- Tests for an infection: $100 (for predicted and actual infections)
- Sanitising a room and moving a patient to a new one: $3,000 (for actual infections)
- Treating an infection early: $10,000 (for actual infections)
- Treating an infection later: $30,000 (for actual infections)

Determine which model is 'better', based on all information that is now available to you.

Note that this exercise is based on one presented on the following website: https://www.kdnuggets.com/2016/12/salford-costs-misclassifications.html.

Note that the section above presents misclassification costs, C_{g_k}, that are fixed for each *actual* class, g_k, regardless of the *predicted* class for each observation. In practice, it might make sense to vary the misclassification cost based on both the actual and predicted classes.

For example, imagine that an auction house builds a model to predict the results of its art auctions in $1,000 buckets. If the model makes an underprediction, the auction house's clients might be angry that they are unable to afford the artwork and might decide to take their business elsewhere, costing the auction house future business from these clients. In contrast, if the model makes an overprediction, it might put some customers off attending the auction, thinking they cannot afford the artwork, which might result in a lower achieved sales price. Therefore, the misclassification cost might differ depending on whether the predicted class under- or over-estimates the actual class. In this example, both the actual and predicted classes have an impact on the misclassification cost.

While the ability to incorporate a model's context into the selection of a classifier is an advantage, in practice, it can be difficult to quantify the actual costs of misclassification at the time of modelling. This is a disadvantage of using a misclassification rate with costs loss function.

The misclassification rate, whether used with or without costs assigned to classes, is 'all or nothing' in the sense that no credit is given for the relative certainty that a model has on a prediction.

Logistic Loss

Logistic loss is an alternative that gives some credit for this relative certainty. Logistic loss can be used for classification models that produce probabilities that an observation belongs to each class—$\hat{p}_{i,g_k}$—as opposed to classifiers that directly produce a class classification—$G(X_i)$. This often allows for more nuanced models, since it explicitly recognises that the 'right' answer for some predictions is uncertain. It is used extensively in binary classification and is the natural loss function for logistic regression. Logistic loss is also known as 'deviance', 'negative log-likelihood' or 'entropy'.

Mean-Squared Error

Mean-squared error (MSE) is a loss function that is more typically used for regression. However, as for logistic loss, MSE can apply to binary classification when the response for each observation, y_i, can only take values of 0 or 1 and when $\hat{y}_{i,1}$ represents $\hat{p}_{i,1}$ rather than $G(X_i)$.

Unlike the misclassification rate, mean-squared error gives credit for the relative certainty that a model has on a prediction. This is because it takes into account the model's estimated probability of belonging to class 1 ($\hat{y}_{i,1}$).

5.2.3. Confusion Matrices

A confusion matrix is a table that is often used to describe the performance of a classifier on a set of data for which the true values are known. In a confusion matrix, all the predictions are cross-tabulated so you can identify where misclassifications are occurring. Entries in each part of the matrix can be either the count of predictions or the percentage of total predictions.

Table 5.4 provides an example of a confusion matrix based on the spam dataset introduced in Section 5.1.2.

Table 5.4: Example confusion matrix

	Predicted: genuine	Predicted: spam	Total
Actual: genuine	2,137	651	2,788
Actual: spam	361	1,452	1,813
Total	2,498	2,103	4,601

The predictions that are summarised in the above confusion matrix were made using a simple rule: emails with a large character frequency of dollar signs (more than 4.5%) were classified as spam. Under this rule, 54% (2,498/4,601) of emails were judged to be genuine, but the rule is more likely to classify a genuine email as spam (14%=651/4,601) than spam as a genuine email (8%=361/4,601).

Exercise 5.5

Download the spam dataset and **construct** a simple predictor based on whether the 'char_freq_!' variable is greater than or equal to 0.045. You should confirm that you can recover the confusion matrix shown above.

Calculate the overall misclassification rate.

Different types of predictions are given specific labels in a confusion matrix, as shown in Table 5.5.

Table 5.5: Confusion matrix labels

	Predicted: class 0	Predicted: class 1	Total
Actual: class 0	true negative (tn)	false positive (fp)	$tn + fp$
Actual: class 1	false negative (fn)	true positive (tp)	$fn + tp$
Total	$tn + fn$	$fp + tp$	n

Models are preferred if they have larger values of true positives (tp) and true negatives (tn), and correspondingly smaller values of false positives (fp) and false negatives (fn). In other words, the accuracy of a classifier, which is measured as $(tn + tp)$ / $(tn + fp + fn + tp)$, is important. However, the inclusion of imbalanced classes in a dataset, where one class has much fewer observations than other classes, and the differing costs of false positives and false negatives motivate the use of several other metrics such as:

- precision = tp / $(tp + fp)$ = proportion of true 1s among the predicted 1s;
- sensitivity/recall (true positive rate) = tp / $(fn + tp)$ = proportion of true 1s among the actual 1s;
- specificity (true negative rate) = tn / $(tn + fp)$ = proportion of true 0s among the actual 0s;
- false negative rate = fn / $(fn + tp)$ = 1 – sensitivity; and
- false positive rate = fp / $(tn + fp)$ = 1 – specificity.

Exercise 5.6

Calculate the accuracy, precision, sensitivity, specificity, false negative rate, and false positive rate for the spam predictions shown in Table 5.4.

Another metric that is commonly used in relation to confusion matrices is the F1 measure, which is a harmonic average of precision and sensitivity/recall. It is defined as:

Equation 5.3

$$F_1 = \frac{2}{\frac{1}{precision}+\frac{1}{recall}} = \frac{2}{\frac{tp+fp}{tp}+\frac{tp+fn}{tp}} = \frac{2tp}{2tp+fp+fn}$$

It is worth noting that measures such as F1 are impacted by which class is chosen as '0' and, therefore, which class is chosen as '1'. It is typical to assign the class of interest (e.g. spam in the spam classification example) to class '1'.

In cases with many classes, it may be evident from the confusion matrices that certain groups of classes are most easily confused. For example, in an image classification task, dingoes and dogs may be more commonly mistaken than elephants and dogs. It may be that a model is very effective in identifying some classes, but poor in identifying others. This is important to understand both from a technical perspective, when trying to improve the accuracy of model predictions, and from a practical perspective, when considering the potential commercial outcomes and risks from deploying the model.

A slightly less serious example of images that classifiers may have trouble distinguishing between is shown in Figure 5.3.[5]

[5] Cave, J. (2016). Is This A Muffin Or A Chihuahua? *Huffington Post* https://www.huffingtonpost.com.au/entry/muffin-or-chihuahua_n_56e05c3ee4b065e2e3d461f2

Figure 5.3 – Chihuahua or muffin?

5.2.4. The ROC Curve

Classifiers typically produce outputs for each observation i and each class k, $\hat{y}_{i,g_k}$, that represent scores or probabilities, $\hat{p}_{i,g_k}$, where $\hat{p}_{i,g_k}$ is an estimate of $P(X_{i.})$ rather than a straight class prediction, $G(X_{i.})$. This means that there is not just one choice of predictions of classes. Depending on the end users' preferences for false positives or false negatives across different classes, these classifier scores can be used to produce different classifications. This can be achieved by setting a threshold value that is used to transform the estimated probabilities, $\hat{Y}_{g_k}$, into predicted classes, $G(X)$.

For example, in binary classification, if the classifier's score is higher than the threshold value, the observation is assigned to class 1. Otherwise, it is assigned to class 0.

The trade-off that is made between false positives and false negatives can be understood by considering the full range of different threshold selections available in a binary classification setting.

Let $f(X_{i.})$ be a function that produces a score, with higher values of $f(X_{i.})$ indicating that the observation is more likely, according to the model, to belong to class 1. Suppose the threshold for classifying an observation to class 1 or class 0 is varied. For example, the probability required for classifying an email as spam could be varied. At one extreme, all observations could be labelled as 0. This would correspond to setting the threshold for $f(X_{i.})$ at infinity. This method would obtain:

- a 0% false positive rate (perfect specificity) as no true 0s would be falsely classified as 1; but
- a 0% true positive rate (zero sensitivity) as no true 1s would be correctly classified as 1.

Conversely, classifying all cases as 1 by setting the threshold for $f(X_{i.})$ at minus infinity would give:

- a 100% true positive rate (perfect sensitivity) as all true 1s would be correctly classified as 1; but
- a 100% false positive rate (zero specificity) as all true 0s would be falsely classified as 1.

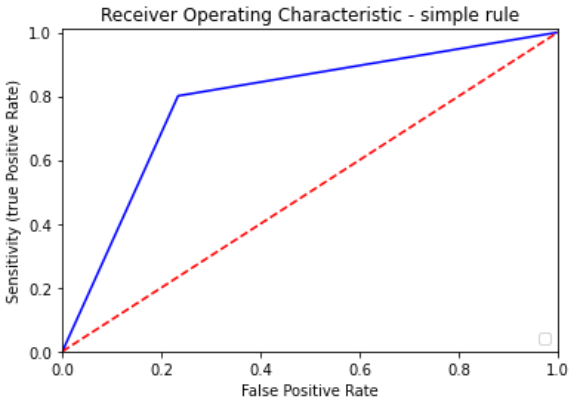
As the chosen threshold levels vary, intermediate values for sensitivity and specificity are obtained. A plot of the true positive rate (sensitivity) against the false positive rate (1-specificity) for different threshold levels is known as the **receiver operator characteristic** (ROC) curve. Each instance of a confusion matrix is represented as one point on the ROC curve.

For the spam example outlined above, a ROC curve that plots the true positive rate on the y-axis and the false positive rate on the x-axis can be constructed using the following three points:

- (0,0) if every email is classified as genuine;
- (1,1) if every email is classified as spam; and
- (0.234, 0.801) if emails are classified as spam when the frequency of the '!' character is greater than or equal to 0.045.

This ROC curve is shown in Figure 5.4.

Figure 5.4 – ROC curve for spam classifier



A perfect prediction would be represented on the ROC curve in the upper-left corner at the coordinate (0,1) (i.e. the area under the curve is exactly 1), representing 100% sensitivity (no false negatives) and 100% specificity (no false positives). In contrast, a random guess would give a point along the dotted red diagonal line in Figure 5.4 (i.e. the area under the curve is 0.5). An example of random guessing is when a decision is made by flipping unbiased coins. As the size of the sample increases, a random classifier's ROC point tends towards the diagonal line. In the case of a balanced coin, it will tend to the point (0.5, 0.5).

The diagonal line shown in Figure 5.4 divides the ROC space. Points above this line represent classification results that are better than random guesses whereas points below the line represent results that are worse than random guesses.

The area under the ROC curve (between the curve and the x-axis) is a common single-number diagnostic known as the **area under the curve** (AUC). It is generally a very robust measure with a small increase in the AUC representing a genuine improvement in the model. A perfect model will have an AUC of 1, whereas a random model will have an AUC of 0.5.

Exercise 5.7

Calculate the AUC for the simple spam classifier.

Exercise 5.8

A binary classifier produces the following predictions and actual experience when tested on a validation dataset.

Observation group	Actual number of observations in class 0 (g_1)	Actual number of observations in class 1 (g_2)	Prediction of $P(y_i = g_2 \mid X_i)$ (probability of belonging to class 1)
A	4	2	0.10
B	5	1	0.30
C	3	3	0.35
D	4	7	0.60
E	1	10	0.80
Total	**17**	**23**	

Construct a plot of the ROC curve. **Calculate** the AUC.

The ROC curve concept can also be used for multiclass classification by considering each class g_k in turn. The model's ability to predict class g_1 is plotted first, with all other classes grouped as class 0. Then, the model's ability to predict g_2 is plotted, and so on.

5.2.5. Validation and Generalisation Error

When a model is fitted to data, it aligns to the idiosyncrasies and noise in that data. This is particularly true of complex models that make use of many features. These complex models can be at risk of 'overfitting' to the data that they are trained on. One consequence of overfitting is that the performance of the model on the data used for model training, whether assessed using a loss function or other measures introduced above, will overstate the expected performance on new unseen data.

Further, many predictive models use validation data to optimise the model's hyperparameters. Hyperparameters are parameters that define the model's structure or architecture. 'Optimising' these hyperparameters means finding the set of hyperparameter values that are expected to maximise the model's performance on new unseen data. Since this validation data is used in building the model, performance on this model will also be biased to the validation data used. This idea is closely related to the concept of the bias-variance trade-off that is discussed in Section 5.4.1.

Training, validation and test data

To avoid overfitting to the data that a model has been trained on, best practice modelling involves dividing the data into:

- a **training** dataset that is used for the main fitting of the model;
- a **validation** dataset that is used for model tuning; and
- a **test** dataset that is reserved for the end of the modelling process to test how well the model generalises to unseen data.

It is important to note that the test dataset should be split out from the rest of the data as early as possible in the modelling process. This even includes splitting out the test data prior to any exploratory data analysis steps that might influence the model to be built. The test dataset should not be used until after the final model has been selected and is ready to be deployed. This allows a more accurate estimate to be made of how well the model will perform in production because the test data is truly 'unseen' in that it has in no way been used to build the model.

This process of splitting the data is depicted in Figure 5.5.

Figure 5.5 – Splitting data into training, validation and test datasets

There are no firm rules on the right ratios for these three components. In situations with plentiful data, a 50–25–25 split for training–validation–test data might be reasonable. However, in situations where reducing the training dataset degrades the model's performance, retaining a larger fraction for training might be necessary.

In addition, these days, it is not very common to just use one validation dataset unless:

- the dataset is very small; or
- the model is very expensive to fit but it needs a large amount of training data to be of any predictive value.

For example, for tree-based methods (see Section 5.3.2) that often require a large training dataset, a training-test split of 90-10 or 95-5 is common, with cross-validation (see below) used to further split the training data into training and validation datasets.

Cross-validation

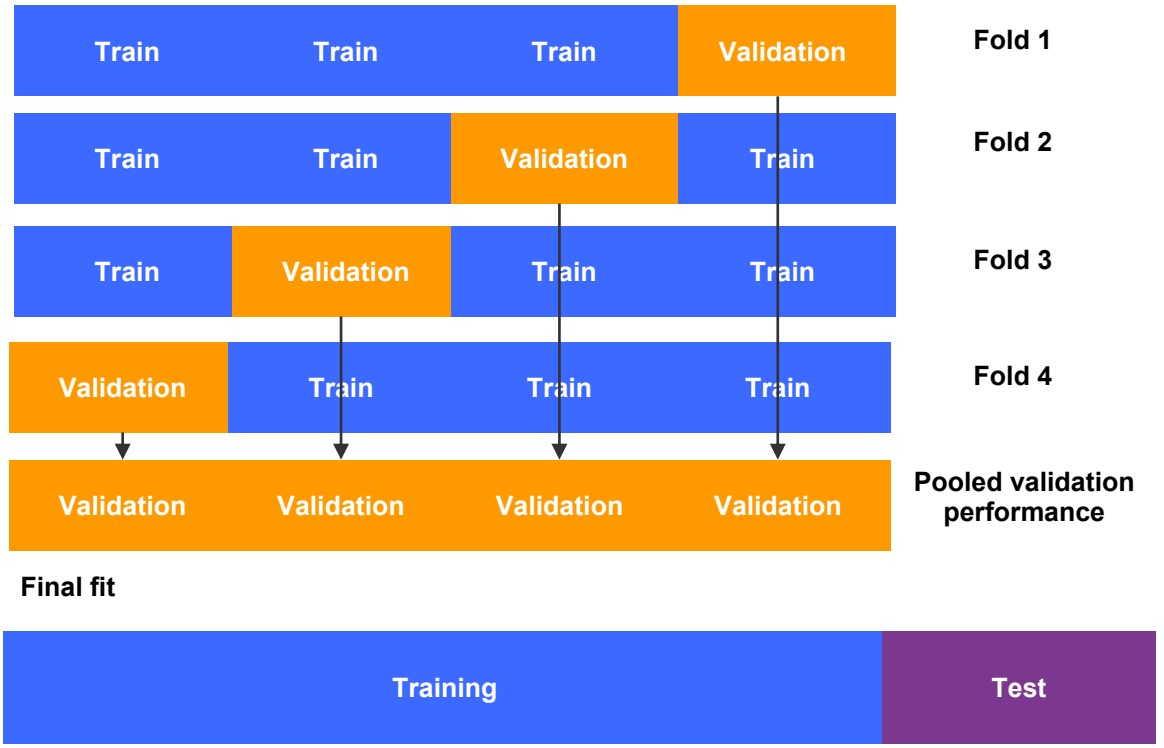
Another option that makes good use of the limited available observations is to pool the training and validation data and use a technique called cross-validation to tune the model's parameters. This technique uses the following steps:

1. the combined training and validation data is broken into L pieces;
2. a model is trained on each unique set of $L-1$ pieces and scored or validated on the omitted piece;
3. the model is 'tuned' by selecting parameters of the model that maximise its performance across all L 'folds' of the combined training/validation data;
4. the tuned model is then fit on the whole training/validation set; and
5. the model's generalisation or test error is estimated on the test dataset that the model has not seen to this point.

The approach requires more processing power to run as it requires L models to be trained, but with the availability of cloud resources this has become a more popular approach.

This cross-validation process is depicted in Figure 5.6.

Figure 5.6 – Illustration of cross-validation



Video 5.1 further helps to illustrate the process of cross-validation. Note that Video 5.1 ignores the testing step discussed in this chapter. The 'test' data referenced in Video 5.1 is 'validation' data using the terminology from this subject. In addition, Video 5.1 focusses on how to make a selection of the best model to use, but does not cover the next step of fitting the model to the training and validation data. In other words, Video 5.1 covers only the first three of the five steps outlined on the previous page,

Video 5.1 – Cross-validation

https://www.youtube.com/watch?v=fSytzGwwBVw&t=38s (6 mins)

Record your video notes here

Avoiding Leakage

Leakage involves using information from outside the training dataset to train the model or, in other words, using information to train a model that would not normally be known at the time of making a prediction. This usually results in a model that appears to perform better, when evaluated during the development process, than it will actually perform when deployed.

Some common sources of leakage include:

- using future data that would not be available at the time of prediction;
- using information from the validation or test partitions;
- including correlated observations across the training, validation, and test splits;
- humans intervening based on past assumptions or modelling; and
- using erroneous logic to create the data splits.

Using future data that would not be available at the time of prediction can arise as part of data cleaning and feature engineering. An example of using future data that would not be available at the time of prediction is as follows: Imagine you are building a model to predict customer retention three months after each customer joins and the model uses the total number of interactions a customer has had to date. By including customer interactions more than three months after a customer has joined as features in your model, the model is 'cheating' by looking into the future from the point of prediction. Other examples of using future data that is not available at the time of prediction are:

- feature engineering using aggregation across an entire data partition;
- feature engineering using aggregation across all data for an entity;
- using the current value of a data field instead of the historical value;
- using publicly available economic statistics that hadn't been published until weeks after the time period;
- rescaling for long term trends over time; and
- smoothing values across time.

Using information from the validation or test partitions can also arise as part of data cleaning and feature engineering. For example, missing value imputation, principal components analysis, text tokenisation, and scaling should be based only on the values from the training partition. Do not rescale validation partition values using the averages in the validation partition. Do not impute missing values using averages calculated from the validation partition. It is best practice to partition the data before commencing any feature engineering.

Correlated observations across training, validation, and test splits can arise, for example, if there are multiple records from the same customer and some of these are included in the training split and some in the test set. Similarly, the claims experience of a group of travel insurance policyholders that are all travelling in the same tour might be split across the training, validation, and test splits. To determine whether these correlated observations are problematic, you should consider whether such correlations might naturally occur between (past) data used to train a model and (future) unseen data that the model will make predictions on. Sometimes you must choose between partitioning by entity versus partitioning by time, and this is a choice of the lesser evil, specific to the context. In such cases, explain your choice, detailing the trade-offs between the two options.

Humans intervening based on past assumptions or modelling can also lead to leakage. To illustrate this, consider a fraud prediction model. The model might be used by the investigation team to prioritise areas predicted to be at high risk of fraud. Naturally, when training future versions of the model, there will be more identified past fraud cases in areas where the investigation team has been more actively investigating (i.e. in areas that previous models have flagged as having high risks of fraud). This may result in the model being good at identifying certain types of fraud and poor at identifying other types of fraud. This could be mitigated by, for example, having the investigation team sample past records from both high and low risk areas.

An example of **using erroneous logic to create the data splits is as follows**: If today's date is March 2024, using ’data up to the end of 2023’ for training data and ‘the last 12 months’ for test data will result in an overlap between the two ranges. This issue can be mitigated through code review and documenting of modelling assumptions.

5.2.6. Feature Importance

Feature importance measures how much each feature contributes to the predictions made by a model. This can help to:

- understand which features are most relevant to the response variable;
- detect leakage; and
- validate the model based on the commercial context and understanding of the underlying process (e.g. Does it make sense that these are the most important features in the model, or has something gone wrong with the modelling process?).

There are different methods available to calculate feature importance, depending on the type of model. Some methods include:

- coefficients;
- split-based feature importance; and
- permutation feature importance.

For linear models, such as linear regression or logistic regression, the **coefficients** of the features indicate how much the features affect the prediction. The larger the absolute value of the coefficient, the more important the feature is. However, scale also needs to be considered. For example, the coefficient for 'height in metres' will be 100 times smaller than the equivalent coefficient for 'height in centimetres'.

For tree-based models (see Section 5.3.2), **split-based feature importance** can be used. Under this method, the feature importance values are calculated based on how much each feature is used in the splitting rules used by the tree.

Permutation feature importance is a model-agnostic method, meaning that it can be applied to any model, regardless of the model's architecture or complexity. The idea is to randomly shuffle the values of each feature and measure how much the prediction accuracy (or other metric of importance) drops. The more the prediction power reduces, the more important the feature is.

SHAP feature importance is another model-agnostic method that can be applied to any model, regardless of the model's architecture or complexity. It measures the mean absolute value of the contribution of each feature to the predicted value, using Shapely Values. The higher the absolute value of the Shapely values, the more important the feature is.

An example of a feature importance plot is provided in Case Study 1.

5.2.7. Partial Dependence Plots

Partial dependence plots (PDPs) provide a visualisation of the average effect a change in one or two features has upon the predicted value, while keeping the other features fixed. The techniques used to generate PDPs can be applied to any type of data science model including neural networks or tree-based models.

