

MATH3431: Machine learning and neural networks

John Paul Gosling, edited by Cuong Nguyen

2026-10-18

Contents

Preface	5
Extra reading	7
1 Preliminaries	9
1.1 Data matrices	9
1.2 Distances between individuals	12
1.3 Linear regression	18
1.4 Linear discriminant analysis	25
2 Fundamental concepts	31
2.1 Types of learning	31
2.2 Training basics	32
2.3 Model performance	34
2.4 Cross-validation	52
2.5 Missing or few data	54
3 Supervised Learning — Classification	57
3.1 Historical notes	57
3.2 Running examples	57
3.3 k -nearest neighbours	59
3.4 Naive Bayes classifier	62
3.5 Decision trees	65
3.6 Perceptron	69
3.7 Balancing classes	74
4 Supervised Learning — Regression	77
4.1 Historical notes	77
4.2 Running example	77
4.3 Regularised Regression	79
4.4 k -nearest neighbours	87
4.5 Decision trees	89
4.6 Multivariate Adaptive Regression Splines	90
5 Ensemble methods	95
5.1 Weak and strong learners	96
5.2 Stacking	96

5.3	Bagging	101
5.4	Boosting	109
5.5	Error-Correcting Output Codes	113
5.6	Other ensemble methods	116
6	Model Interpretation	119
6.1	Variable importance	119
6.2	Main effect visualisations	127
7	Unsupervised learning	133
7.1	Traditional statistical approaches	133
7.2	Modern unsupervised learning techniques	141
7.3	Other ideas in unsupervised learning	146

Preface

This is a module of two halves. The first half contains a brief review of some basic ideas in statistics that will lead us on to key concepts in machine learning (ML). We will be focusing on various common algorithms, when they are useful and how to implement them in R. The second half, in Epiphany term, will be a more detailed look at the mechanics of optimisation and its importance to machine learning and moving onto neural networks and other more advanced techniques.

By the end of this term, I want you to be able to:

1. Understand some common terms in machine learning.
2. Be able to use R to utilise some of the most common machine learning techniques.
3. Be able to evaluate the performance of a machine learning model.
4. Realise that there are many difficult questions that arise in practice, and these are lurking behind the scenes in their R implementations.

I have chosen to deliver the material in four ways: this complete set of lecture notes, handwritten notes in lectures to highlight key points, lecture slides to give more realistic examples and the four practical sessions where you will be able to implement the techniques in R. This term will be focussed on machine learning techniques in general and will not be specifically looking at neural networks.

The general structure for this half of the course will be as follows:

1. key concepts in statistics for ML,
2. fundamental ML concepts,
3. classification techniques,
4. regression techniques,
5. making use of multiple models,
6. making ML models more interpretable,
7. unsupervised learning.

This half of the course will be supplemented by four assignments and four practical sessions. For the R elements, you can use your own installations of R or utilise the GitHub Codespace version that I have set up especially for this course. The latter will allow you to run R code in the cloud without having to install anything on your own machine. A link to the Codespace set-up page is given on the Ultra Blackboard site.

Formal, summative assessment has two elements: there will be a practical exam after the second half of the course and a written exam at the end of the academic year. The practical exam will be a test of your ability to implement the techniques in R and to evaluate the performance of a model. The written exam will be a test of your understanding of the theory behind the techniques.

Extra reading

There are many good books on machine learning and neural networks. I have listed a few below that I think are particularly useful (although many go far beyond the scope of this module). As we go through the various topics, I will also point you to some useful resources that will help you to understand the concepts in more depth.

- **Pattern Recognition and Machine Learning** by Christopher M. Bishop. This classic text is renowned for its rigorous mathematical and Bayesian approach to machine learning. It covers topics from probability theory to neural networks.
- **The Elements of Statistical Learning: Data Mining, Inference and Prediction** by Trevor Hastie, Robert Tibshirani and Jerome Friedman. This book is a cornerstone text that delves into the statistical and algorithmic aspects of machine learning.
- **An Introduction to Statistical Learning: with Applications in R** by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani. A more accessible counterpart to *The Elements of Statistical Learning*, this book focuses on applying statistical learning methods using the R programming language. It is ideal for bridging the gap between theory and practical application and includes numerous examples and labs.
- **Machine Learning: A Probabilistic Perspective** by Kevin P. Murphy. This is another comprehensive and mathematically-driven book that treats machine learning from a probabilistic viewpoint.

I would also recommend that you consider how large language models (LLMs) such as ChatGPT and Gemini can be used to help with your studies and to tailor resources to your needs. That is, however, a big topic that I will not be covering.

Chapter 1

Preliminaries

We will start with some building blocks from statistics so we can start to lay out the notation and ideas that we will use