PDPs are useful because they provide some interpretability and insight into how the model makes predictions. They can reveal the shape and direction of the relationship between the features and the output from the model. These plots can be validated by the data science actuary for reasonableness and are also useful for communications with business stakeholders about how the model makes predictions.

Examples of partial dependence plots are provided in Case Study 1.

5.2.8. Prediction Explanations

SHAP prediction explanations provide a visualisation of the contribution of each feature value in a data row towards the predicted value for that data row. Using Shapely Values from game theory, the contribution of each feature adds to the prediction. Note that some implementations of SHAP prediction explanations may be approximations, or explain the logit transformed value of the predicted value in a classification problem.

Since prediction explanations are based upon case study examples, they are a powerful tool for intuitively communicating how a model makes its predictions.

5.2.9. Model Drift

Model drift refers to the degradation of a model's performance over time due to changes in the data or the environment. Model drift can arise due to data drift or concept drift.

Data drift occurs when the statistical properties of the input data change over time, meaning that the data distribution changes, either for features in the data or the response variable. For example, a model trained to predict customer behaviour may not be able to account for changes in customer demographics or preferences over time.

Concept drift refers to when the relationship between the features and the response variable changes over time, meaning that the definition of the problem changes. For example, a model trained to detect spam emails may not be able to recognise new techniques spammers develop to circumvent the spam detection algorithm.

Model drift can be detected by measuring the accuracy, precision, recall, or other metrics of the model on new data and comparing them with the expected or baseline values. If the metrics drop significantly, it may indicate model drift. There are also specific statistical tests that can be used to identify drift over time. To address model drift, new or updated data can be used to retrain the model or adjust its parameters to account for the model drift observed.

Data drift can be detected by:

- Subjectively by plotting the
  - average value of a numeric feature over several consecutive time periods, and checking whether the later average values diverge from the previous historical ranges

  - frequency of specified values of a categorical feature over several consecutive time periods, and checking whether the later average values diverge from the previous historical ranges

- Splitting the data into two time periods and applying statistical tests:
  - Kolmogorov-Smirnov (KS) test: For continuous variables, it compares the distributions of period 1 vs. period 2
  - Chi-square test: For categorical variables, it checks whether the distribution of categorical values has changed from period 1 vs. period 2
  - Population Stability Index (PSI): Common in credit risk modelling, the PSI measures the shift in distribution between period 1 vs. period 2
- Splitting the data into two time periods and fitting a **discriminator model** to predict the time period each data row was sourced from:
  - Feature importance of the discriminator model will tell you which features have experienced data drift
  - Partial dependence of the discriminator model will show you which data values are more likely to come from one period versus the other period

Concept drift can be detected by:

- Plotting the average predictor/target value (vertical axis) versus input feature value (horizontal axis), with a separate line for each time period. Concept drift will manifest as lines that are different shapes or slopes.
- Splitting the data into two time periods and fitting two separate predictive models, one for each time period. Concept drift will manifest and significantly different features importances and/or partial dependences between the two fitted models.

Model drift can be difficult to mitigate, requiring domain knowledge, human judgement and creativity. Here are a few ideas for you to try:

- Add a binary input feature that flags when behaviours changed e.g. when a new regulatory regime commenced
- Rescale data values for long term trends e.g. use inflation adjusted values
- Frequently retrain your model with new data. The schedule (weekly, monthly) depends on the rate of data change in your domain.
- Incorporate time-window-based features in your model, which account for recent changes in data behaviour (e.g., rolling averages, moving time windows).

- Use techniques like regularization (L1/L2) or dropout (in neural networks) to make the model less sensitive to noise and minor variations in the data, thus more robust to gradual changes in distributions.
- Feature engineer for movements in data values, instead of their absolute levels.

5.3. Common Classification Models

This section provides a brief recap of the approaches to classification introduced in the Data Science Principles subject.

5.3.1. Logistic Regression

Generalised linear models were taught in the Actuarial Statistics and Data Science Principles subjects. Logistic regression is a type of generalised linear model that uses the logit function ($h(x) = log\ \frac{x}{1-x}$) as the link function. For observation i, the raw output of logistic regression, $\hat{y}_{i,1}$, is the probability of observation i belonging to class 1, $\hat{p}_{i,1}$. This is estimated as follows:

Equation 5.4

$$\hat{\boldsymbol{y}}_{\boldsymbol{i,1}} = \hat{\boldsymbol{p}}_{\boldsymbol{i,1}} = \frac{e^{\beta_0+\beta^T X_{i.}}}{1+e^{\beta_0+\beta^T X_{i.}}}$$

where β_0 and β^T represent the coefficients of a linear transformation of $X_{i.}$, β_0 is the intercept term and β^T is a row vector of weights, $\beta_1, \beta_2, \dots, \beta_j$, that are applied to each feature in $X_{i.}$.

The use of the logit link function in logistic regression ensures that the linear portion of the function, $\beta_0 + \beta^T X_{i.}$, is transformed into a number between 0 and 1. This makes it suitable as a probability estimate.

The fitting of a logistic regression model is usually done by finding the maximum likelihood estimators for the parameters $\beta_0, \beta_1, \beta_2, \dots, \beta_j$. This corresponds to the choice of the logistic loss function (see Section 5.2).

Logistic regression is one of the fundamental approaches to binary classification. It takes the machinery of linear models and applies it in a classification context. While logistic regression is rarely the most accurate classifier that can be used, it is popular because of its speed, robustness and the fact that in many cases it is 'good enough'.

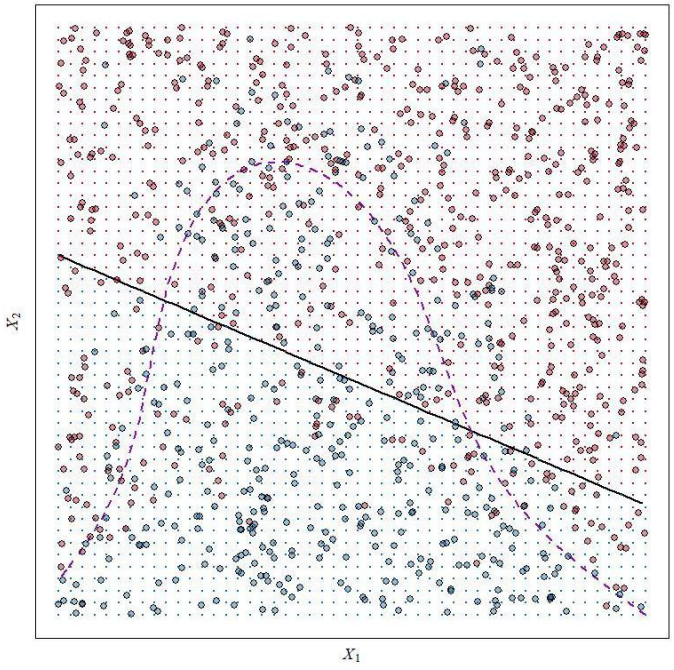
When certain statistical properties of logistic regression are satisfied (e.g. when there is no multicollinearity between the features), this method also has the powerful property of providing confidence intervals on predictions (i.e. measures of uncertainty), as opposed to many other classification algorithms that are largely non-parametric.

One natural limitation of logistic regression is it creates linear decision boundaries for the classifier. For example, if a threshold of $\hat{y}_{i,1} > 0.5$ is used to classify an observation into class 1, then that classification is made if, and only if, $\beta_0 + \beta^T X_{i.} > 0$.

This represents a linear hyperplane, which means that any nonlinearities in the true decision boundary are ignored.

Figure 5.7 shows the results of fitting a logistic regression classifier to the toy dataset introduced in Section 5.1.2.

Figure 5.7 – Toy dataset with a logistic regression fit—decision boundaries



In Figure 5.7, the dashed purple line represents the true decision boundary from which the data was generated. The black line is the decision boundary from the logistic regression model. This model classifies every observation above the modelled decision boundary as red and every observation below it as blue, as indicated by the grid points. This correctly classifies a large proportion of observations, but there are clear regions of the data that are poorly handled by the classifier.

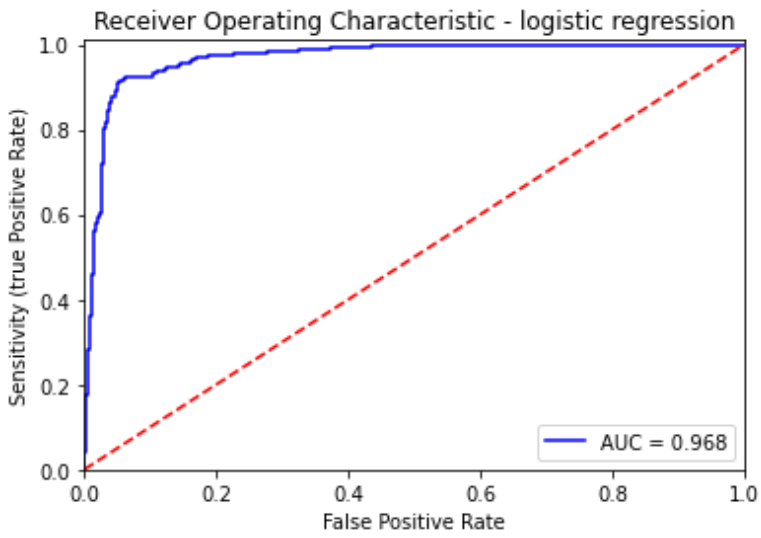
Logistic regression can be made to recognise nonlinearities in the true decision boundary by deriving new feature variables. For each observation, $X_{i.}$, features could be created using terms such as:

- polynomial terms (e.g. x_{ij}^2, x_{ij}^3);
- interaction terms (e.g. $x_{i1} \times x_{i2}$);
- spline terms, such as $x_{ij} I\{x_{ij} > t\}$; and
- other nonlinear transformations, such as $log\ log\ x_{ij}$.

This is often a manual process, with terms hand-picked to fit the data well. This contrasts to the approach used in other classifiers, such as decision trees or neural networks, that automatically detects and fits nonlinear effects by design.

As another example of the effectiveness of logistic regression in classification, consider the spam dataset introduced in Section 5.1.2. A simple logistic regression model with no variable transformation produces reasonable predictions, as shown in the resulting ROC curve in Figure 5.8.

Figure 5.8 – Spam dataset with a logistic regression fit—ROC curve



The resulting accuracy from the logistic regression model (92.3% using a 50% threshold for classification) is much higher than the simple classifier discussed in Section 5.2 (78.0%). Similarly, the ROC curve is much stronger, with an AUC of 0.968 compared to 0.784 under the simple classifier.

Extensions of Logistic Regression to Multiclass Classification

Logistic regression can be extended to model more than two classes. Rather than having a single linear predictor (LP), one is defined for each class, each with its own parameter set:

class g_1	$LP_1 = 0$
class g_2	$LP_2 = \beta_{02} + \beta_2^T X_{i.}$
class g_3	$LP_3 = \beta_{03} + \beta_3^T X_{i.}$
⋮	⋮
class g_K	$LP_K = \beta_{0K} + \beta_K^T X_{i.}$

Class 1 does not require parameters—the other class probabilities are set relative to this base class. The probability for class k is then estimated as:

Equation 5.5

$$\hat{\boldsymbol{p}}_{i,g_k} = \frac{e^{LP_k}}{\sum_{d=1}^{K} e^{LP_d}}$$

By design, these class probabilities will sum to one. As for logistic regression in a binary setting, the parameters are found by maximum likelihood using the multiclass version of logistic loss. The mathematics for this loss function is outside the scope of this subject. Instead, standard packages in Python and other programming languages can be used to minimise this loss function for multiclass logistic regression.

Exercise 5.9

Demonstrate that Equation 5.5 ensures the final probability estimates will sum to one.

Suppose that class 2 becomes the new reference class, so that $LP_2 = 0$ and all other linear predictors are redefined as $LP_k(base\_2) = LP_k(base\_1) - (\beta_{02} + \beta_2^T X_{i.})$.

Demonstrate that the same set of probability predictions can be recovered as when class 1 was the reference class.

5.3.2. Decision Trees

Decision trees are another common type of model that can be used for both classification and regression. A decision tree is essentially a set of logic rules based on the features of the training data. These rules dictate which class to assign each observation to and are set out in a flowchart-like structure, where:

- each internal node denotes a test on a feature in the data;
- each branch represents an outcome of a test; and
- each leaf or terminal node contains a class label.

Video 5.2 illustrates a simple decision tree.

Video 5.2 – Decision trees

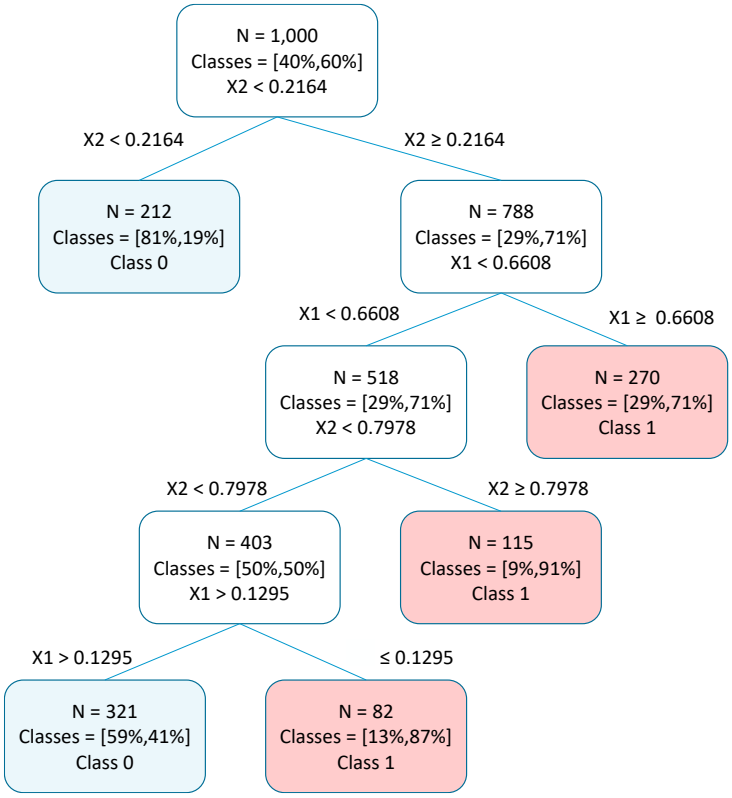
https://www.youtube.com/watch?v=eKD5gxPPeY0 (9 mins)

Record your video notes here

There are many variations of decision tree models. This chapter focuses on the popular classification and regression tree (CART) framework, which uses recursive binary partitions.

Consider the decision tree shown in Figure 5.9 that has been built on the toy example from Section 5.1.2.

Figure 5.9 – Decision tree for the toy example



In Figure 5.9, the starting node—at the top of the tree—contains all 1,000 observations, 40% of which are class 0 (blue) and 60% class 1 (red). This decision tree has determined that the 'best' split of this starting node is to divide it into one subset where $X_{.2} < 0.2164$, and another subset where $X_{.2} \geq 0.2164$.

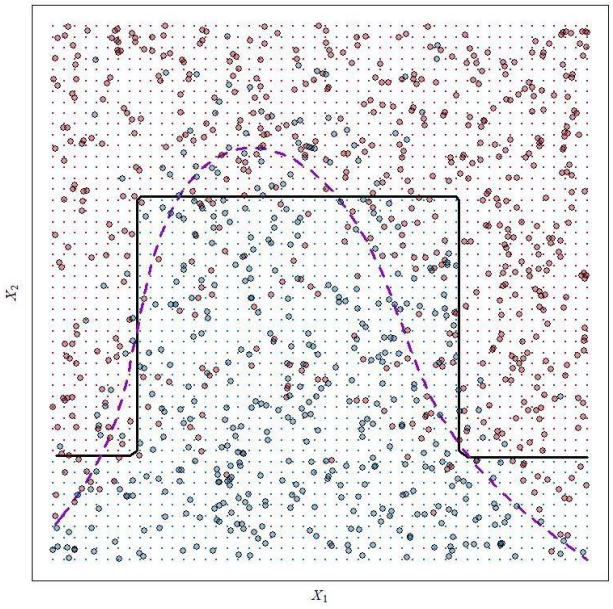
A convention in decision trees is to write the rule used to split each node within the node (e.g. '$X_{.2} < 0.2164$' in the starting node), with observations that satisfy the rule going to the branch to the left of the node.

In Figure 5.9, the left node in the second layer of the tree has 212 observations, 81% of which belong to class 0. This node has not been split further and is, therefore, called a terminal node, with the prediction for these observations being class 0. The right node in the second layer of the tree is further split based on the value of $X_{.1}$.

In total, there are four splits shown in this tree with five terminal nodes. The class to be assigned to each terminal node has been chosen as the class with the greatest number of observations in that node. The blue-shaded terminal nodes represent observations that should be classified as class 0 (blue), and the red-shaded terminal nodes represent observations that should be classified as class 1 (red). A new observation can then be 'dropped' down the tree (i.e. the decision rules in the tree are applied to the features of the new observation) until it reaches a terminal node. This terminal node determines the class to which the new observation should be assigned.

The decision tree in Figure 5.9 produces a decision boundary that is the combination of rectangular regions, as shown by the black line in Figure 5.10. The shape of this line roughly echoes the true curved decision boundary (shown as a dashed purple line).

Figure 5.10 – Output of fitting a decision tree with five terminal nodes to the toy dataset



A more complex tree with additional branches and nodes may produce a boundary that is even closer to the true boundary. However, more complex decision trees require a lot of training data. Without enough training data, the number of observations in individual nodes will quickly become too small to allow any further meaningful subdivision.

The main design decisions for building a tree are:

- choosing the rule that splits a node into two groups; and
- deciding when to stop splitting a node.

Splitting Rules

Splitting rules can be made via optimising an impurity measure, Q, for a tree, T, with the aim being to maximise the proportion of observations in the node that belong to the correct class (i.e. maximise the purity of the node). The impurity measure is analogous to the role of a loss function and is defined for each terminal node of a tree.

For a decision tree T, let:

- R_m denote terminal node m;
- N_m denote the number of observations in R_m;
- $k(m)$ denote the class assignment of R_m; and
- $\hat{r}_{m,k}$ denote the proportion of observations in R_m that belong to class k.

Common choices for the node-level impurity measure, Q_m , are shown in Table 5.6.

Table 5.6 – Common node-level impurity measures

Impurity measure	Formula
Misclassification error	$\frac{1}{N_m}\sum_{i\in R_m} I\left(g_i \neq k(m)\right)$
Gini index	$\sum_k \hat{r}_{m,k}(1-\hat{r}_{m,k})$
Cross-entropy or deviance	$-\sum_k \hat{r}_{m,k}\log_{10}\hat{r}_{m,k}$

Note that in calculating the cross-entropy or deviance impurity measure, any base can be used for the log function, provided you use the same base when comparing outcomes for different models or different versions of a single model.[6] The overall impurity of tree T, $Q(T)$, can be calculated as the sum of weighted node-level impurities:

Equation 5.6

$$\boldsymbol{Q(T) = \frac{\sum_m N_m Q_m(T)}{\sum_m N_m}}$$

[6] The answer to the following Stack Exchange post provides a mathematical explanation of this: https://stats.stackexchange.com/questions/295174/difference-in-log-base-for-cross-entropy-calcuation/295191

The misclassification error is coarse, in that it relies on the overall classification of a node. In contrast, Gini and cross-entropy impurity measures recognise the proportions of observations within classes. This means that even if two potential split rules produce similar misclassification errors, one potential split rule may be better than the other, based on the Gini or cross-entropy measures, as illustrated in Exercise 5.10 below.

Video 5.3 helps to illustrate the impurity calculations set out above.

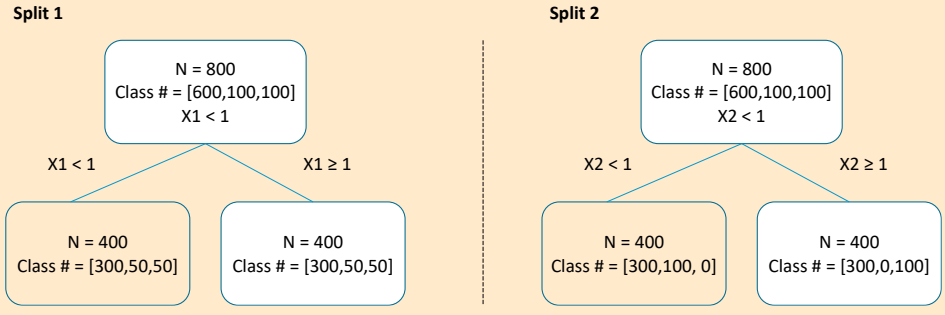
Video 5.3 – Impurity measures

https://youtu.be/8zyLiGtMZyM (12 mins)

Record your video notes here

Exercise 5.10

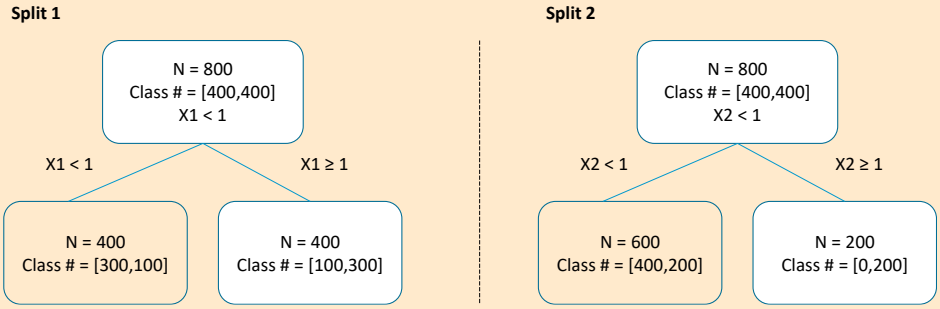
Consider the following two possible splitting rules for a three-class classification problem.



Determine[1] which splits are preferred under misclassification, Gini and cross-entropy impurity measures.

Exercise 5.10

Now consider the following two possible splitting rules for a binary classification.



Determine[1] which splits are preferred under each impurity measure for this problem.

[1] For each decision tree, you should assume that observations in a terminal node are assigned to the class with the most observations. This is known as the majority vote rule.

CART models can also be built for regression problems. In this case, the node impurity is a loss function such as MSE (see Section 5.2), and the prediction within a terminal node is the mean value of observations in that node.

A CART model is grown in a greedy fashion. It is greedy in the sense that, for a given tree, every current terminal node is considered for splitting in turn, and all possible split rules at each terminal node are considered. The next split to be performed is the one that reduces the overall tree impurity by the largest amount.

Choosing the optimal split at each stage does not guarantee that the overall tree is optimal. This is because the splitting rule applied is not forward-looking—the aim is to reduce the impurity of the tree at its current stage, not the overall impurity of the tree after all splitting rules have been applied. However, this approach makes the fitting process more tractable for computation.

The final splitting rules that an algorithm uses help to determine which features are important in predicting the response variable. Measures such as the Mean Decrease in Impurity (MDI) and Mean Decrease in Accuracy (MDA),[7] can be used to assess the extent to which individual features help to decrease the impurity or improve the accuracy of the tree's predictions respectively. This is akin to using p-values to determine feature importance in linear regression, but is even more powerful in practice as it does not require the dataset to satisfy the statistical assumptions behind linear regression (linearity, homoscedasticity, independent, and normality).

Selecting the Optimal Tree Size

There are various ways to find the optimal tree size (i.e. the number of splits that should be made). One common method is known as cost-complexity pruning. Under this approach, a 'full' tree is grown—perhaps with a constraint on the minimum number of observations in each node—and then pairs of terminal nodes are pruned back to minimise the resulting increase in impurity. The terminal nodes that are pruned back in each stage of this process are referred to as the 'weakest link' of the tree.

The goal of the cost-complexity pruning approach is to minimise the following cost-complexity criterion:

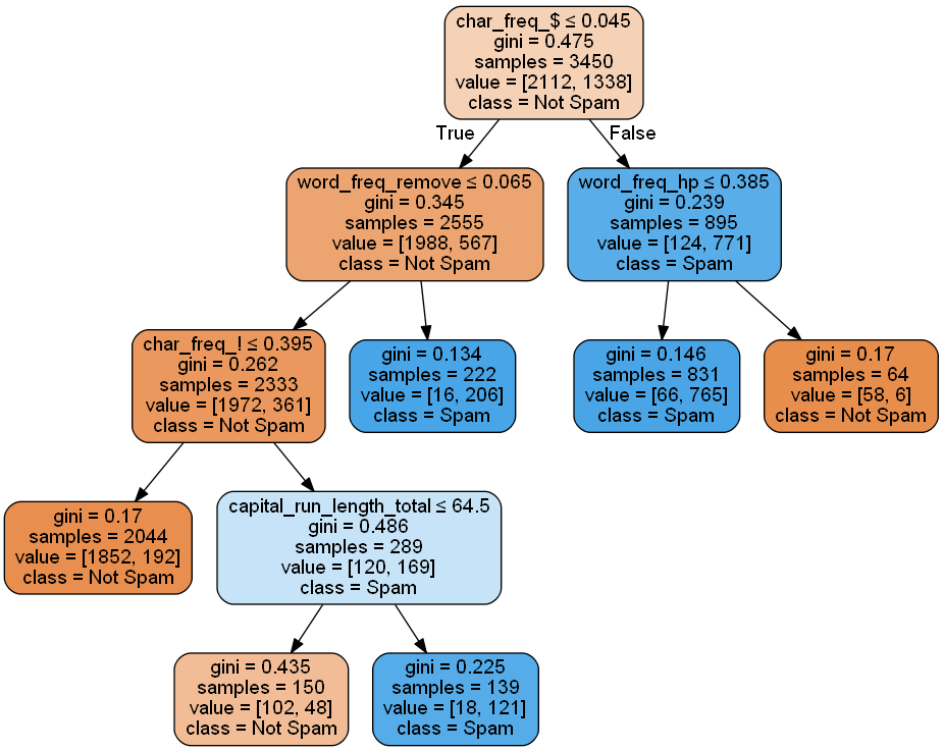
Equation 5.7 $$\boldsymbol{C_\alpha(T) = \sum_m N_m Q_m(T) + \alpha|T|}$$

where $|T|$ denotes the number of terminal nodes in the tree and α can take a value from 0 (the full tree) to ∞ (an empty tree). To avoid overfitting, the optimal value for α should be determined using a validation dataset or through cross-validation (see Section 5.2).

The tree in Figure 5.11 has been fitted to the spam dataset from Section 5.1.2 using $\alpha = 0.01$. This value of α gives a tree with six terminal nodes.

[7] Note that the calculations required for these methods are out of scope for this subject.

Figure 5.11 – Decision tree applied to the spam dataset



Better impurity performance is obtained with a larger tree. Using $\alpha = 0.001$ gives a tree with 30 terminal nodes a validation data accuracy of 0.91 and an ROC AUC of 0.957. This AUC is lower (inferior) to that obtained using the logistic regression classifier shown in Section 5.3.1 (0.968).

Exercise 5.11

Run the code provided in 'DSA_C05_Ex11.ipynb' to fit a decision tree on the spam dataset.

Discuss whether accuracy rates, AUC and tree complexity differ if the 'criterion' is changed from 'Gini' to 'entropy'.

Describe how accuracy rates, AUC and tree complexity vary with different rates of α.

Decision trees have the following advantages over logistic regression and other classifiers such as neural networks:

- they handle categorical, ordinal and continuous variables out of the box (i.e. do not need to transform a categorical variable with m categories into $m-1$ dummy variables);
- they handle missing observations well;
- they are interpretable, as it is easy to see the variables driving decisions and the allocation process used for each decision;
- their predictions are robust;
- they are insensitive to feature transformation; and
- they are nonparametric, with no assumed parametric form for the predictions.

In a decision tree, the m categories of a categorical variable can be combined into two groups. For example, a categorical variable that represents clothing size might have five categories: extra small, small, medium, large, and extra large. In a decision tree, these categories might be combined into two groups: one that includes extra small, small, and medium, and the other that includes large and extra large. A decision rule in the tree can then be based on which of the two groups an observation is in. Ordinal variables can be treated similarly to continuous variables, with a threshold level used to allocate observations to different groups.

Decision tree predictions are robust since all predicted probabilities are averages of known nodes. Outliers are, therefore, often given plausible estimates. This compares favourably to regression models, where outliers can sometimes be given outlandish estimates.

In a decision tree, a monotonic transformation of a continuous feature, such as when used in the common process of normalising features, will not change the results, since a tree looks at all possible thresholds and the observations above and below those thresholds. Other classification models, such as neural networks, require more attention to the impact of feature transformation on the model and the relative scales of the features generally.

Non-parametric models are very powerful for data sets where you do not know what the underlying distribution is, which is often the case in practice.

The main disadvantage of decision trees compared to other classifiers relates to the accuracy of their predictions. Decision trees typically produce good, but not necessarily great, predictions for the following reasons:

- having constant estimates within each rectangle is unlikely to closely mimic reality;
- effects that are common to all observations (e.g. a consistently increasing age effect) are reproduced inefficiently and imperfectly in decision trees (see below);
- model complexity grows relatively slowly with the number of observations—the depth of a tree grows in proportion to $log\ log\ n$;
- tree stability can be an issue, with slight variations in splits often leading to markedly different predictions; and
- categorical variables with many levels can quickly result in the overfitting of a tree.

To help understand how a decision tree model can be less efficient and less precise in modelling effects that are common to all observations, imagine that you are building a model to predict an individual's risk of dying from influenza based on their age and various health factors. Assume that an individual's risk of dying from influenza increases monotonically as age increases. A logistic regression model can handle this data feature well by including age as a feature, $X_{.j}$, and applying the linear transformation $\beta_j X_{.j}$ such that as $X_{.j}$ increases, $\beta_j X_{.j}$ increases and $e^{\beta_j X_{.j}}$ increases.

On the other hand, a decision tree would need to use many node splits to capture this monotonic relationship between age and mortality risk, as each split will only capture ages above and below a threshold (e.g. age < 40 and age $\geq$ 40). In this case, the decision tree will likely be less efficient (and possibly less effective) in modelling the relationship between age and mortality risk compared to a logistic regression model. This might result in a stepped decision boundary like the one shown in Figure 5.10, as opposed to a smoother curve that better approximates the true decision boundary.

In situations where improved accuracy is required, an ensemble of decision tree models will often produce very good results, as discussed in Section 5.3.3.

Exercise 5.12

Describe some classification problems that are well-suited or poorly suited to single decision tree models, given the advantages and disadvantages discussed above.

5.3.3. Ensembles—Bagging and Boosting

A powerful class of techniques to improve predictive accuracy is ensemble models, where multiple predictions from smaller sub-models are combined to produce an overall estimate.

An ensemble contains many smaller sub-models that are usually called ‘base learners’. Ensembles are appealing because they can boost ‘weak learners’, such as small individual decision trees that might only provide a slight improvement over random guessing, to ‘strong learners’ that can make very accurate predictions. For this reason, ‘base learners’ are also referred to as ‘weak learners’.

A disadvantage of ensembling is that interpretability and explainability of predictions become more challenging, as each prediction is now a complex combination of potentially many sub-models. However, techniques such as feature importance**,** local interpretable model-agnostic explanations (LIME) and Shapley additive explanations (SHAP) can be used to interpret the predictions of a complex model.[8]

Ensembling, which is the process of building an ensemble model, can be applied to any base learner model such as a decision tree or a neural network. Different types of base learner models might also be used within an ensemble.

This section focusses on two types of ensembling:

- bagging; and
- boosting.

Bagging and boosting are very popular techniques for boosting the performance of a single decision tree. Therefore, the discussion of these methods focuses on their use in the context of decision trees. However, the comments made below are generally relevant to any type of base learner model.

[8] While outside the scope of this subject, students wishing to learn more about LIME and SHAP can refer to the following articles:
Hulstaert, L. (2018). Understanding model predictions with LIME. *Towards Data Science*. https://towardsdatascience.com/understanding-model-predictions-with-lime-a582fdff3a3b;
Ma, E. (2018). Interpreting your deep learning model by SHAP. *Towards Data Science*. https://towardsdatascience.com/interpreting-your-deep-learning-model-by-shap-e69be2b47893.

Bagging

Bagging is an ensemble model approach that relies on bootstrapping the training data. Bootstrapping is a type of resampling whereby large numbers of smaller samples of the same size are repeatedly drawn, with replacement from a single original sample. The bagging approach takes bootstrap samples of the training data and fits separate models to each sample.

A popular variant of bagging that uses decision tree models is Random Forests. In addition to taking bootstrap samples of the training data to grow multiple trees, Random Forests applies the following adjustments to the standard decision tree approach:

- for each tree that is estimated from a bootstrapped sample of data, a random subset of features is used at each split point; and
- large trees are grown rather than an optimal tree size being selected for each model.

These adjustments produce additional variability in the trees. This is because the random subsetting of features reduces the correlation of predictions from each model that is averaged. For example, some trees might exclude most of the highly predictive features such that a feature with lower predictive power is used as the first split in the tree. This produces more diversified models. When averaged over a large number of models, this technique often leads to more accurate predictions compared to standard tree classifiers.

Boosting

Another important class of ensemble models is boosted models. A base learner model, such as a single decision tree, is trained on the data and the model's residuals are calculated for each observation as the difference between the observed and predicted values ($y_i - \hat{y}_{i,g_i}$). Another model is then trained on the data, with increased weight given to observations that were poorly estimated (i.e. had high residuals) under the base learner model. This new model is used to update the running total predictions, and new residuals are calculated. This process is continued until some stopping rule is reached, such as when performance starts to deteriorate on a validation dataset. Since each sub-model focuses on the observations that were poorly estimated in the previous sub-model, this approach can produce highly accurate predictions that are materially better than those under any individual base learner model.

Generally, boosting does not involve bootstrap sampling (bagging) but rather updating a model fit over many iterations using the original data. Therefore, unlike bagging, where models are fit independently to each dataset, boosting involves fitting a model at each iteration that depends on the models fitted at each previous iteration. Figure 5.12 shows the resulting decision boundary for a boosted model that combines four simple decision trees, each with less than four split rules applied to any single observation (i.e. each sub-model is simpler than the one shown in Figure 5.10).

Figure 5.12 – Decision boundary for simple tree boosted model, toy example

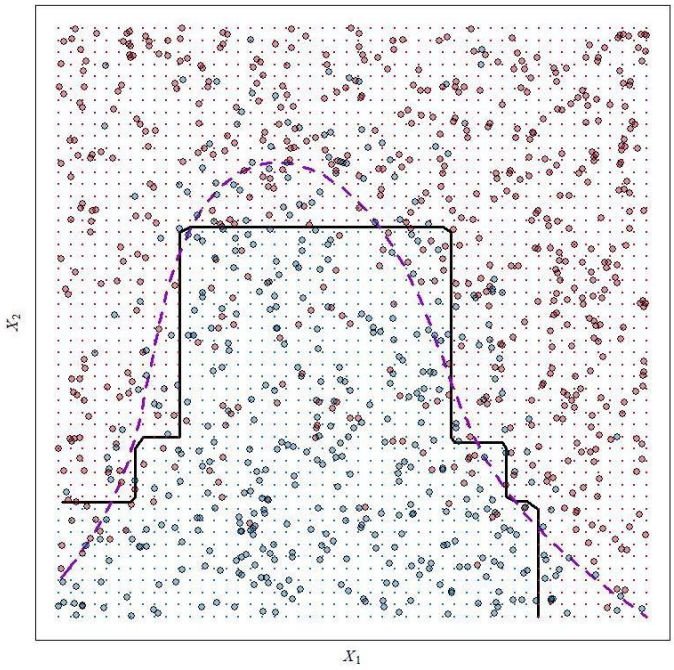


Figure 5.12 shows that rather than the decision boundary being made up of large rectangles, a 'staircase' pattern is produced that hugs the optimal decision boundary (shown as a dashed purple line) more tightly.

In practice, it is common to have hundreds of decision trees (or other types of models) contributing to a boosted model. Variability in the resulting trees can be achieved by using a sub-sample of observations and/or features at each step. This can further improve the accuracy of the boosted model. The accuracy of the model can also be improved by setting the 'learn' rate to a small number such as 0.1 or 0.01, so that only a fraction of each incremental tree is applied to the running estimate of the prediction.

Example

Kaggle is a website devoted to data science competitions. On Kaggle, boosted decision tree models, such as the XGBoost implementation, have emerged as one of the most popular general-purpose modelling tools, appearing in many solutions of competition winners.

Another popular technique is to combine multiple different models, which is usually referred to as 'stacking'. This approach takes test predictions from each model and uses these as inputs into a second-stage prediction model. This extends the idea of bagging and boosting to more general ensemble models of different predictions.

As an example, the Allstate Claims severity competition[9] asked participants to predict insurance claim loss from about 260 different variables. The winning solution fit 81 different first-level models, many of them neural networks and boosted decision tree models. These were used as inputs into three second-layer models (one XGBoost model and two neural networks). A combination of these three models was used as the final competition submission.

Exercise 5.13

Find another completed competition on the Kaggle website that has a winner's interview or summary.

Identify whether the final solution stacked together various models.

Identify whether decision tree boosting was used.

Identify the programming languages that were used.

[9] https://www.kaggle.com/c/allstate-claims-severity/overview

5.4. Other Key Concepts

This section provides a reminder of two other data science concepts that are important to understand before studying neural networks. These are:

- the bias-variance trade-off; and
- gradient descent.

Many students will have learned these concepts in previous subjects, such as Data Science Principles. Students who are unfamiliar with these ideas may want to seek further information:

- for bias-variance trade-off, see Section 2.2.2 of *An Introduction to Statistical Learning*;[10] and
- for gradient descent, see Weeks 1 and 2 of Stanford's *Machine Learning* online course.[11]

5.4.1. The Bias-Variance Trade-Off

Bias error (or 'underfitting') occurs when a model is assumed to be too simple and may miss relevant patterns between the features and response variable. For example, linear regression often has high bias relative to more complex models that better capture the relevant patterns in the data. Conversely, variance error (or 'overfitting') arises when a model is too complex and begins to fit volatility in the training data that does not allow the model to generalise well to unseen test data. Neural networks, which are introduced in Section 5.5, often face high variance unless regularisation is used to penalise a model for being too complex.

Error on a test dataset will arise from both bias and variance error. This can be demonstrated by considering the below formula for the mean square error of a prediction:

Equation 5.8

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_{i,1})^2 \text{ (Table 5.2)}$$

$$= E[(Y - \hat{Y})^2]$$

$$= (E[Y] - E[\hat{Y}])^2 + Var(\hat{Y}) + Var(Y)$$

$$= \text{bias}^2 + \text{estimation variance} + \text{process variance}$$

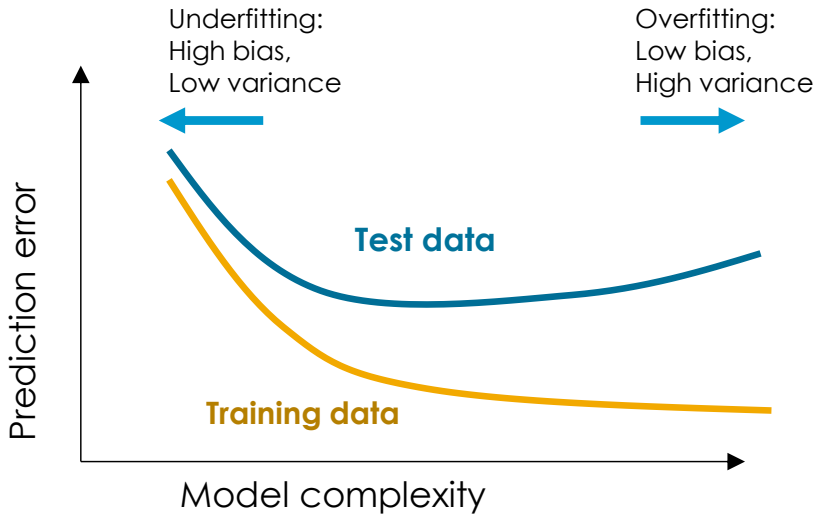
Equation 5.8 shows that it is desirable to minimise both bias and estimation variance to obtain the lowest MSE and, therefore, most accurate predictions. Note that the process variance term at the end of Equation 5.8 is fixed as it is a function of the observed responses.

[10] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2017). *An Introduction to Statistical Learning*. Springer.
[11] Stanford University. *Machine Learning*.[Online course]. Coursera. https://www.coursera.org/learn/machine-learning

The bias-variance trade-off refers to the idea that a model with low bias (i.e. one that is flexible enough to recover something that is close to the true shape of $P(y_i = g_k|X_{i.})$ or $f(X_i)$) will suffer from high variance. For example, you might introduce more parameters to reduce a model's bias but that will increase the variance error. Therefore, a model must strike a balance between bias and variance to get the best performance on test data. The bias-variance trade-off will underpin much of the discussion around tuning models in this subject.

The bias-variance trade-off is shown schematically in Figure 5.13, which considers a spectrum of models of increasing complexity.

Figure 5.13 – The bias-variance trade-off—schematic

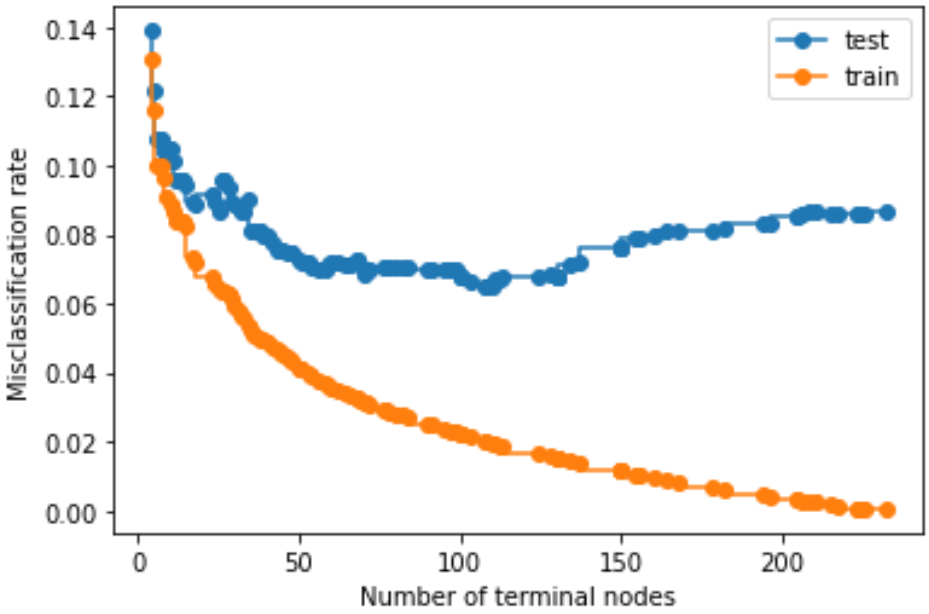


At every point along the training data curve in Figure 5.13, the performance on the training data improves. This is because as the model complexity increases, the model provides a better fit to the data on which it is being trained. However, for test data, performance improves initially as bias reduces, but then starts to increase as bias reductions are outweighed by variance increases.

The concept of models having different levels of complexity was explored in Section 5.3.2. A decision tree model can be made more complex by adding additional nodes, but a tree that is too large can experience greater test error than a smaller tree.

Figure 5.14 shows this trade-off for the spam dataset by plotting misclassification rates against the number of terminal nodes used in the decision tree.

Figure 5.14 – The bias-variance trade-off—decision tree applied to the spam dataset



From Figure 5.14, the optimal number of nodes appears to be in the 75–125 range for this dataset. Trees with more than 125 nodes experience worsening performance on the test data misclassification rate.

Exercise 5.14

Explain how the bias-variance trade-off manifests when fitting a logistic or multinomial regression, or a boosted decision tree model.

Neural network models, which are introduced in the next section, are a classic model class for suffering from overfitting or underfitting, depending on how long the algorithm is allowed to iterate over the training data. Figure 5.15 shows the outcome of fitting a relatively simple (single hidden layer) neural network on the previously introduced toy dataset.

Figure 5.15 – An appropriately tuned neural network, toy example

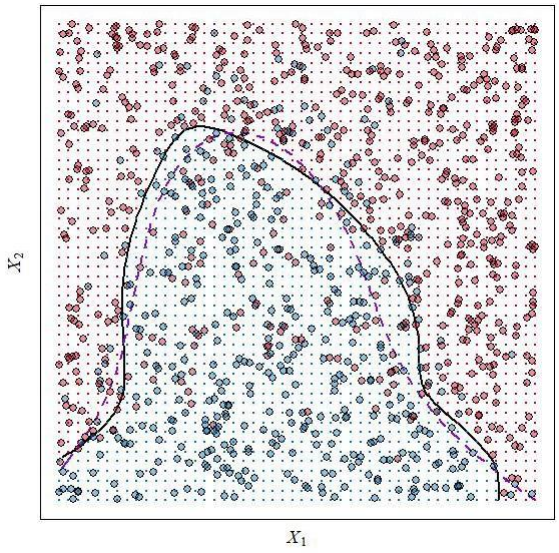
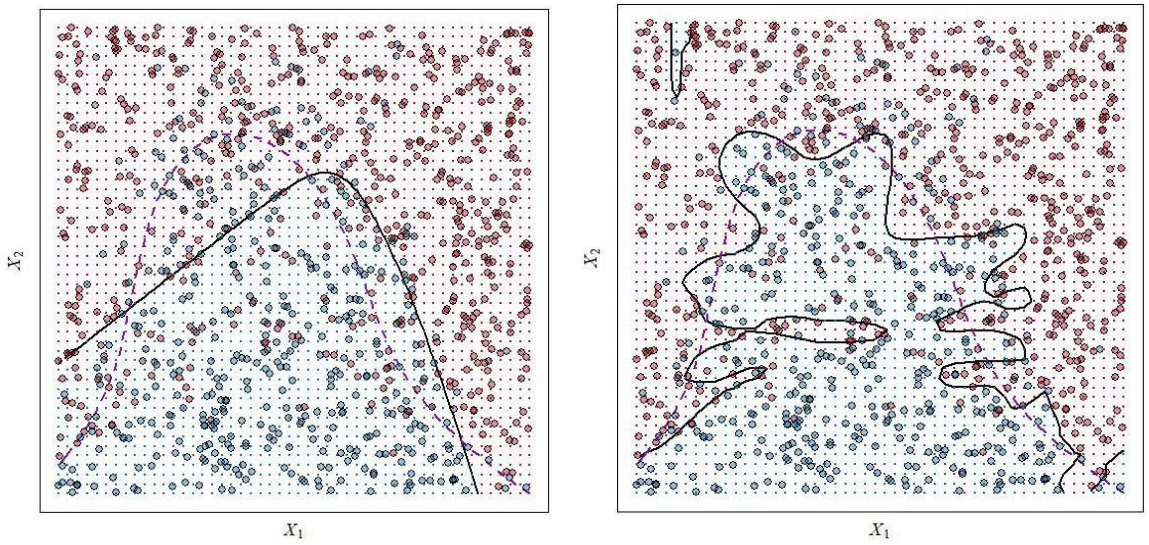


Figure 5.15 shows that when tuned properly, this simple neural network can fit the data very effectively. However, it is possible to underfit a neural network model by not giving it enough flexibility in design or not allowing it to run for sufficient time. It is also easy to overfit a neural network by allowing too much flexibility in the model or allowing the model to run for too long. Figure 5.16 shows the impact of underfitting and overfitting a neural network model to the toy dataset.

Figure 5.16 – An underfit (left panel) and overfit (right panel) neural network



In Figure 5.16, the underfit shape (left panel) misses much of the subtlety of the nonlinear curve, although this model might still be preferred over other attempts, such as the logistic regression. The overfit model (right panel) has a very wobbly decision boundary that follows spurious patterns in the training data. When fit to test data sets, this wobbly boundary will likely lead to misclassifications around the boundary.

Appropriate use of validation data is a straightforward way to manage the risks of underfitting and overfitting. Other techniques, such as parameter regularisation, which is taught in Data Science Principles, address this problem.

5.4.2. Gradient Descent

A few models, such as simple linear regression, have closed-form solutions for finding parameter values that minimise a loss function. Most other models, such as those discussed in this chapter, do not have closed-form solutions. Instead, other techniques need to be employed for finding good parameter values that minimise the model's chosen loss function. Gradient descent is one such technique that is commonly used in a wide variety of machine learning algorithms.

Suppose the loss function to be minimised is $J(\theta_1, \ldots \theta_p)$, which is made up of p parameters. The aim is to find parameters $(\theta_1, \ldots \theta_p)$ that minimise J.

For example, J could correspond to the logistic loss function, and 5.8

the θs are the parameters of a logistic regression. Suppose further that the partial derivatives of J with respect to each parameter, θ_j, can be calculated easily. These partial derivatives are expressed as $\frac{\partial}{\partial \theta_j} J(\theta_1, \ldots, \theta_p)$.

Under the gradient descent algorithm, J is minimised by **simultaneously** updating each parameter, θ_j, in the model using the partial derivatives of J as follows:

Equation 5.9

$$\theta_j \leftarrow \boldsymbol{\theta_j} - \boldsymbol{\alpha} \frac{\partial}{\partial \theta_j} \boldsymbol{J}(\boldsymbol{\theta_1}, \ldots, \boldsymbol{\theta_p})$$

By updating the parameters using the partial derivatives of J, a step is taken in the parameter space along the (local) path of greatest descent. It can be helpful to think of this in reference to climbing down a hill. You start at a random spot on the hill, and the partial derivatives tell you the direction on the hill that has the steepest downwards slope. A step is taken in this direction. From your new position, you then recalculate the direction with the steepest downward slope before taking another step in that new direction. This continues until you find a low point on the hill. This

may not be the bottom of the hill (the 'global minimum') but might instead represent a trough on the hill that is flat (a 'local minima').

For a function J that has multiple local minima, there is no guarantee that implementation of the gradient descent algorithm will find the global minimum. One consequence of this is that the starting values of parameters or different choices for the learning rate α (see below) can affect the final solution, as they may lead to a different local minimum. In such cases, multiple runs of the gradient descent algorithm may be performed, each with different random starting values of parameters or different values of α, to see which of these runs produces the smallest value of J.

The term α in Equation 5.9 is the learning rate. It denotes the size of the step to be taken. A small value of α will result in small steps being taken, which reduces the chance of overshooting the minimum value of the loss function. This is a particular risk if the partial derivatives are changing rapidly. However, a small value of α will require more steps to be taken before the algorithm converges on a solution, so the algorithm takes longer to run.

Video 5.4 provides a summary of the gradient descent algorithm outlined above and includes a helpful visualisation of the application of this algorithm.

Video 5.4 – Gradient descent

https://www.youtube.com/watch?v=yFPLyDwVifc (11 mins)

Record your video notes here

‘Batch’ gradient descent refers to gradient descent where the partial derivatives are defined using the whole set of training observations. This is sometimes also referred to simply as ‘gradient descent’ or ‘vanilla gradient descent’. ‘Stochastic’ gradient descent refers to a version of gradient descent where small portions of the training data are used at each step. This includes using:

- one observation at each step to update the model's weights (strictly speaking this is the 'pure' definition of stochastic gradient descent); or
- a small set of observations at each step (known as 'mini-batch gradient descent', but in practice, this is often also included under the term 'stochastic gradient descent').

Stochastic gradient descent (whether using individual observations at each step or mini-batches) requires less computation than batch gradient descent and can lead to significantly more efficient fit times in cases with large amounts of data.

5.5. Neural Networks

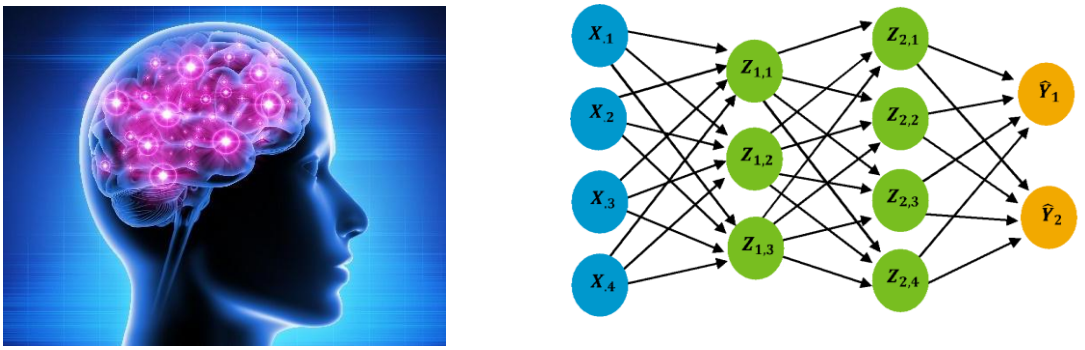
5.5.1. Introduction to Neural Networks

Section 5.3 discussed three common classification techniques: logistic regression, decision tree models and ensembling. These techniques can all be used to solve classification problems where the relationship between the input variables (features) and output variable (response) is nonlinear.

This section discusses another technique, neural networks, that can also be used to model complex nonlinear relationships. While neural networks can solve both classification and regression problems, the focus in this chapter is on neural networks in a classification setting. The application of neural networks in a regression setting is briefly discussed in Section 5.5.6.

The name 'neural network' derives from the fact that the different 'nodes' or 'neurons' of the model act as we would expect brain neurons to act. That is, the neurons fire or do not fire based on the input of prior input neurons. Figure 5.17 illustrates this connection between different neurons in the human brain.

Figure 5.17 – Natural and artificial neural networks



Natural neural network

Artificial neural network

The left-hand side of Figure 5.17 depicts a neural network in the human brain. These 'natural neural networks' comprise billions of neurons that are connected to each other via axons. Neurons receive or send messages to each other by transmitting electrical or chemical impulses along their axons. This complex neural network process allows us, for example, to see a step on the path in front of us, send this information to our brain, process this information within the neural network and then send a message to the muscles in our feet and legs telling them to move our body up the step.

The right-hand side of Figure 5.17 depicts an artificial neural network that attempts to mimic some of this structure using a collection of neurons and connections between them. Section 5.5.2 discusses this artificial neural network architecture in more detail.

A Brief History of Neural Networks

Neural networks, which are sometimes referred to as 'neural nets', were one of the earliest of the highly successful machine learning modelling approaches. This is partly due to their effectiveness in modelling nonlinear behaviours and complex interactions.

They are also among the earliest and most famous examples of 'black box' models, where interpreting what is happening inside the model is quite difficult.

Neural networks had an initial burst of interest in the computer science community in the 1980s and 1990s. At this time, neural network model complexity was constrained by a lack of computing power and lack of large volumes of quality training data. Interest in neural networks subsequently faded.

However, neural networks have become extremely popular more recently under the moniker 'deep learning'. Here, the term 'deep' refers to large, more complex networks that are supported by significantly larger datasets. This rise in popularity is partly due to the following breakthroughs that occurred in 2012:

- a University of Toronto research team used deep learning models to halve the error rate on ImageNet, an **image classification** project that had previously seen only modest gains in prediction performance over time;
- a broader group of researchers showed a deep learning model beating existing technologies in **speech recognition;**[12] and
- a Google research team trained a very large deep learning model on a large set of YouTube **videos**. Among other findings, it was demonstrated that it correctly recognised a large number of cat videos.

Such simultaneous advances in the use of neural networks to solve image, speech and video recognition problems have led to a significant leap forward in the use of neural networks in research, as well as consumer technologies such as digital voice assistants and improved mobile photography.

[12] Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A.R., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B., Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82-97.

The area of neural networks is continuing to evolve rapidly, with new software and hardware tools continuing to improve the performance of models.

There are many types and customised versions of neural networks geared towards specific applications. The focus in this chapter will be on a 'feed-forward' neural network that is fitted via a 'backpropagation' algorithm. These concepts will be explained in Section 5.5.3 and Section 5.5.4. Section 5.7.1 will then discuss some extensions and alternative versions of neural networks.

Video 5.5 touches on the concepts outlined in this introductory section and those covered in more detail in the rest of this chapter.

Video 5.5 – Introduction to neural networks

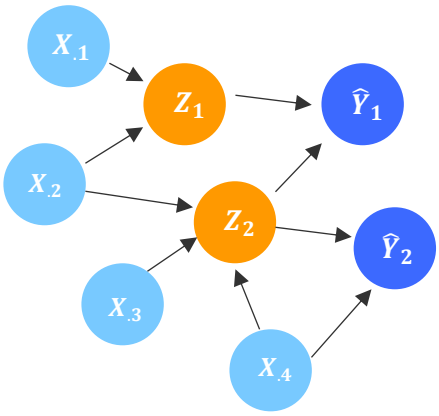
https://www.youtube.com/watch?v=bfmFfD2RIcg (6 mins)

Record your video notes here

5.5.2. Neural Network Model Representation

A neural network can be visualised as a network of interconnected neurons, as shown in Figure 5.18.

Figure 5.18 – A neural network



The three main types of neurons in a neural network are:

- original predictor neurons (the blue Xs);
- derived predictor neurons (the orange Zs); and
- response neurons (the purple $\hat{Y}$s).

The following occurs within a neural network:

- data is passed to the neural network via the original predictor neurons;
- this data is processed and transformed within the neural network through the derived predictor neurons; then
- the neural network produces outputs or predictions via the response neurons.

Other classification models often require humans to spend significant amounts of time on feature engineering, which involves preparing features for input into a model. In contrast, neural networks effectively automate the feature engineering process, without explicit instruction from a human, via the derived predictor neurons. This concept will make more sense as you work through the forward and backward propagation steps in Sections 5.5.4 and 5.5.5.

The following rules apply when constructing a neural network:

- each neuron represents a one-dimensional numeric variable (a 'scalar');
- the arrows pointing towards each neuron represent the other neurons that are used to derive it;
- original predictor neurons should not have arrows pointing towards them, as they are the provided input for the model and have fixed values;
- every other neuron should have at least one arrow pointing towards it (otherwise, it is a redundant component of the model);
- response neurons correspond to an actual response that has been observed;
- actual responses are used to train the network to reduce model error by making response neurons as close as possible to their corresponding actual responses.

Feed-Forward Neural Networks

One basic type of neural network, which is the focus of this chapter, is the feed-forward network. This has the following structure imposed on the network's architecture:

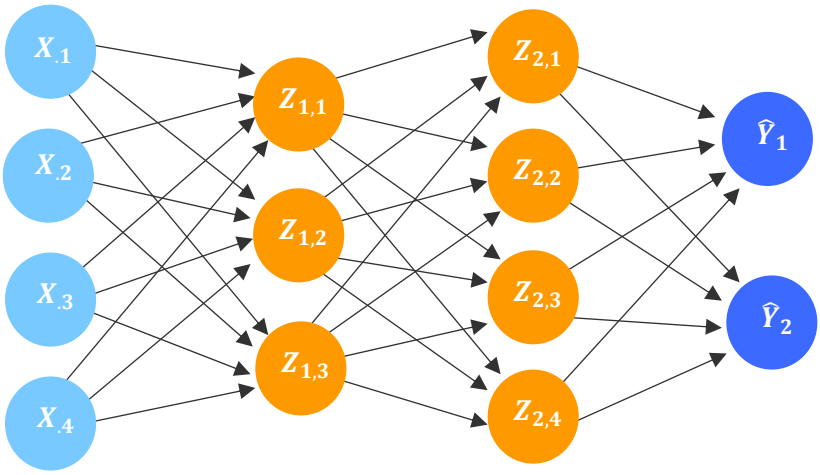
- the neurons are categorised into a series of layers;
  - the original predictor neurons form the input layer;
  - the derived predictor neurons form one or more hidden layers; and

  - the response neurons form the output layer;

- each layer is formed using only the preceding layer; and
- each predictor neuron points to all neurons in the next layer (this is referred to as 'dense' layering).

Figure 5.19 shows an example of a feed-forward neural network with two hidden layers.

Figure 5.19 – A feed-forward neural network



The neural network shown in Figure 5.19 contains:

- 4 (light blue) neurons in the input layer ($X_{.1}, X_{.2}, X_{.3}, X_{.4}$);
- 3 (orange) neurons in the first hidden layer ($Z_{1,1}, Z_{1,2}, Z_{1,3}$);
- 4 (orange) neurons in the second hidden layer ($Z_{2,1}, Z_{2,2}, Z_{2,3}, Z_{2,4}$); and
- 2 (dark blue) neurons in the output layer ($\hat{Y}_1, \hat{Y}_2$).

The neural network shown in Figure 5.19 is just one example of a feed-forward network. The number of hidden layers and neurons forms part of the neural network's 'model architecture'. There are endless options available for designing a neural network's architecture, with different networks having different numbers of hidden layers and different numbers of neurons within each hidden layer.

There is an art to building multi-layer neural networks. The most appropriate architecture to use is often problem-specific. One way to make decisions about the model's architecture is to use validation data (see Section 5.2) to choose the architecture that minimises the model's validation error.

In most cases, a constant variable or 'bias unit' is added as a neuron within each hidden layer. For example, in Figure 5.19:

- a $Z_{1,0}$ neuron with a constant value of 1 could be added as a bias unit to the first hidden layer; and
- a $Z_{2,0}$ neuron with a constant value of 1 could be added as a bias unit to the second hidden layer.

Alternatively, a bias term can be included as an intercept term within the computation of each hidden layer neuron. This is the approach taken in this chapter and is described within the discussion on weights below.

Weights and Activation Functions

Each neuron in a neural network that depends on other neurons (i.e. the orange Zs and dark blue $\hat{Y}$s in Figure 5.19) is estimated by applying an activation function, σ, to a weighted linear combination of the neurons in the preceding layer.

Thus, for the neural network shown in Figure 5.19, the hidden layer and output layer neurons are estimated using the formulae shown in Table 5.7.

Table 5.7 – Estimation of hidden and output layers in a neural network

First hidden layer	Second hidden layer	Output layer
$Z_{1,1} = \sigma(\alpha_{01} + \alpha_1^T X)$	$Z_{2,1} = \sigma(\beta_{01} + \beta_1^T Z_1)$	$\hat{Y}_1 = \sigma(\gamma_{01} + \gamma_1^T Z_2)$
$Z_{1,2} = \sigma(\alpha_{02} + \alpha_2^T X)$	$Z_{2,2} = \sigma(\beta_{02} + \beta_2^T Z_1)$	$\hat{Y}_2 = \sigma(\gamma_{02} + \gamma_2^T Z_2)$
$Z_{1,3} = \sigma(\alpha_{03} + \alpha_3^T X)$	$Z_{2,3} = \sigma(\beta_{03} + \beta_3^T Z_1)$	
	$Z_{2,4} = \sigma(\beta_{04} + \beta_4^T Z_1)$	

Weights

In Table 5.7, the terms α_1^T, α_2^T, α_3^T, β_1^T, β_2^T, etc. represent row vectors of weights that are applied to each neuron in the preceding layer of the network. For example, if α_1^T is a row vector containing the weights 0.2, 0.1, 0.4 and 0.1 (i.e. α_1^T = [0.2, 0.1, 0.4, 0.1]), then the first neuron in the first hidden layer ($Z_{1,1}$) is estimated as:

$$Z_{1,1} = \sigma(\alpha_{01} + 0.2 \times X_{.1} + 0.1 \times X_{.2} + 0.4 \times X_{.3} + 0.1 \times X_{.4}).$$

The terms α_{01}, α_{02}, α_{03}, β_{01}, β_{02} etc. are the bias terms that are included in the hidden and output layers of the model. These are analogous to the role of a constant or intercept in a linear function, whereby the line is effectively shifted by the constant value. This inclusion of a bias term provides greater flexibility to the model to help it better fit the data on which it is trained.

One of the key stages in building a neural network is to estimate the 'best' weights to use in the network, so the model's chosen loss function is minimised. In other words, the neural network must find a set of weights that results in a good fit to the training data. This process of finding a good set of weights is discussed in Section 5.5.5.

Activation Functions

In Table 5.7, the function represented by σ is called the activation function. The activation function of a neuron defines the output of that neuron, given a set of inputs. Activation functions tell a neuron whether to 'activate' based on the weighted linear combination of neurons in the preceding layer. This is akin to the neurons in the brain that either fire or do not fire, based on the input of prior input neurons, as explained in Video 5.6.

Video 5.6 – Activation functions

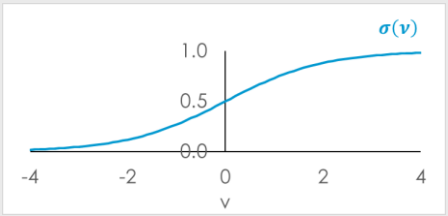
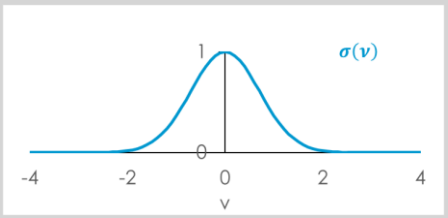
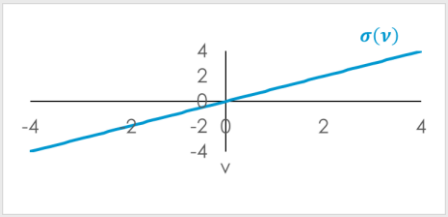
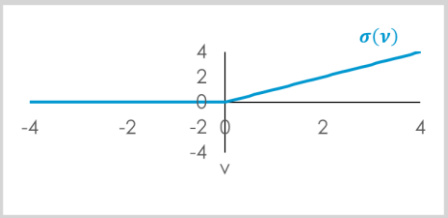
https://www.youtube.com/watch?v=m0pIlLfpXWE (5 mins)

Record your video notes here

Activation functions allow a neural network to capture nonlinear relationships between the feature variables and response variable in the training data.

Some common choices for activation functions are shown in Table 5.8.

Table 5.8 – Common activation functions

Function	Formula	Diagram/comment
Sigmoid	$\sigma(v) = \frac{1}{1+e^{-v}}$	
Radial basis	$\sigma(v) = e^{-v^2}$	
Identity	$\sigma(v) = v$	
Rectified Linear Unit or ReLU	$\sigma(v) = max(0, v)$	
Softmax	$\sigma(v_i) = \frac{e^{v_i}}{\sum_{j=1}^{k} e^{v_j}}$ k = number of neurons in the current layer	This activation function ensures that the sum of the neurons in a layer equals 1.

In Table 5.8, v represents the weighted linear combination of neurons in the preceding layer of the neural network. For example, in estimating the first neuron in the second hidden layer from Figure 5.19 ($Z_{2,1} = \sigma(\beta_{01} + \beta_1^T Z_1)$), the activation function, σ, applies to $v = \beta_{01} + \beta_1^T Z_1$.

Exercise 5.15

Outline your observations on each of the activation functions shown in Table 5.8.

The most common setup for a binary classification network is to use the Sigmoid activation function in the hidden and output layers, with the single response neuron having a value between 0 and 1, which indicates the probability of a positive response.

However, the activation function does not have to be identical in each layer. In multiclass classification, where three or more classes are involved, it is common to use an activation function such as Sigmoid or ReLU for each of the hidden layers, and the Softmax function for the output layer.

Exercise 5.16

Explain why the Softmax function might be suitable for the output layer in a classification setting.

When choosing the activation functions to use in a neural network, it is important to consider factors such as:

- the type of problem you are trying to solve, such as regression, binary classification or multinomial/multiclass classification;
- how difficult it is to compute the derivative of the function (you will learn about the need to compute derivatives in Section 5.5.4);
- how quickly a network with your chosen activation function converges to a solution; and
- how well the resulting model fits to the training and test data.

You may need to try different activation functions in your network, using validation data or cross-validation to decide on the most suitable function(s) to use.

It is also worth noting that different activation functions used in the final output layer will impact the form that the model's outputs, $\hat{Y}$, take. As highlighted in Table 5.8, use of the Softmax function in the output layer will ensure that the sum of the neurons in the output layer equals 1, such that $\hat{y}_{i,g_k}$ represents $\hat{p}_{i,g_k}$. This will not necessarily be the case for the other activation functions shown in Table 5.8.

Exercise 5.17

Consider a binary classification setup for a neural network that has no hidden layers and uses the Sigmoid function for the output layer.

Identify the supervised learning model that this neural network is equivalent to.

5.5.3. Forward Propagation

Forward propagation is the process in a neural network where the input data is fed in a forward direction through the network to make a prediction or classification of the response variable. The steps in this process are as follows:

- the first hidden layer accepts input data from the original predictor neurons—the light blue Xs in Figure 5.19—and converts this input into derived predictor neurons—the orange Zs in Figure 5.19;
- each subsequent hidden layer accepts the derived predictor neurons from the previous hidden layer and further converts these into new derived predictor neurons; and
- the output later accepts the derived predictor neurons from the last hidden layer and converts these into response neuron(s)—the dark blue $\hat{Y}$s in Figure 5.19.

After the above three steps have been taken, the response neuron(s) are converted into a prediction of the response for each observation in the input data.

In a binary classification setting, the response can take only two values, such as 'positive' (1) or 'negative' (0). In this case, there is usually only one response neuron, $\hat{Y}_1$ or just $\hat{Y}$, which takes a value between 0 and 1. This value represents the probability that the response is 'positive'. Any values of $\hat{y}_{i,1}$ at or above a certain threshold (e.g. 0.5) are classified as being 'positive' and any values of $\hat{y}_{i,1}$ below this threshold are classified as 'negative'. For example, in a spam classifier model, if the classification threshold is set at 0.5 and $\hat{y}_{i,1}$ = 0.79 for observation i, this email would be classified as spam because
$\hat{y}_{i,1} \geq 0.5$.

As discussed in Section 5.2, the threshold used to classify each observation based on the modelled values of $\hat{Y}$ does not have to be 0.5. In practice, the threshold chosen will depend on the relative cost of false positives and false negatives for the given business context. For example, in a pandemic, when a model might be trying to predict whether someone has the virus and, therefore, needs to be quarantined, a lower threshold than 0.5 might be chosen to reduce the number of people with the virus accidentally escaping quarantine.

In a multiclass classification setting, the response can take one of the K values in $G = \{g_1, g_2, \dots, g_K\}$. In this case, the neural network will have K response neurons $(\hat{Y}_1, \hat{Y}_2, \dots, \hat{Y}_K)$. A classification for each observation, i, is made by selecting the class, k, with the largest value of $\hat{y}_{i,g_k}$. For example, a neural network may be built to classify an image as being a cat, dog, horse or fish. In this case, there are four possible classes (K=4), and the neural network will contain four response neurons $(\hat{Y}_1, \hat{Y}_2, \hat{Y}_3, \hat{Y}_4)$, with:

- 1 representing cats;
- 2 representing dogs;
- 3 representing horses; and
- 4 representing fish.

Imagine that for one image:

- $\hat{y}_{i,1}$ = 0.23;
- $\hat{y}_{i,2}$ = 0.70;
- $\hat{y}_{i,3}$ = 0.05; and
- $\hat{y}_{i,4}$ = 0.02.

This image would be classified as being a dog (g_2= dog) because $\hat{y}_{i,2}$ is the response neuron with the largest value.

XNOR Gate Example[13]

This section sets out a simple example to help illustrate how the forward propagation process works in a neural network. The example also helps to provide you with an intuitive understanding of what a neural network does.

[13] This example has been sourced from Ng., A. (2020). Week 4: Neural Networks: Representation [MOOC lecture]. In Stanford University. *Machine Learning*. Coursera. https://www.coursera.org/learn/machine-learning/home/welcome.

Imagine that you have a hallway light in your house that is connected to two switches—one at each end of the hallway. Each switch can be in one of two positions: up or down. If both switches are up or both switches are down, the light will be on. If one switch is up but the other switch is down, the light will be off.[14] This is known as an 'XNOR Gate', which is a type of digital logic gate that receives two inputs, such as the position of the two switches, and produces one output; in this case, whether the light is on or off.

Let:

- $X_{.1}$ represent the position of the first switch, where 1 = up and 0 = down;
- $X_{.2}$ represent the position of the second switch, where 1 = up and 0 = down; and
- Y represent the light's setting, where 1 = on and 0 = off.

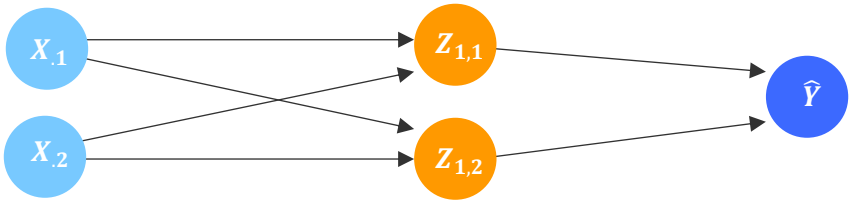
Table 5.9 shows the four different value combinations for x_{i1} and x_{i2} and the corresponding value for y_i.

Table 5.9 – XNOR example

x_{i1}	x_{i2}	y_i
0	0	1
0	1	0
1	0	0
1	1	1

This XNOR Gate can be represented using a suitably designed neural network, such as the one shown in Figure 5.20.

Figure 5.20 – XNOR example as a neural network



[14] Most light switches are wired to achieve the opposite effect to that described, where both switches being in the same position turns a light off and the switches in different positions turns a light on.

The value of y_i within this neural network can be predicted for each possible combination of x_{i1} and x_{i2} using:

- the weights for each neuron, as shown in Figure 5.20 (α_{11}, α_{12}, etc.);
- the sigmoid function ($\sigma(v) = \frac{1}{1+e^{-v}}$) as the activation function in all layers;
- bias terms in the hidden layer of $\alpha_{01} = -30$ and $\alpha_{02} = 10$; and
- a bias term in the output layer of $\beta_{01} = -10$.

Table 5.10 shows the calculations required to forward propagate the neural network described above and estimate values for y_i given different values of x_{i1} and x_{i2}.

Table 5.10 – Forward propagation for XNOR example

x_{i1}	x_{i2}	$z_{1,1}$	$z_{1,2}$	$\hat{y}_i$	y_i	$y_i - \hat{y}_i$
0	0	$= \sigma(\alpha_{01} + \alpha_1^T X)$ $= \sigma(-30 + 20 \times 0 + 20 \times 0)$ $= \sigma(-30)$ ≈ 0	$= \sigma(\alpha_{02} + \alpha_2^T X)$ $= \sigma(10 - 20 \times 0 - 20 \times 0)$ $= \sigma(10)$ ≈ 1	$= \sigma(\beta_{01} + \beta_1^T Z_1)$ $= \sigma(-10 + 20 \times 0 + 20 \times 1)$ $= \sigma(10)$ ≈ 1	1	0
0	1	$= \sigma(\alpha_{01} + \alpha_1^T X)$ $= \sigma(-30 + 20 \times 0 + 20 \times 1)$ $= \sigma(-10)$ ≈ 0	$= \sigma(\alpha_{02} + \alpha_2^T X)$ $= \sigma(10 - 20 \times 0 - 20 \times 1)$ $= \sigma(-10)$ ≈ 0	$= \sigma(\beta_{01} + \beta_1^T Z_1)$ $= \sigma(-10 + 20 \times 0 + 20 \times 0)$ $= \sigma(-10)$ ≈ 0	0	0
1	0	$= \sigma(\alpha_{01} + \alpha_1^T X)$ $= \sigma(-30 + 20 \times 1 + 20 \times 0)$ $= \sigma(-10)$ ≈ 0	$= \sigma(\alpha_{02} + \alpha_2^T X)$ $= \sigma(10 - 20 \times 1 - 20 \times 0)$ $= \sigma(-10)$ ≈ 0	$= \sigma(\beta_{01} + \beta_1^T Z_1)$ $= \sigma(-10 + 20 \times 0 + 20 \times 0)$ $= \sigma(-10)$ ≈ 0	0	0
1	1	$= \sigma(\alpha_{01} + \alpha_1^T X)$ $= \sigma(-30 + 20 \times 1 + 20 \times 1)$ $= \sigma(10)$ ≈ 1	$= \sigma(\alpha_{02} + \alpha_2^T X)$ $= \sigma(10 - 20 \times 1 - 20 \times 1)$ $= \sigma(-30)$ ≈ 0	$= \sigma(\beta_{01} + \beta_1^T Z_1)$ $= \sigma(-10 + 20 \times 1 + 20 \times 0)$ $= \sigma(10)$ ≈ 1	1	0

As shown in Table 5.10, the neural network constructed in this section successfully predicted that:

- the light will be on ($\hat{y}_i = 1$) when both switches are in the same position ($x_{i1} = x_{i2}$); and
- the light will be off ($\hat{y}_i = 0$) when the switches are in different positions ($x_{i1} \neq x_{i2}$).

The neural network used the given weights and bias terms to make perfect predictions of the response variable for every observation in the training data (i.e. $\hat{y}_i = y_i$ for each observation). Therefore, using these weights and bias terms leads to a model with a loss function of 0. This is a rare outcome as it is usually not possible, nor even desirable, to build a model that results in a loss function of 0.

This is an example of a neural network where the raw outputs, $\hat{Y}_{g_k}$, represent values in G, rather than $\hat{P}_{g_k}$, and no further transformations are required.

Exercise 5.18

Explain why it might be undesirable to build a model that has a loss function of 0.

In practice, a goal in building a neural network is to find weights and bias terms that reduce the model's prediction error or loss function to an acceptable level. This is referred to as 'training' the model. One process to estimate these weights and bias terms—backpropagation—is discussed in the next section.

5.5.4. Backpropagation

Backpropagation is a popular method for training a feed-forward neural network. Backpropagation is a gradient descent algorithm (see Section 5.4.2) that is tailored to the model structure of a neural network. The intuition behind backpropagation is explained in Video 5.7.

Video 5.7 – Backpropagation

https://www.youtube.com/watch?v=XE3krf3CQls (11 mins)

Record your video notes here

To apply a gradient descent algorithm, it is necessary to first define:

- the loss function, $J(\theta_1, \dots, \theta_p)$, that you are trying to minimise; and
- the partial derivatives of the loss function with respect to each of the weights $(\theta_1, \dots, \theta_p)$ in the model $(\frac{\partial}{\partial \theta_j} J(\theta_1, \dots, \theta_p))$.

Once these are defined, the standard gradient descent approach to updating the weights can be applied.

Neural Network Loss Function

Section 5.3.1 provided a recap on logistic regression and its use in solving classification problems. A loss function that is commonly used for neural network classification is the logistic loss function, as per logistic regression. For binary classification, this is shown in Equation 5.10, using the 0–1 representation of the categories.

Equation 5.10 $$\boldsymbol{J(\theta_1, \dots, \theta_p) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_{i,1}) + (1 - y_i) \log(1 - \hat{y}_{i,1})]}$$

where y_i is the actual response and $\hat{y}_{i,1}$ is the estimated response for the i^{th} observation.

For a neural network used to solve a classification problem with k classes, the following modified version of the logistic regression loss function can be used:

Equation 5.11 $$\boldsymbol{J(\theta_1, \dots, \theta_p) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} I\{g_i = g_k\} \log \hat{y}_{i,g_k}}$$

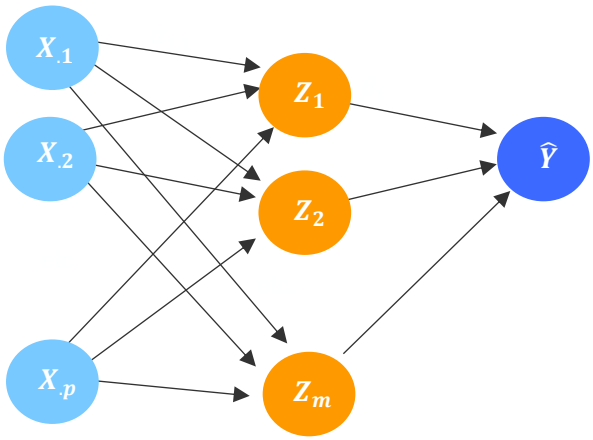
In Equation 5.10 and Equation 5.11, $\theta_1, \dots, \theta_p$ denotes all the weights ($a_1^T, a_2^T, \dots, a_m^T, \beta_1^T, \beta_2^T, \dots, \beta_m^T$ etc.) in the neural network. Often the $\frac{1}{n}$ term is dropped from these expressions as it is a constant for a given dataset and, therefore, it does not impact the choice of weights when trying to minimise $J(\theta_1, \dots, \theta_p)$.

Partial Derivatives of the Loss Function

For a specified neural network, weights and activation functions, it is possible to write down the model's loss and 'unpack' this loss into its different components. These components are contributed from each of the different weights in the model that go into the estimates $\hat{Y}_{g_k}$. From there, standard calculus, including repeated application of the chain rule,[15] allows the partial derivatives to be estimated in respect of each of the model's weights.

This concept is illustrated below in the context of a single-layer feed-forward neural network, as depicted in Figure 5.21.

Figure 5.21 – A single-layer feed-forward neural network



In Figure 5.21, α and β index the weights for the first and second layers of the network, as described in Section 5.5.2. For this binary classification illustration:

- there is a single response neuron $\hat{Y}$;
- the bias terms $\alpha_{01}, \alpha_{02}, \ldots, \alpha_{0m,} \beta_{01}$ are ignored (i.e. set to 0) for simplicity;
- the same activation function, σ, with derivative, σ', is used for all neuron calculations; and
- the overall loss is calculated using Equation 5.10.

[15] The following video provides a reminder of the chain rule in calculus: https://www.khanacademy.org/math/ap-calculus-ab/ab-differentiation-2-new/ab-3-1a/v/chain-rule-introduction.

For each observation i, the prediction $\hat{y}_{i,1}$ can be expressed in terms of its reliance on both the first and second layers of the neural network:

Equation 5.12 $$\hat{\boldsymbol{y}}_{i,1} = \boldsymbol{\sigma}(\boldsymbol{\beta}^T \boldsymbol{Z}_i) = \boldsymbol{\sigma}\left(\sum_{m=1}^{M} \boldsymbol{\beta}_m \boldsymbol{Z}_{m,i}\right) = \boldsymbol{\sigma}\left(\sum_{m=1}^{M} \boldsymbol{\beta}_m \boldsymbol{\sigma}(\boldsymbol{\alpha}_m^T \boldsymbol{X}_{i.})\right)$$

Where $Z_{m,i}$ represents the m^{th} neuron in the single hidden layer for observation i.

Substituting Equation 5.12 into Equation 5.10, and taking partial derivatives for the β and α weights respectively, generates:

Equation 5.13 $$\frac{\partial J}{\partial \boldsymbol{\beta}_m} = -\frac{1}{n}\sum_{i=1}^{n}\left\{\frac{y_i}{\sigma(\boldsymbol{\beta}^T \boldsymbol{Z}_i)}\boldsymbol{\sigma}'(\boldsymbol{\beta}^T \boldsymbol{Z}_i) - \frac{1-y_i}{1-\sigma(\boldsymbol{\beta}^T \boldsymbol{Z}_i)}\boldsymbol{\sigma}'(\boldsymbol{\beta}^T \boldsymbol{Z}_i)\right\}\boldsymbol{Z}_{m,i}$$

Equation 5.14

$$\frac{\partial J}{\partial \boldsymbol{\alpha}_{m,j}} = -\frac{1}{n}\sum_{i=1}^{n}\left\{\frac{y_i}{\sigma(\boldsymbol{\beta}^T \boldsymbol{Z}_i)}\boldsymbol{\sigma}'(\boldsymbol{\beta}^T \boldsymbol{Z}_i)\boldsymbol{\beta}_m\boldsymbol{\sigma}'(\boldsymbol{\alpha}_m^T \boldsymbol{X}_{i.}) - \frac{1-y_i}{1-\sigma(\boldsymbol{\beta}^T \boldsymbol{Z}_i)}\boldsymbol{\sigma}'(\boldsymbol{\beta}^T \boldsymbol{Z}_i)\boldsymbol{\beta}_m\boldsymbol{\sigma}'(\boldsymbol{\alpha}_m^T \boldsymbol{X}_{i.})\right\}\boldsymbol{X}_{i,j}$$

The derivatives set out in Equation 5.13 and Equation 5.14 are all sums of known quantities for a given set of starting parameters. One way of viewing the partial derivatives is by considering the functions inside the curly brackets as new weights or factors that are used to create a linear sum in $Z_{m,i}$ or $X_{i,j}$ to construct the partial derivative.

The other observation is that many of the terms in $\frac{\partial J}{\partial \beta_m}$ also need to be calculated in $\frac{\partial J}{\partial \alpha_{m,j}}$.

Therefore, it is most efficient to work backwards in calculating terms for the later layers in the neural network first, which then get passed back to earlier layers in the network.

Thus, the backpropagation algorithm can be implemented by:

- performing a 'forward pass' through the network with the starting set of parameters to calculate the value for each neuron in each layer, including $\hat{y}_k$ for each class k in the output layer;
- using these values and Equation 5.13 and Equation 5.14 to calculate partial derivatives; and
- updating each weight in the network using gradient descent as per Equation 5.9.

This process can be repeated until a point of (potentially local) minimum loss is found, or an alternative stopping rule is applied.

In Python, packages such as Keras, which runs on top of the TensorFlow library, automatically perform forward and backpropagation for you. Similar packages also exist in other programming languages such as R. Therefore, you do not usually need to work through the above calculations yourself, and you are not expected to do so in the assessments for this subject. However, it is helpful to understand what these packages are doing as this can help you to interpret the output of these packages and make decisions such as how many iterations you want an algorithm to perform.

Exercise 5.19

The notebook 'DSA_C05_Ex19.ipynb' builds and trains a neural network for the spam classifier problem. Review the code in the notebook and play with different neural network architectures by varying:

- the number of hidden layers;
- the number of neurons in the hidden layer(s); and
- the activation functions used in the hidden layer(s).

Compare the impact of these different architectures on the validation accuracy of the model after 100 epochs (an epoch refers to one cycle through the full training dataset).

Identify, for each of these model architectures, how many epochs were required before this validation accuracy was first reached.

There are alternative ways to train feed-forward networks apart from the backpropagation method taught in this section. These alternatives include:

- second derivatives techniques, such as Newton's method;
- conjugate gradients; and
- variable metric methods.

These alternative methods are outside the scope of this subject.

5.5.5. Practical Considerations

Sections 5.5.2 to 5.5.4 outlined the mathematical theory and concepts behind neural networks. This section discusses some of the important considerations when applying the theory of neural networks in practice. These considerations fall into the following groups:

- unit of analysis;
- feature engineering;
- building and training;
- testing; and
- deployment.

Unit of Analysis

In machine learning, the unit of analysis refers to the primary entity or object that you are studying and making predictions about. Each row in a dataset typically represents a unique observation, instance, or unit of analysis. This could be an individual person, transaction, event, or any discrete entity that the model is designed to learn from.

The **entity** refers to the subject of each observation. For example, in customer churn prediction, the entity might be a customer; in a transaction dataset, it could be a specific purchase or financial transaction. Entities provide the "who" or "what" the prediction is focused on, and each entity usually corresponds to a distinct row (though there may be multiple observations per entity in time series problems).

When timestamps are involved, the unit of analysis might be an observation at a specific time for a given entity. For example, in a time-series dataset where you're predicting future values based on historical data, the unit of analysis is a time-stamped observation of a particular entity (such as the daily temperature in a specific city). In this case, the uniqueness of a row is defined by a combination of the entity and the timestamp (i.e., the specific point in time for a given entity).

Understanding the unit of analysis is essential because it guides how you frame the problem, prepare the data, and interpret the model's predictions.

Feature Engineering

Feature engineering is a crucial step in the machine learning process, where raw data is transformed into meaningful features that enhance a model's ability to make accurate predictions. The purpose of feature engineering is to create or modify features in ways that improve the model's performance, make it more interpretable, and ensure it generalises well to new data.

Key purposes of feature engineering:

- Improve Model Performance: Well-engineered features capture the important patterns and relationships in the data, making it easier for machine learning algorithms to distinguish between different outcomes. Transforming raw data into more informative features often leads to higher predictive accuracy by giving the model clear and relevant inputs.
- Simplify Models: By crafting useful features, you can often reduce the complexity of a model. For example, instead of feeding the raw date of a transaction into a model, extracting features like the day of the week or hour of the day can provide more intuitive and relevant information.
- Increase Model Interpretability: Carefully engineered features can help make models more interpretable by providing inputs that are easier to understand and explain. This is especially important in domains like healthcare or finance, where model transparency is critical.
- Handle Data Quality Issues: Feature engineering often involves dealing with missing data through imputation or creating "missingness indicators" to capture whether data is absent. Smoothing or aggregating raw data can help remove noise, leading to more stable and reliable features for the model to learn from.
- Enhance the Model's Generalisation Ability: Well-engineered features help a model generalize better to unseen data, reducing overfitting and ensuring it performs well on out-of-sample predictions. Incorporating domain knowledge into the features allows the model to leverage context-specific information that can improve generalization in real-world settings.

Feature engineering ideations is easier when you approach it within a framework. One such framework is feature signal types. Lists of signal types and entities can serve as inspiration for feature engineering by providing ideas on how to create new features based on different types of signals and entities. Features with diverse signal types are typically less correlated, which can be beneficial for model performance as they provide varied information to the model. Business Subject Matter Experts can easily grasp the concept of signal types, making it easier to collaborate on feature engineering tasks.

Here is a list of signal types, with their definitions:

- Clumpiness signals refer to the pattern of events occurring randomly across time or if there are long gaps between intense bursts of activity. These signals help in understanding the distribution and timing of events. Examples of clumpiness signals include identifying binge-watching customers and analysing equipment failures occurring in cascades.
- Inventory signals refer to the counts or amounts of events or available resources categorized by an item label. These signals include metrics such as counts of item types, sum of values of item types, and maximum values of item types. Examples of inventory signals include counts of each item type in a shopping basket, total value of item types for customer shipping orders, and maximum weight of each type of item in a shipping order.
- Similarity signals refer to the measurement of how similar one entity is to a group of others. This can be determined by ratios of numeric values or cosine similarities of cross-aggregations between entities. Examples include comparing a customer's purchases to others of the same age and gender.
- Stability signals refer to assessing if an entity's recent events are similar to its past occurrences. This involves analysing metrics like the ratio of the latest numeric value to historical averages or max values, and cosine similarity of cross-aggregations from different time periods.
- Change signals refer to identifying if there have been alterations in typically static attributes and quantifying the extent of these changes. This involves tracking metrics such as the number of times an attribute has changed and the magnitude of each change. Examples include tracking changes in customer addresses over a period or monitoring password resets within a defined timeframe.
- Diversity signals refer to the variability in data values, measured by metrics such as the coefficient of variation of amounts and entropy of labels. These signals help assess how different or varied the data points are. Examples of diversity signals include determining the variability of monthly invoice amounts over the past 6 months, assessing the stability of a patient's blood pressure, and analysing if a customer consistently purchases similar products.
- Location signals refer to the static or dynamic position of an event or entity. This includes metrics such as latitude and longitude, distance between locations, or distance moved within a specific time frame. Examples of location signals include determining the distance between a shop and a customer address, identifying if a shipping address is in a remote location, and calculating population density in a particular state.

- Regularity signals refer to identifying any patterns or seasonality in the timing of events. This involves analysing metrics such as the entropy of the day of the week of events and cross-aggregation of events by hour or month. Examples of regularity signals include measures of whether a customer consistently shop on the same days of the week, and whether equipment failures tend to occur at specific times of the day or year.
- Recency signals refer to attributes of the latest event that has occurred. This includes metrics such as the time since the last event, the label of the last event, and the magnitude of the last event. For example, recency signals can help determine how much a customer spent on their last purchase, how long it has been since the last equipment failure, or whether a patient's last visit was for an emergency.
- Frequency signals refer to how often events occur within a specified time frame. It involves measuring the number of events happening over a particular period. For example, tracking the frequency of hospital admissions in the past 12 months or the number of international phone calls made by a customer in the last month.
- Monetary signals refer to the amounts associated with events occurring over a specific time window. This includes the total cost and average cost of these events. Examples of monetary signals include determining how much a customer spent over the past 12 months and calculating the average discount applied to a customer's purchases over the past month.

Similarly to your unit of analysis, features are calculated at an entity level, but that entity may be different to the entity in your unit of analysis. A feature at parent entity level, aggregates data to a group that your primary entity (from your unit of analysis) belongs to e.g. gender, or country, and can be joined to the unit of analysis as a robust signal source. Features at child entity level are more granular than your unit of analysis, e.g. a single item with a basket of purchases, and must be aggregated before they can be used at your unit of analysis.

Building and Training

When training a neural network to find weights that minimise the loss function, choices must be made about, among other things:

- the learning rate; and
- the batch size.

As discussed in Section 5.4.2, a **learning rate** is used in gradient descent to determine the step size that the algorithm will use when updating model parameters such as the weights in a neural network.

If a small learning rate is used, more iterations of the gradient descent algorithm will be required before a minimum value of the loss function is found. If the learning rate is too low, the number of iterations might need to be restricted to ensure the neural network is trained in a timely manner (see the discussion on early stopping below).

Conversely, if the learning rate used is too high, then there is a risk that the algorithm will 'overshoot' the minimum value of the loss function and not find a good fit for the data. It can be helpful to plot the calculated loss after each iteration of the gradient descent algorithm to ensure the algorithm is converging on a minimum value.

To achieve a balance between learning quickly in the early stages of the training and achieving a fine-tuned fit in the later stages of training, the learning rate can be varied over the training stage with a 'learning rate scheduler'.

Another consideration in training a neural network is the **batch size** of data that is used. For larger datasets, it is common to update a neural network using batches of data rather than the full dataset. Similarly, in many applications, observations may come in either one at a time or in small batches, and the neural network may need to be retrained on each of these small batches.

If a neural network is updated using batches of data, the backpropagation algorithm can be adjusted so the partial derivatives are calculated only with respect to a small group of observations, rather than performing the algorithm over the entire dataset. As outlined in Section 5.4.2, this is known as stochastic gradient descent.

Video 5.8 discusses an experiment that uses different batch sizes to train a neural network.

Video 5.8 – Learning rates and batch sizes

https://www.youtube.com/watch?v=dSqKGNT0qaw (7 mins)

Record your video notes here

The following practical issues must also be considered when building and training a neural network:

- overfitting;
- underfitting;
- starting values;
- multiple minima;
- vanishing and exploding gradients;
- categorical features;
- feature scaling; and
- granularity requirements.

Overfitting

The number of parameters in a neural network grows very quickly as the number of layers and neurons increase. In a neural network with many parameters, the iterative model fitting process may lead to an extremely good fit on the training data, but the trained network may fail to generalise well to new data. Overfitting is the largest challenge faced with using neural networks compared to other statistical learning methods such as linear models or tree-based models.

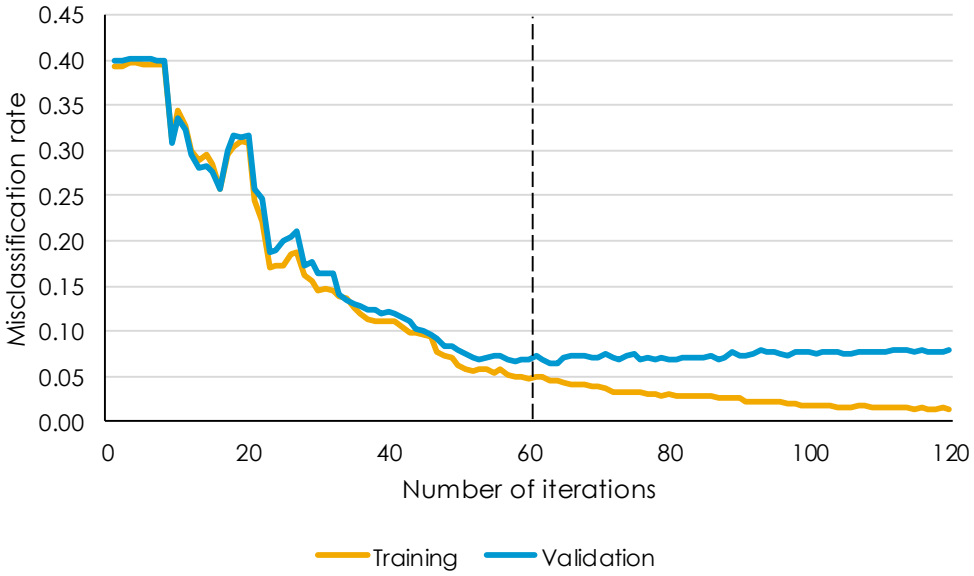
As described in Section 5.4.1, this is known as a model with high variance. Four common methods to avoid such overfitting to the training data are:

- early stopping;
- pruning;
- regularisation; and
- initial dimension reduction of the features.

Early stopping involves training a neural network until the performance of the model on a validation dataset starts to deteriorate. The training algorithm is run iteratively until the validation error starts to increase. This method allows the model to run long enough to learn the relationships in the data, but not so long that the model overfits the training data.

Figure 5.22 illustrates the use of early stopping in training a neural network to classify the spam dataset introduced in Section 5.1.2.

Figure 5.22 – Misclassification rates for spam neural network, 10 hidden neurons



In Figure 5.22, the validation error (the blue line) no longer decreases after approximately 60 iterations, so the training of the neural network might be forced to stop after 60 iterations to avoid overfitting the model.

Early stopping can have issues. Firstly, the validation error will never be perfectly smooth, so knowing when to stop can be difficult. To get around this, early stopping might involve continuing to train the network until the validation error has not improved for, say, 10 iterations.

Also, the optimal stopping point will depend on the initial parameters chosen, which are generally chosen randomly (see the discussion below on starting values/multiple minima).

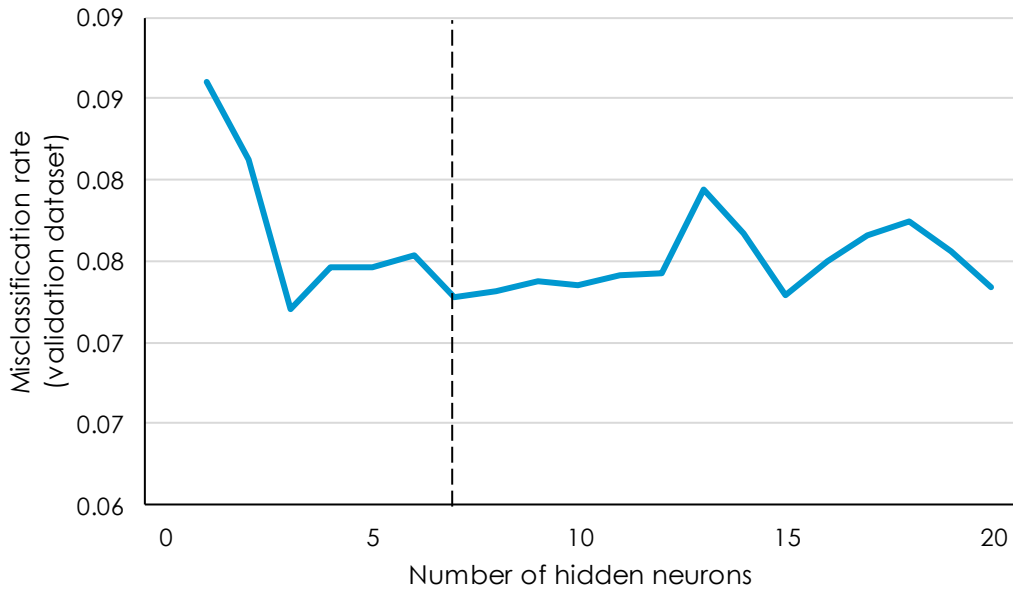
Pruning is another method used to avoid overfitting. It involves choosing the optimal number of hidden layers and neurons within each hidden layer so the validation error will not begin to decay after a given number of iterations.

The idea behind pruning is that among the many parameters in the network, some are redundant. Various methods can be used to rank the importance of neurons in the network. The details behind these methods are outside the scope of this subject. The least important neurons can be removed from the network, resulting in a smaller and faster network that is more likely to generalise well (i.e. perform well on a validation dataset).

The optimal number of hidden neurons could also be found by initially starting with a relatively simple network with a small number of neurons. This simple network will tend to underfit the training data. Upon each iteration of the fitting procedure—backpropagation or otherwise—the model complexity increases. Using this process, there will usually be some optimum point where the model is simple enough to avoid overfitting the training data but complex enough to capture the key features and patterns in this data. This optimal point can be discovered by monitoring each subsequent network's performance on a validation set. The final model chosen is the one that minimises the error on the validation set.

Figure 5.23 illustrates the use of pruning in training a neural network to classify the spam dataset from Section 5.1.2.

Figure 5.23 – Misclassification rates for spam neural network, varying hidden neurons



Based on the information provided in Figure 5.23, the misclassification rate on the validation dataset does not continue to decrease with more than 7 hidden neurons, so there is little additional value in using a model with more than 7 hidden neurons for this single-layer setup.

Another practical way to avoid overfitting is through **regularisation**. This involves adjusting the loss function, $J(\theta_1, \dots, \theta_p)$, shown in Equation 5.11 as follows:

Equation 5.15

$$J^*(\boldsymbol{\theta_1}, \dots, \boldsymbol{\theta_p}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} I\{g_i = g_k\} log\, \hat{y}_{i,g_k} + \lambda \sum_{j=1}^{p} \theta_j^2$$

$$= J(\boldsymbol{\theta_1}, \dots, \boldsymbol{\theta_p}) + \lambda \sum_{j=1}^{p} \theta_j^2$$

where λ is the regularisation parameter and $\sum_{j=1}^{p} {\theta_j}^2$ is the sum of the squares of the weights in the network.

The regularisation term $\lambda \sum_{j=1}^{p} {\theta_j}^2$ in Equation 5.15 penalises a model that has too many parameters or large parameter values. This has the effect of shrinking some of the weights in a network towards zero, effectively removing the relevant neurons from the model. Higher values of λ will lead to a greater penalty being applied to the model's weights. This will, in turn, lead to a simpler neural network being fitted.

As for the other methods to avoid overfitting that are discussed above, the optimal value of λ can be chosen using cross-validation or a single validation dataset.

Dimension reduction can also help to avoid overfitting. The number of input variables or features of a dataset is referred to as its dimensionality. Dimension reduction involves using techniques that reduce the number of input variables in a dataset, while still retaining some meaningful properties of the original data. Dimension reduction can be achieved through a feature selection process (as covered in Data Science Principles), or by adopting a technique such as principal component analysis (see Chapter 6, Unsupervised learning).

Underfitting

Underfitting is the reverse of overfitting and occurs when too simple a model is used. This results in the model not fitting very well to the training data. Underfitting can be overcome by:

- obtaining more training data that the model can learn from; and
- adding more hidden layers and/or hidden neurons to the model's architecture.

Starting Values

When training a neural network, initial values for the model's weights must first be chosen. This is known as initialising the weights. Unlike in other classification models such as logistic regression, initialising the hidden layer weights to zero does not work with neural networks. This is because when the backpropagation algorithm is run, if the initial weights are all set to zero, each of the neurons within each hidden layer will update to the same value repeatedly. This is undesirable, as the power of the model with multiple neurons in each hidden layer will be lost.

Therefore, whilst the output layer weights can be initialised to zero (and this may help with exploding gradient and convergence issues associated with a higher learning rate—see below), it is common to initialise the hidden weights to random, non-zero values. These are usually chosen as small values, as starting with large weights can also lead to poor model outcomes.

Multiple Minima

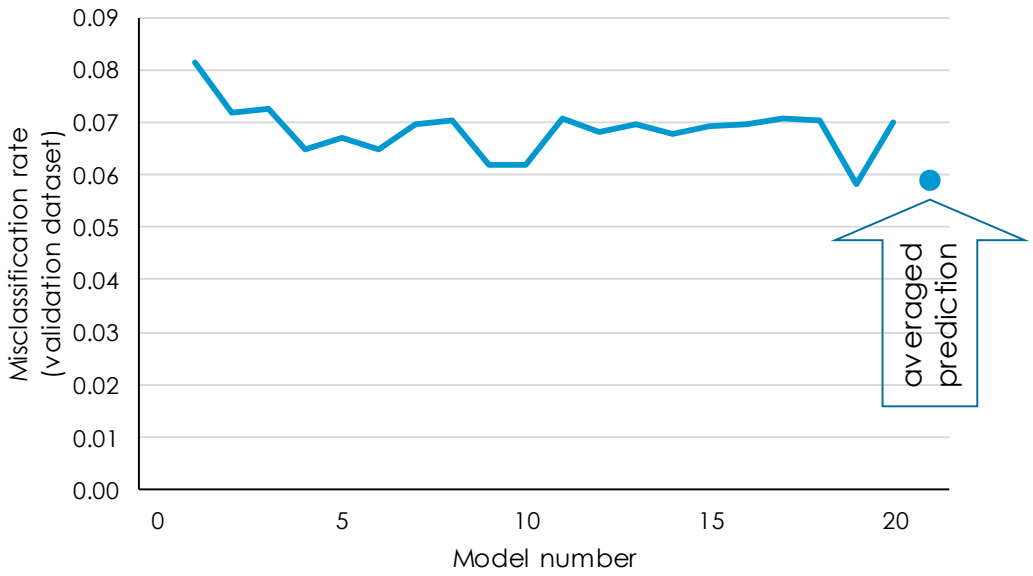
One consequence of randomising the initial weights—particularly if the algorithm is stopped after a fixed number of iterations—is that the trained model will be different depending on the chosen starting weights. This is one of the reasons the model contained in the notebook for Exercise 5.19 produces slightly different outcomes each time the model is run.

There are often many possible local minima for the loss function, so different starting weight values can converge to different trained models.

One way to overcome this limitation of randomising the initial weights is to train several models and select the best-performing model based on cross-validation or the use of a single validation dataset.

An alternative is to average the predictions of several trained networks. This approach is illustrated in Figure 5.24, which shows the validation error achieved for 20 neural networks that were trained to classify the spam dataset from Section 5.1.2.

Figure 5.24 – Misclassification rates for spam neural network with 20 different initialisations



In Figure 5.24, the validation error of the averaged predictions is lower than all but one of the 20 individual simulations. This suggests that a more stable set of estimates has been obtained by ensembling the 20 individual models.

Vanishing and Exploding Gradients

The **vanishing gradient** problem is encountered when training a neural network with gradient-based learning methods and backpropagation. Under these methods, as discussed in Section 5.5.4, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that, in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.

The simplest solution for vanishing gradients is to experiment with different activation functions that do not cause very small derivatives and, therefore, prevent the gradients from vanishing.

On the flip side, when using activation functions with derivatives that can take on larger values, there is a risk of encountering the related **exploding gradient** problem. This problem arises when large error gradients accumulate and result in very large updates to neural network model weights during training. This has the effect of the model being unstable and unable to learn from the training data.

Exploding gradients may be addressed by redesigning the network to have fewer layers. There may also be some benefit in using a smaller batch size or learning rate while training the network.

Categorical Features

Categorical features are those that contain label values, such as 'black', 'green' or 'red', rather than numerical values. Unlike tree-based classification models that handle categorical features without the need to transform them, neural networks require such features to be transformed or 'encoded' into numerical values.

A popular method of encoding categorical features in neural networks is one-hot encoding. This involves converting a single categorical feature with c categories into c binary features. These features each take the value of 0 or 1, indicating the category to which each observation in the dataset belongs. An explanation and example of one-hot encoding in Python are provided in Video 5.9.

Video 5.9 – One-hot encoding

https://www.youtube.com/watch?v=agvfUvUNI4A (7 mins)

Record your video notes here

Feature Scaling

Neural networks tend to experience better performance if all the features are on a similar scale. Feature scaling should generally be performed as best practice, but should *always* be applied when regularisation is used as a method to avoid overfitting.

The most common way to convert all features to a similar scale is known as standardisation. This involves converting all features to have a mean of 0 and a standard deviation of 1, as follows:

Equation 5.16

$$x^*_{ij} = \frac{x_{ij} - \bar{x}_j}{\sigma_j}$$

where x_{ij} is the original value of the j^{th} feature for the i^{th} observation, x^*_{ij} is the standardised value, $\bar{x}_j$ is the average of all observations for the j^{th} feature and σ_j is the standard deviation of the j^{th} feature.

Feature scaling should be performed after data cleaning has taken place, as large outliers can skew the calculation of the mean and standard deviation.

Feature scaling is a necessity when model regularisation is being applied in a neural network model.

Granularity Requirements

As part of the control cycle approach to solving problems, it is important to have an upfront understanding of the model's requirements before you start building it. One such consideration is the level of granularity that will be required of the model and whether it is possible to achieve such a level of granularity based on the available data.

For example, in an image-classification problem, the task might be to classify car images. Using a neural network to work out whether there is a car in a photo might be a relatively straightforward problem to solve. However, determining the make and model of the car will be a more difficult problem. The level of data available will help determine whether this level of granularity is possible. In addition, understanding why such a level of granularity is being requested can help you determine whether the extra time to build and train such a complex model is warranted given its end-use. When determining a possible level of granularity based on the data available, a good litmus test is to ask 'can a human expert make a reasonable prediction with this available information?'

Testing

As outlined in various sections throughout this chapter, it is important to test the performance of a trained neural network. This should be done on independent test data that has not been used to build the model.

In addition to looking at high-level model metrics such as those outlined in Section 5.2, you should test your model against specific examples to understand how it performs in different scenarios. For example, if you have built a model to classify images of dogs and cats, you could experiment with your model to see how it responds to images that:

- contain dogs and cats with different rotation, lighting, focus or greyscale values;
- combine the right side of a dog and the left side of a cat; or
- do not contain dogs or cats.

For an example of what can go wrong when a model is not adequately tested, revisit the answer to Exercise 3.12 in Chapter 3.

Deployment

Neural networks are a very powerful classification and prediction tool used in a wide range of applications today. However, like any such tool, they are open to misuse, whether accidental or malicious. Video 5.10 discusses some important ethical considerations when deploying a neural network in practice.

Video 5.10 – The ethics of deep learning

https://www.youtube.com/watch?v=-Emyb507X9Y (11 mins)

Record your video notes here

Many of the ethical issues in using neural networks that are discussed in Video 5.10 are exacerbated by the 'black box' nature of neural networks. This means that it is often difficult for people either within a business or outside the business to really understand what such a model is doing.

As discussed in Chapter 3 (Security, privacy and ethics), the more explainable a model is, the more likely it is that people will trust it and choose to rely on its results. This creates a challenge for those who work with neural networks to find effective ways to explain what these models are doing. In addition, the more explainable a model is, the less likely it is that any biases within the model will go unnoticed and uncorrected.

As outlined in Section 5.3.3, techniques such as LIME and SHAP can be used to interpret the predictions of a neural network.

5.5.6. Neural Networks for Regression

This chapter has focused on the use of neural networks for solving classification problems. Neural networks can also be a powerful technique for solving regression problems.

In a regression setting, the most common neural network architecture involves a single continuous response (output) neuron. Typically, all hidden layers will use the ReLU activation function and then, for the final response neuron:

- the sigmoid function can be used to create a response that takes values between 0 and 1, similar to classification;
- an identity function can be used for regression predictions; or
- an exponential function can also be used to make non-negative predictions, similar to the use of a log-link function in generalised linear modelling.

In a regression setting, a loss function such as squared loss can be adopted rather than the log loss function that is described in Section 5.5.4.

5.5.7. Neural Networks Compared to Other Classifiers

This section discusses how neural network models perform and how they compare to the broader ecosystem of potential classification models. These relative strengths and limitations of neural networks are summarised in Table 5.11 and discussed further below.

Table 5.11 – Strengths and limitations of neural networks

Strengths	Limitations
• Able to design custom network shapes for specific purposes • Naturally scale to multiclass responses • Apply to a wide variety of modelling contexts • Can be very accurate • Provide efficient predictions • Updateable • Good free software alternatives exist	• Model fitting can be both data and computationally intensive • Interpretability issues • Randomness associated with model fitting

Neural networks allow you to **customise the design** of a model quite heavily. As will be shown in Section 5.7.1, certain modifications to standard neural networks have made these models powerful solutions to a variety of new learning problems.

Part of the flexibility of neural networks relates to the output layer. Whereas many classification approaches are geared towards binary classification, neural networks can **scale to many classes.** For example, image classification tasks often have hundreds of categories to which images can be allocated.

The flexibility of neural networks makes them **applicable in a wide variety of modelling contexts.** This is seen in the types of input data that can be processed by neural networks, including text, image, video and audio data.

Neural networks can be **very accurate**. When they are appropriately tuned, they are competitive with most high-performing predictive modelling approaches, such as boosted decision trees or other ensemble models.

One difference between neural networks and boosted decision trees is that a neural network is parametric. This means that the number of parameters is specified by the user and does not grow with the size of the dataset. Conversely, tree-based models are nonparametric, and their final form will grow as more data is added and the additional detail is modelled. This parametric nature can make neural networks very **efficient when making predictions**. For example, when neural networks are used as a digital voice assistant on a phone, they can be more efficient than other models in processing and scoring new data points.

Neural networks are also **updateable**. Rather than refitting a network from scratch, an existing model can be updated with new data using a technique like backpropagation. This is useful as new data is collected over time, and neural networks can be implemented in an ‘online’ setting where such updating occurs automatically. This updateability is useful for ‘transfer’ learning, where a model is repurposed. For example, a general-purpose image-recognition model can be adapted to identify a specific type of object, which is far more efficient than building a new model from scratch. Transfer learning is discussed further in Section 5.7.2.

Finding good software to fit neural networks used to be a disadvantage of the approach, but the last few decades have seen **heavy development of commercial-grade software packages**, many of which are freely available, such as TensorFlow and PyTorch. These are heavily optimised to fit models efficiently, even more so when they are run on graphic-card processing units (GPUs) or other specialised hardware.

That said, modern neural networks are still **slow to fit** their models, due to the complexity of the models and the large size of the datasets that are typically used to train them.

Interpretability issues are also a challenge. It is hard to 'see' what a neural network model is doing if it has many layers and neurons. However, general-purpose model-interpretation tools allow some insights to be gathered. For example, analysis can be taken to understand relative variable importance or to attribute outputs to various features.

Exercise 5.20

Go to the TensorFlow playground (https://playground.tensorflow.org/) and experiment with different options for layers, hidden neurons, activation functions, learning rates and other factors. The tool allows you to 'see' the output at individual neurons along the learning space. Can you see how the neural network uses weights and activation functions to create nonlinear decision rules?

Explain why performance is poor on the 'circle' data using linear activation functions and only $X_{\cdot 1}$ and $X_{\cdot 2}$ as inputs.

Finally, there is **randomness associated with fitting neural networks**. For large networks, there are tuning challenges to ensure that a reasonable minimum for the loss function is found and models fit using different initial parameters can perform materially differently. This contrasts to an approach like logistic regression, which will produce the same model parameters on the same input data every time.

5.6. Case Studies

5.6.1. Case Study 1—Customer Churn

Define the Problem

Customer churn, also known as customer attrition, customer turnover or customer defection, is the loss of clients or customers.[16] For many businesses, a high level of customer churn can negatively impact their profits, particularly because it is often quite costly for a business to acquire new customers.

For this reason, many businesses like to understand which customers are likely to churn in a given period. Armed with this information, businesses can employ different strategies to try to retain their customers.

This case study investigates the use of neural networks and gradient boosting machines for predicting which customers are likely to churn.

When trying to predict customer churn, it may seem like a relatively straightforward task to obtain some past customer data and use this to determine whether future customers will churn. However, as you will see in Video 5.11, the task of deciding *exactly* what the output of such a prediction model should be is quite complex, and heavily dependent on how the model will be used by the business.

Video 5.11 – Defining a customer churn prediction model

https://www.youtube.com/watch?v=kE_t3Mm8Z50&feature=share (33 mins)

Record your video notes here

[16] Wikipedia. Customer attrition. https://en.wikipedia.org/wiki/Customer_attrition

Design a Solution

The notebook 'DSA_C05_CS1.ipynb' builds several customer churn prediction models using a dataset from a Kaggle competition[17] that aimed to predict customer churn behaviour for a telecommunications provider. This dataset contains 7,043 rows (one for each customer) and 19 features, including information about each customer's:

- services with the company, such as phone, internet, online security, online backup, device protection, tech support, and streaming of TV and movies;
- account information, such as how long they have been a customer, contract, payment method, paperless billing, monthly charges and total charges; and
- demographic information, such as gender, age range, and whether they have a partner and dependants.

The response variable in this dataset is labelled 'Churn'. It represents whether each customer left the service provider in the month preceding the data extract date.

You should now explore the notebook 'DSA_C05_CS1.ipynb' through each of its stages of importing, exploring and preparing the data. Then use it to build the following classification models:

- a gradient boosting machine;
- a simple neural network with one hidden layer; and
- a more complex neural network.

5.6.2. Case Study 2—Digit Recognition

Define the Problem

Another popular use of neural networks is in image detection. Examples of ways in which neural network image classifiers might be used to solve business problems include:

- analysing drone images to make the insurance claims management process more efficient, such as by classifying the level of damage caused to a property following a natural disaster; and
- extracting information from handwritten correspondence from customers, such as information on insurance claims forms to be entered into the claims database.

[17] https://www.kaggle.com/blastchar/telco-customer-churn

This case study investigates the use of a neural network to decipher handwritten digits (from zero to nine). This might be a useful tool if, for example, you are trying to automate the process of sorting mail based on postcodes written on the front of envelopes. Of course, the task of training a neural network to recognise handwritten digits can also be extended to recognising handwritten letters and then words.

Design a Solution

The dataset used in this case study is a famous Modified National Institute of Standards and Technology (MNIST) dataset of handwritten images (http://yann.lecun.com/exdb/mnist/). The MNIST dataset is popular for use in benchmarking classification algorithms.

The dataset has 42,000 observations, each representing a greyscale image of a hand-drawn digit from zero to nine. Each image is 28 pixels in height and 28 pixels in width, making a total of 784 pixels (28x28). Each pixel has a single pixel value associated with it, from 0 to 255, indicating the lightness or darkness of that pixel. Higher pixel values represent darker pixels. The dataset represents these images as 784 features, with each feature representing a different pixel in the image.

The dataset also contains one response ('label') that takes integer values from zero to nine, indicating the digit drawn in each image.

You should now explore the notebook ‘DSA_C05_CS2.ipynb’ through each of its stages of importing, exploring and preparing the data, and then use it to build an image classifier using a standard (‘vanilla’) feed-forward neural network and a convolutional neural network. Section 5.7.1 contains information about convolutional neural networks and how these differ from the standard feed-forward neural networks that have been described in this chapter.

5.7. Further Neural Network Concepts

5.7.1. Extensions to Standard Neural Networks

The flexibility of the neural network structure means that new ideas are constantly being developed and applied to specific contexts. While the detail of such models is beyond the scope of this subject, it is relevant to look at the intuition behind some of these models.

Therefore, this section provides a brief overview of the following extensions to standard feed-forward neural networks:

- convolutional neural networks;
- deep learning neural networks;
- recurrent neural networks;
- long/short-term memory networks;
- autoencoder networks;
- generative adversarial networks;
- transformer networks; and
- diffusion models.

Convolutional neural networks (CNNs) are quite different from most other neural networks. They are an important tool in image analysis as a way of detecting patterns across different areas of an image. Video 5.12 provides an overview of the intuition behind CNNs.

Video 5.12 – Convolutional neural networks

https://www.youtube.com/watch?v=YRhxdVk_sIs (7 mins)

Record your video notes here

As described in Video 5.12, a convolutional layer in a neural network detects particular patterns of pixels (e.g. a convolutional layer might apply an edge detector to every 16x16 grid of pixels in an image). Also, CNNs often include 'pooling' layers, where results from a pattern detection step can be translated or compressed to reduced dimensionality.

A key aspect of CNNs is that the links between layers are strategically chosen to pick up small segments of an image, and the 'detector' weights are used consistently across these segments.

Deep learning networks (DLNs) are neural networks with very many hidden layers that allow significantly better performance on complex problems, given enough data. One important class of deep learning is deep convolutional neural networks. These build on convolutional network patterns to give strong results across image, video and audio processing.

Recurrent neural networks (RNNs) are useful for data that contains a natural order to its observations (e.g. a time series, or words on a page). Neurons in such a network contain normal links across its layers, but they also contain links to the equivalent neuron across time (i.e. the state of the neuron for the previous observation). This creates a type of memory feature in the network. For instance, a neuron reading text can remember seeing an open bracket and will be on the lookout for a closed bracket.

Long/short-term memory networks (LSTMs) extend recurrent neural network ideas by adding a more formal 'memory cell' structure to the recurrent neurons. These have been found to be very effective in understanding longer, more complex sequences of events, such as those necessary to learn long texts or music.

Autoencoder networks are usually used for unsupervised learning. They are more like a different use of feed-forward networks than a fundamentally different architecture. The basic idea behind autoencoders is to encode or compress information automatically. In other words, they attempt to reproduce the input layer in the output layer, but with smaller middle layers that result in the network having to determine the most important parts of the input to retain. Autoencoder networks can be used to identify the most important components of an input relative to the overall performance function defined by the user.

Generative adversarial networks (GANs) are quite different from standard neural networks. They are made up of two networks:

- generative—this network generates content, such as an image of a face, based on what it has learned from the input data; and
- discriminating—this network judges content by attempting to discriminate between real and synthetic content (e.g. tell which faces are original and which have been generated by a model).

The discriminating network receives either training data or generated content from the generative network.

As both networks improve on training, the generative network has to consistently improve its performance to confuse the discriminating network until both models eventually stabilise. Generative adversarial networks are popular for a range of applications.

Exercise 5.21

Read this article on identifying fake faces generated using generative adversarial networks: https://kcimc.medium.com/how-to-recognize-fake-ai-generated-images-4d1f6f9a2842.

Outline how some of the tell-tale signs that distinguish a generated image from a real image relate to the underlying way the images are generated.

Transformer networks (or 'transformers') are the cornerstone of recent innovations in large language models for text generation and translation (the 'T' in ChatGPT refers to transformer). Transformers use 'attention' layers to identify important components of the input, such as the key words in a sentence. These attention layers enable improved performance and scalability for text compared to the RNN and LSTM approaches. Large language models are discussed further in Chapter 7.s

Diffusion models have been used in models such as Stable Diffusion and DALL-E 3 for image generation from text. Unlike GANs which are trained against a discriminating model, a diffusion model is trained against data with noise deliberately added. The diffusion model attempts to reconstruct the original image from the noisy data.

Exercise 5.22

Read the following article: https://www.asimovinstitute.org/neural-network-zoo/.

Select one network structure from the chart of neural network designs for further research to understand why the design is well-suited to certain types of problems.

5.7.2. Neural Networks—Other Considerations

Off-The-Shelf Models and APIs

There has been a proliferation of tools to support neural network building in recent years, particularly from the major tech companies. This means that building a neural network model from first principles is rarely necessary. Rather, it is possible to use:

- an existing neural network package that simplifies the process of specifying and fitting a model (e.g. the TensorFlow package in Python);
- an existing off-the-shelf model that is adapted to a specific case ('transfer learning'); and
- an existing off-the-shelf model hosted elsewhere and delivered over an API, which allows you to call the existing model (often for a fee) to score it on your data.

An example of transfer learning is using a neural network that has been trained to recognise car images and adapting this for use in a traffic monitoring context.

Off-the-shelf models that are hosted in the cloud effectively outsource the implementation of a model, so all that remains is to integrate this model into a specific purpose.

If using an outsourced model, it is still your responsibility to understand what the model is doing, and provide some assessment of the model's quality. All models have limitations, and some may fail in specific but important circumstances, so model governance remains important. In addition, you may need to consider data governance, security and privacy principles as discussed in Chapter 3 (Security, privacy and ethics).

Practical Failings

Any emerging technology brings risks, and neural networks are no different. Some examples of the risks of using neural networks are as follows:

- reverse engineering;
- unintended classifications;
- model bias; and
- spurious learning.

One risk of using neural networks is the possibility of **reverse engineering** the model to make a detector work in unintended ways. Video 5.13 provides an example where projected street signs from a drone were used to confuse the autopilot feature on a Tesla car. More generally, there are many examples where image classifiers can be fooled by unusual patterns that typically would not confuse a human interpreter.

Video 5.13 – Confusing the Tesla autopilot

https://www.youtube.com/watch?v=C-JxNHKqgtk (1 mins)

Record your video notes here

Another risk is **unintended classifications** that lead to reputational damage. For instance, Google's image-recognition software that was integrated into the Photos app classified some people as 'gorillas' by mistake. The ethical implications of this misclassification were discussed in Chapter 3 (Security, privacy and ethics). Ultimately, Google removed the classification category entirely to remove this unintended outcome of the model.[18]

Model bias is a broad topic that is common to many modelling problems. In some cases, model bias might arise because it reinforces existing biases in the training data. For instance, Amazon attempted to build a model to streamline its hiring process. They found that the model consistently gave negative ratings to terms associated with women.[19] This model bias was likely due to the historical hiring process being biased towards men. This bias proved difficult to undo in the Amazon model.

[18] https://www.theverge.com/2018/1/12/16882408/google-racist-gorillas-photo-recognition-algorithm-ai
[19] https://www.reuters.com/article/us-amazon-com-jobs-automation-insight-idUSKCN1MK08G

Another potential issue for neural networks is spurious learning. This occurs when a classifier predicts well, but only due to features that do not generalise to future cases. For instance, researcher Thomas Dietterich recounts one incidence of spurious learning:[20]

> *We photographed 54 classes of insects. Specimens had been collected, identified, and placed in vials. Vials were placed in boxes sorted by class. I hired student workers to photograph the specimens. Naturally they did this one box at a time; hence, one class at a time. Photos were taken in alcohol. Bubbles would form in the alcohol. Different bubbles on different days. The learned classifier was surprisingly good. But a saliency map revealed that it was reading the bubble patterns and ignoring the specimens. I was so embarrassed that I had made the oldest mistake in the book (even if it was apocryphal). Unbelievable. Lesson: always randomize even if you don't know what you are controlling for!*

In this case, the model had found an efficient way to detect the day the photo had been taken, which correlated with the response.

Finally, Chapter 3 (Security, privacy and ethics) introduced issues of governance around data science models more broadly, and these are all equally relevant to neural network models.

[20] https://www.gwern.net/Tanks

5.8. Key Learning Points

- Classification is a supervised learning method that predicts discrete or categorical responses.
- Statistical measures that help to assess a classifier's performance include loss functions, confusion matrices, the ROC curve, and validation and generalisation error.
- Logistic regression is a generalised linear model with a logit link function. It is relatively fast, robust, and often produces reasonable prediction accuracy but is not naturally suited to modelling nonlinear relationships in the data.
- Decision trees provide logic rules set out in a flowchart-like structure. They handle different types of features well, scale well to large datasets and are interpretable. However, they are sometimes less accurate than other classifiers.
- Ensemble models, such as bagging, boosting, and stacking, improve prediction accuracy by combining small sub-models.
- When model building, a trade-off usually occurs between model bias and variance.
- Gradient descent is a common technique for minimising a model's loss function, particularly when closed-form solutions do not exist.
- Neural networks can be used to model complex nonlinear relationships.
- A neural network is made up of three main types of neurons: original predictor, derived predictor, and response neurons.
- Neural networks effectively automate the feature engineering process, without explicit instruction from a human, via the derived predictor neurons.
- In a feed-forward neural network, derived predictor and response neurons are estimated by applying an activation function to a weighted linear combination of the neurons in the preceding layer.
- Neural networks are fitted to data via repeated application of the forward propagation and backpropagation processes.
- Neural networks can be very accurate; they naturally scale to multiclass responses and can make efficient predictions. However, the model fitting process can be data and computationally intensive and they can suffer from a lack of transparency.
- There are many extensions to neural networks such as convolutional, deep learning, recurrent, long/short-term memory, autoencoder, generative adversarial networks, transformer networks, and diffusion models.

5.9. Answers to Exercises

Answer to Exercise 5.1

Task	Response variable	Model type
Using the intensity of gene expressions to predict the type of cancer a person is suffering	Type of cancer	Classification
Using suburb and house characteristics to predict the market value of a home	Market value	Regression (unless the market value is predicted in value 'buckets')
Using customer characteristics to predict whether they will click on an internet advertisement	Will they click on ad: Yes or no	Classification
Using car sensor data to identify whether a pedestrian is crossing the road ahead	Is a pedestrian crossing: Yes or no	Classification

Answer to Exercise 5.2

In this production line, the probability that a widget is faulty is:

- 30% on a weekday: P(faulty|weekday) = 30%, P(not faulty|weekday) = 70%; and
- 60% on a weekend: P(faulty|weekend) = 60%, P(not faulty|weekend) = 40%.

Therefore, a Bayes classifier will predict that a widget is:

- not faulty if produced on a weekday, as P(not faulty|weekday) > P(faulty|weekday); and
- faulty if produced on a weekend, as P(faulty|weekend) > P(not faulty|weekend).

This classifier is expected to be wrong:

- when widgets are faulty on a weekday (30% of the time on these days); and
- when widgets are not faulty on a weekend (40% of the time on these days).

Answer to Exercise 5.2

Therefore, the expected error rate for a Bayes classifier (the Bayes error rate) is:

30% x $\frac{5}{7}$ + 40% x $\frac{2}{7}$ = 32.9%.

Answer to Exercise 5.3

Other examples of classification where having different costs across classes might be appropriate include:

- classification models that try to predict whether someone is guilty of a crime—it may be more important to ensure that innocent people are not punished than it is to ensure all guilty people are punished;
- during a pandemic, a predictive model might be used to screen passengers arriving from overseas to see if they should be quarantined. Due to the severity of the disease, it may be more important to quarantine every person who is flagged as likely to have the disease, even though this may result in many false positives; and
- health testing where three or more diagnoses are possible—in this case, the potential diagnoses might be weighted according to how life-threatening they are, as it might be more important that all potential cancers are investigated than it is to correctly diagnose all cases of less life-threatening conditions such as a benign growth or skin infection.

Answer to Exercise 5.4

Without cost information:

The confusion matrices and their associated measures of success for Model A and Model B are shown below.

Model A

True class	Predicted class: No infection	Predicted class: Infection	Total
No infection	630	50	680
Infection	170	150	320
Total	800	200	1,000

tn	fp
fn	tp

Model B

True class	Predicted class: No infection	Predicted class: Infection	Total
No infection	480	200	680
Infection	70	250	320
Total	550	450	1,000

tn	fp
fn	tp

Measures of 'success':

	Model A numerator	Model A denominator	Model A rate	Model B numerator	Model B denominator	Model B rate	'Best' mode
misclassification rate	220	1,000	22.0%	270	1,000	27.0%	A
accuracy	780	1,000	78.0%	730	1,000	73.0%	A
precision	150	200	75.0%	250	450	55.6%	A
sensitivity/recall	150	320	46.9%	250	320	78.1%	B
false negative rate	170	320	53.1%	70	320	21.9%	B
specificity	630	680	92.6%	480	680	70.6%	A
false positive rate	50	680	7.4%	200	680	29.4%	A

Different measures of success favour Model A and Model B, so at this stage, it is difficult to determine which model is 'better'.

You might use an F1 score or another index to combine some of the above confusion matrix metrics to decide which model, overall, is better. However, this approach fails to consider which measures are most important to the hospital.

Answer to Exercise 5.4

With cost information:

In this example, the misclassification cost of each model is calculated as:

$$misclassification\ cost = (fp \times cost\ of\ false\ positives + fn \times cost\ of\ false\ negatives) \div n$$

where:

- n is the total number of observations;
- $cost\ of\ false\ positives$ is the marginal cost of predicting an infection when no infection exists ($100 = the cost of testing); and
- $cost\ of\ false\ negatives$ is the marginal cost of predicting no infection when an infection exists ($20,000—the difference between the cost of treating an infection early and waiting until later to treat it).

These costs are shown below for each model:

Model A

	Predicted class		
True class	No infection	Infection	**Total**
No infection	630	50	**680**
Infection	170	150	**320**
Total	**800**	**200**	**1,000**

tn	fp
fn	tp

Model B

	Predicted class		
True class	No infection	Infection	**Total**
No infection	480	200	**680**
Infection	70	250	**320**
Total	**550**	**450**	**1,000**

tn	fp
fn	tp

Costs of misclassification

	Predicted class	
True class	No infection	Infection
No infection	$0	$100
Infection	$20,000	$0

The misclassification cost for Model A is $3,405 (50 x $100 + 170 x $20,000) / 1,000.

The misclassification cost for Model B is $1,420 (200 x $100 + 70 x $20,000) / 1,000.

Therefore, based on the misclassification costs set out in this exercise, Model B is better for this hospital than Model A because it results in a lower cost of misclassification.

Answer to Exercise 5.5

The misclassification rate is the number of incorrect classifications as a proportion of the total classifications: $\frac{1}{n}\sum_{i=1}^{n} I\{g_i \neq G(X_i)\}$

Incorrect classifications

= $\sum_{i=1}^{n} I\{g_i \neq G(X_i)\}$

= genuine emails classified as spam + spam classified as genuine emails

= 651 + 361 = 1,012

Total classifications (n) = 4,601

Misclassification rate = 1,012 / 4,601 = 22.0%

Answer to Exercise 5.6

accuracy = $(tn + tp)$ / $(tn + fp + fn + tp)$ = 78.0%

precision = tp / $(tp + fp)$ = 69.0%

sensitivity/recall = tp / $(fn + tp)$ = 1,452 / (361+1,452) = 80.1%

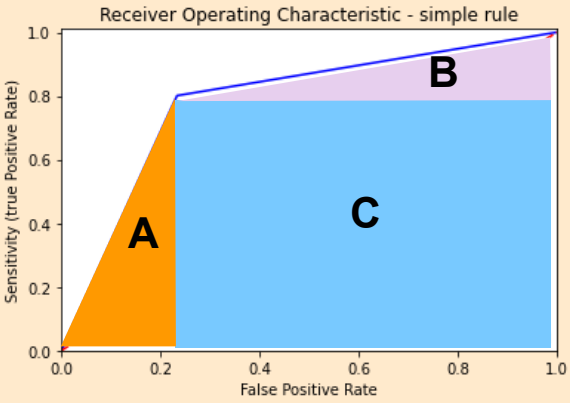
specificity = tn / $(tn + fp)$ = 2,137 / (2,137 + 651) = 76.6%

false negative rate = fn / $(fn + tp)$ = 1 - 80.1% = 19.9%

false positive rate = fp / $(tn + fp)$ = 1 - 76.6% = 23.4%

Answer to Exercise 5.7

The AUC for the ROC curve shown in Figure 5.4 can be manually calculated by dividing the area under the curve into three smaller areas:



- A—the triangle bounded by points (0,0), (0.234, 0.801), and (0.234, 0);
- B—the triangle bounded by points (0.234, 0.801), (1, 1) and (1, 0.801); and
- C—the rectangle bounded by points (0.234, 0), (0.234, 0.801), (1, 0.801) and (1,0).

The area of A = ½ x h x w = ½ x 0.801 x 0.234 = 0.094.

The area of B = ½ x h x w = ½ x (1-0.801) x (1-0.234) = 0.076.

The area of C = h x w = 0.801 x (1-0.234) = 0.614.

Therefore, the AUC is 0.784 (0.094 + 0.076 + 0.614).

Answer to Exercise 5.8

You could calculate the ROC curve by hand for these observations by repeating the following steps for different threshold values:

- choose a threshold value to assign each observation to class 0 or class 1:
  - e.g. a threshold value of 0.5 would assign observation groups A, B and C, which have predicted probabilities less than 0.5, to class 0, and groups D and E, which have predicted probabilities greater than 0.5, to class 1;
- calculate the confusion matrix entries for the chosen threshold value:
  - e.g. for a threshold value of 0.5:
    - true negatives = 12 (4+5+3);
    - false positives = 5 (4+1);
    - false negatives = 6 (2+1+3); and
    - true positives = 17 (7+10);
- calculate the true negative rate and sensitivity (true positive rate):
  - e.g. for a threshold value of 0.5:
    - false positive rate = $fp \,/\, (tn + fp)$ = 5/(12+5) = 29.4%; and
    - sensitivity (true positive rate) = $tp \,/\, (fn + tp)$ = 17/(6+17) = 73.9%; and
- plot the false positive rate and sensitivity for each threshold value.

The following table shows the different values of the confusion matrix entries, the false positive rate and the sensitivity for different threshold values.

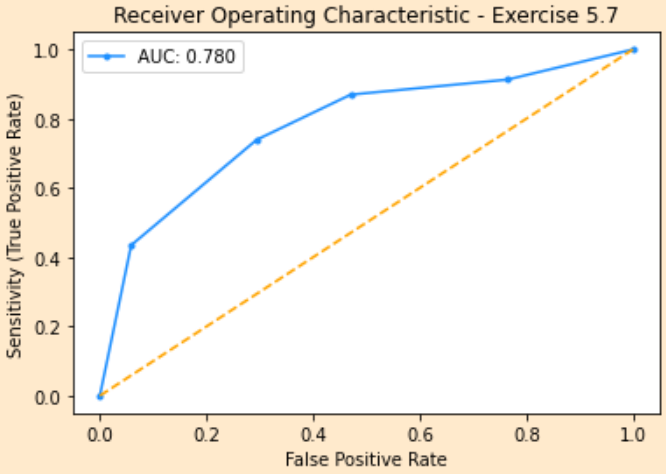
Answer to Exercise 5.8

Threshold	True negative (tn)	False positive (fp)	False negative (fn)	True positive (tp)	False positive rate	Sensitivity (True positive rate)
0.0	0	17	0	23	100.0%	100.0%
0.1	4	13	2	21	76.5%	91.3%
0.2	4	13	2	21	76.5%	91.3%
0.3	9	8	3	20	47.1%	87.0%
0.4	12	5	6	17	29.4%	73.9%
0.5	12	5	6	17	29.4%	73.9%
0.6	16	1	13	10	5.9%	43.5%
0.7	16	1	13	10	5.9%	43.5%
0.8	17	0	23	0	0.0%	0.0%
0.9	17	0	23	0	0.0%	0.0%
1.0	17	0	23	0	0.0%	0.0%

The ROC curve could then be plotted based on the values above and the area under the curve (AUC) calculated by hand, as per Exercise 5.7. The AUC for this curve is 0.780.

Alternatively, the roc_curve and roc_auc_score functions from scikit-learn could be used to answer this question in Python. The notebook ‘DSA_C05_Ex8.ipynb’ contains the code required to produce the following ROC curve and associated AUC:

Answer to Exercise 5.8



Answer to Exercise 5.9

The sum of all the class probabilities can be calculated as:

$$\sum_{k=1}^{K} \hat{p}_{i,g_k}$$

$$= \sum_{k=1}^{K} \frac{e^{LP_k}}{\sum_{d=1}^{K} e^{LP_d}}$$

$$= \frac{e^{LP_1}}{\sum_{d=1}^{K} e^{LP_d}} + \frac{e^{LP_2}}{\sum_{d=1}^{K} e^{LP_d}} + \cdots + \frac{e^{LP_K}}{\sum_{d=1}^{K} e^{LP_d}}$$

$$= \frac{\sum_{k=1}^{K} e^{LP_k}}{\sum_{d=1}^{K} e^{LP_d}}$$

$$= 1$$

When class 1 is used as the reference class, $\hat{p}_{i,g_1} = \frac{e^0}{\sum_{d=1}^{K} e^{LP_d}} = \frac{1}{\sum_{d=1}^{K} e^{LP_d}}$.

Answer to Exercise 5.9

Each of the other class probabilities can be expressed as a factor of $\hat{p}_{i,g_1}$ as follows:

$$\hat{p}_{i,g_k} = \frac{e^{\beta_{0k}+\beta_k^T X_{i.}}}{\sum_{d=1}^{K} e^{LP_d}} = e^{\beta_{0k}+\beta_k^T X_{i.}} \times \hat{p}_{i,g_1}.$$

So, for example, $\hat{p}_{i,g_2} = e^{\beta_{02}+\beta_2^T X_{i.}} \times \hat{p}_{i,g_1}$.

If class 2 becomes the new reference class so that $LP_2 = 0$ and all other linear predictors are redefined as $LP_k(base_2) = LP_k(base_1) - (\beta_{02} + \beta_2^T X_{i.})$, then:

$$\hat{p}_{i,g_2} = \frac{e^0}{\sum_{d=1}^{K} e^{LP_d}} = \frac{1}{\sum_{d=1}^{K} e^{LP_d}} \text{ and}$$

$$\hat{p}_{i,g_k} = \frac{e^{\beta_{0k}+\beta_k^T X_{i.}-(\beta_{02}+\beta_2^T X_{i.})}}{\sum_{d=1}^{K} e^{LP_d}} = e^{(\beta_{0k}-\beta_{02})+(\beta_k^T-\beta_2^T)X_{i.}} \times \hat{p}_{i,g_2}.$$

Therefore, when class 2 is the base, $\hat{p}_{i,g_1} = e^{(\beta_{01}-\beta_{02})+(\beta_1^T-\beta_2^T)X_{i.}} \times \hat{p}_{i,g_2}$.

Rearranging this yields $\hat{p}_{i,g_2} = e^{(\beta_{02}-\beta_{01})+(\beta_2^T-\beta_1^T)X_{i.}} \times \hat{p}_{i,g_1}$.

For any other class g_k,

$$\hat{p}_{i,g_k} = e^{(\beta_{0k}-\beta_{02})+(\beta_k^T-\beta_2^T)X_{i.}} \times \hat{p}_{i,g_2} = e^{(\beta_{0k}-\beta_{02})+(\beta_k^T-\beta_2^T)X_{i.}} \times e^{(\beta_{02}-\beta_{01})+(\beta_2^T-\beta_1^T)X_{i.}} \times \hat{p}_{i,g_1}.$$

Simplifying this expression, $\hat{p}_{i,g_k} = e^{(\beta_{0k}-\beta_{01})+(\beta_k^T-\beta_1^T)X_{i.}} \times \hat{p}_{i,g_1}$.

So when class 1 is the base:

$\hat{p}_{i,g_2} = e^{\beta_{02}+\beta_2^T X_{i.}} \times \hat{p}_{i,g_1}$; and

$\hat{p}_{i,g_k} = e^{\beta_{0k}+\beta_k^T X_{i.}} \times \hat{p}_{i,g_1}$.

When class 2 is the base:

$\hat{p}_{i,g_2} = e^{(\beta_{02}-\beta_{01})+(\beta_2^T-\beta_1^T)X_{i.}} \times \hat{p}_{i,g_1}$; and

$\hat{p}_{i,g_k} = e^{(\beta_{0k}-\beta_{01})+(\beta_k^T-\beta_1^T)X_{i.}} \times \hat{p}_{i,g_1}$.

Answer to Exercise 5.9

These will yield the same results if $\beta_{01} = 0$ and $\beta_1^T = 0$, which fits the initial definition of $LP_1 = \beta_{01} + \beta_1^T X_i = 0$. So the same set of probability predictions can be recovered regardless of which class is chosen as the base class.

Answer to Exercise 5.10

For the three-class classification problem, the table below shows the misclassification, Gini and cross-entropy impurity measures for each terminal node and for the tree:

Impurity measure	Split 1 Q_1	Split 1 Q_2	Split 1 $Q(T)$	Split 2 Q_1	Split 2 Q_2	Split 2 $Q(T)$
Misclassification	0.250	0.250	0.250	0.250	0.250	0.250
Gini	0.406	0.406	0.406	0.375	0.375	0.375
Cross-entropy	0.319	0.319	0.319	0.244	0.244	0.244

For this classification problem, both splits result in the same number of incorrectly classified observations (200), so the overall misclassification error is 0.25 (200/800).

However, Split 2 results in more 'pure' terminal nodes. This is because in terminal node 1, there are no observations found in class 3, and in terminal node 2, there are no observations found in class 2. This is reflected in lower Gini (0.375 v 0.406) and cross-entropy (0.244 v 0.319) impurity scores for Split 2, making Split 2 a preferred split based on the Gini and cross-entropy measures.

For the binary classification problem, the table below shows the misclassification, Gini and cross-entropy impurity measures for each terminal node and for the tree:

Answer to Exercise 5.10

Impurity measure	Split 1 Q_1	Split 1 Q_2	Split 1 $Q(T)$	Split 2 Q_1	Split 2 Q_2	Split 2 $Q(T)$
Misclassification	0.250	0.250	0.250	0.333	0.000	0.250
Gini	0.375	0.375	0.375	0.444	0.000	0.333
Cross-entropy	0.244	0.244	0.244	0.276	0.000	0.207

Just as for the three-class problem, both Split 1 and Split 2 result in 200 observations being misclassified. Therefore, the misclassification measure again treats both splits the same, assigning a misclassification error of 0.25 to each.

For Split 2, terminal node 1 is less 'pure' than terminal node 1 for Split 1. This is reflected in a higher Gini (0.444 v 0.375) and cross-entropy (0.276 v 0.244) score for terminal node 1 in Split 2. Conversely, terminal node 2 for Split 2 is more 'pure' than terminal node 2 in Split 1, as all observations in this node belong to class 1. This results in Gini and cross-entropy scores of 0. The overall weighted impurity scores are lowest for Split 2 using the Gini and cross-entropy measures (0.333 v 0.375 and 0.244 v 0.207), making Split 2 the preferred split based on these measures.

Answer to Exercise 5.11

The following table shows the accuracy, AUC and number of terminal nodes for classifiers built with Gini and cross-entropy criterion.

Criterion	Test accuracy	Test AUC	# Terminal nodes
Gini	0.9123	0.9572	37
Cross-entropy	0.9227	0.9545	88

Changing to an entropy criterion gives similar performance. The accuracy increases slightly to 0.92, but the AUC decreases very slightly to 0.9545. Interestingly, the complexity of the tree is higher under the cross-entropy criterion, with 88 terminal nodes compared to only 37 under the Gini criterion.

The following table shows the accuracy, AUC and number of terminal nodes for different values of α.

α	Test accuracy	Test AUC	# Terminal nodes
0.1	0.7793	0.7439	2
0.01	0.8923	0.8883	6
0.001	0.9123	0.9572	37
0.0001	0.9010	0.9507	94
0.00001	0.9010	0.9528	108

As the value of α decreases, the test accuracy and AUC test increase, as does the complexity of the tree, indicated by the number of its terminal nodes. For values of α shown in the table above, the test accuracy and AUC plateau at around 0.91 and 0.95 respectively. Beyond this point, the extra complexity of the tree does not bring about improvements in either the tree's accuracy or AUC; hence, the original proposed value of 0.001 for α appears to be a reasonable choice.

Answer to Exercise 5.12

A customer segmentation project, where different products are to be marketed to different groups, might be well-suited to a single decision tree since it provides easy-to-interpret segment definitions.

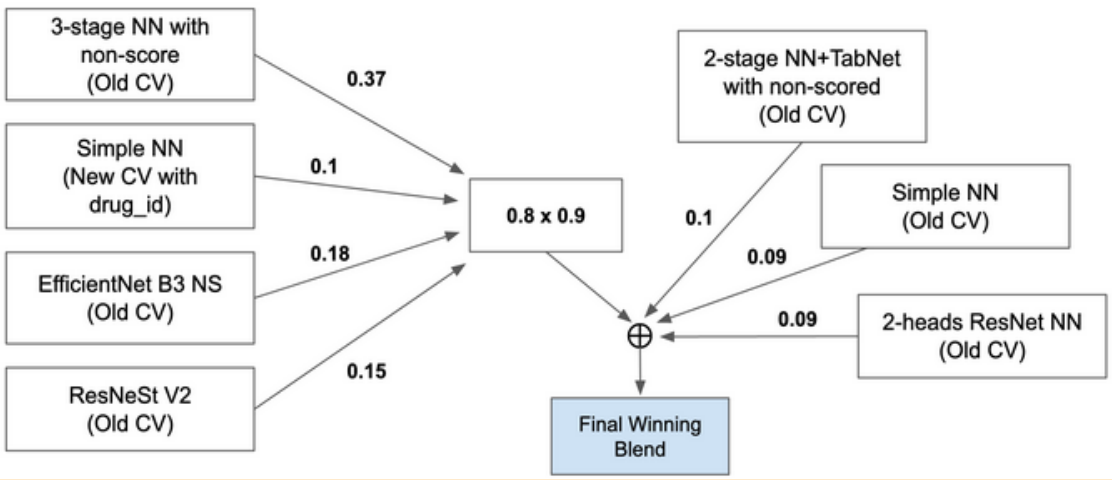
An insurance pricing model with the price expected to vary continuously with the sum insured might be poorly suited to a decision tree since it will produce constant estimates of risk across portions of the sum insured curve.

Answer to Exercise 5.13

Here is a link to the winning solution for a Kaggle competition that challenged participants to 'improve the algorithm that classifies drugs based on their biological activity': https://www.kaggle.com/c/lish-moa/discussion/201510.

The solution description for this competition provided the following visual summary of the seven models that were stacked together to create the winning Kaggle entry:

Figure 1. Winning Weighted-Average Blend.



Answer to Exercise 5.13

The seven models used to create the 'Final Winning Blend' were:

- a 3-stage neural network (NN);
- a 2-stage neural network (NN) and TabNet (a 'tabular deep learning network);
- a simple neural network (NN) with cross-validation that stratified on the multiple classes ('old CV');
- a simple neural network (NN) with a double-stratified cross-validation ('new CV');
- a 2-heads residual neural network (ResNet);
- a DeepInsight EfficientNet B3 NS (a convolutional neural network); and
- a DeepInsight ResNeSt (a variant on a residual neural network).

It is outside the scope of this subject for you to understand all the detail of the above models. However, you can see that they are all neural networks of some form. Therefore, decision trees and decision tree boosting were not part of this winning solution.

An inspection of the source code for this solution, located on GitHub, reveals that the team used Python to create their solution to the competition.

Answer to Exercise 5.14

When fitting a logistic or multinomial regression model, decisions that impact a model's complexity include:

- the number of features from the training dataset that will be used in the model (the more features, the more complex the model);
- the number of new features that will be created by transforming features from the training dataset, such as through the use of polynomial, interaction or spline terms; and
- if regularisation is used, the size of the penalty applied to the weights in the model (the bigger the penalty, the less complex the model will be).

Decisions that impact the complexity of a boosted decision tree model include:

- the number of features used (as for logistic regression above);
- the complexity of each sub-tree in the model;
- the number of sub-trees used in the model; and
- the 'learn' rate that determines the fraction of each incremental tree that is applied to the running estimate of the prediction.

When making decisions that impact the complexity of a model, such as those outlined above, the bias-variance trade-off comes into play because an increase in the model's complexity might reduce the model's bias but will also increase its variance.

Answer to Exercise 5.15

The following observations can be made about each of the activation functions shown in Table 5.8:

- Sigmoid:
  - the resulting neuron will take on a value between 0 and 1;
  - if v is a large negative number, the neuron's value will be close to 0;
  - if v is a large positive number, the neuron's value will be close to 1;
  - the function is insensitive to very large absolute values of v; and
  - the function has a smooth gradient, with no 'jumps' in the output values.
- Radial basis:
  - the resulting neuron will take on any value between 0 and 1;
  - the function is insensitive to very large absolute values of v; and
  - the function has a smooth gradient, with no 'jumps' in the output values.
- Identity:
  - the resulting neuron is unbounded in value;
  - the function is sensitive to very large absolute values of v; and
  - the function has a constant gradient, with no 'jumps' in the output values.
- ReLU:
  - the resulting neuron can take on any value greater than or equal to 0;
  - the function is sensitive to very large positive values of v;
  - the function returns 0 for all negative values of v; and
  - the function has a smooth gradient, with no 'jumps' in the output values.
- Softmax:
  - the resulting neuron will take on a value between 0 and 1;
  - the sum of all neurons in a layer estimated with the Softmax function is 1; and
  - the value of each neuron in the layer is dependent on the value of each of the other neurons in the layer.

Answer to Exercise 5.16

The use of the Softmax function ensures that all neurons in the output layer sum to 1. In a multiclass classification setting, where you are trying to classify observations into three or more classes, the Softmax function can be used to represent the probability that each data point belongs to each class in the model.

For example, you might be using a neural network to classify handwritten digits into 10 classes that represent the 10 digits from 0 to 9. Each observation that is fed into the neural network can be assigned to the class corresponding to the output layer neuron with the greatest value.

Answer to Exercise 5.17

A binary classification setup with no hidden layers and the sigmoid function used in the output layer is logistic regression (see Section 5.3.1).

Answer to Exercise 5.18

It might be undesirable to have a model with a loss function of 0, or even one that is too close to 0, as this is a likely indicator that the model has been overfit to the training data. In this case, the model is likely to have captured too much noise in the training data and is unlikely to generalise well to test data or unseen data on which the model will be scored.

In addition, depending on the context, the cost of refining a model so that the loss function is close to 0 may be greater than the benefit to the end-user of having such a precise model. It may also require an overly complex neural network design that takes too long to score on real data.

Answer to Exercise 5.19

The following table shows the validation accuracy that was achieved for various choices for the neural network's architecture.[21]

Hidden layers	Neurons in each hidden layer	Activation function in hidden layer(s)	Validation accuracy after 100 epochs	Epochs before validation accuracy plateaued
2	16,12	ReLU	0.9531	32
2	16,12	sigmoid	0.9418	20
2	4,4	ReLU	0.9418	33
1	12	ReLU	0.9479	39
3	16,12,12	ReLU	0.9383	12

In this example, the validation accuracy is very high (>93%) under all the architectures. However, the lower validation accuracy when 3 hidden layers were used suggests that this last model might be overfitted to the training data; hence, a less complex neural network might be better.

Answer to Exercise 5.20

The circle data requires a nonlinear function of $X_{.1}$ and $X_{.2}$ to produce good predictions. Therefore, a linear activation function will mean that no matter how many neurons are added to the network's hidden layers, the final output will still be a linear function of the inputs and will, therefore, predict the circle data poorly.

[21] Note that neural network algorithms are stochastic. This means that they make use of randomness, such as initialising to random weights. Therefore, the same network trained on the same data can produce different results, so the results that you get when you run this code will be slightly different to those reported in the table above.

Answer to Exercise 5.21

Memory across the entire image is difficult for the models, so features such as earrings on both ears and other asymmetries require significant sophistication by the model. Similarly, the model is tuned to producing faces, so it is not surprising that background images or text are not rendered as accurately.

Answer to Exercise 5.22

Answer not provided.

Actuaries Institute.

About the Actuaries Institute

The Actuaries Institute is the sole professional body for actuaries in Australia. The Institute provides expert comment on public policy issues where there is uncertainty of future financial outcomes. Actuaries have a reputation for a high level of technical financial skills and integrity. They apply their risk management expertise to allocate capital efficiently, identify and mitigate emerging risks and to help maintain system integrity across multiple segments of the financial and other sectors. This expertise enables the profession to comment on a wide range of issues including life insurance, health insurance, general insurance, climate change, retirement income policy, enterprise risk and prudential regulation, finance and investment and health financing.

Published December 2024

Institute of Actuaries of Australia

ABN 69 000 423 656
Level 2, 50 Carrington Street,
Sydney NSW 2000, Australia
t +61 (0) 2 9239 6100
f +61 (0) 2 9239 6170
actuaries@actuaries.asn.au
www.actuaries.asn.au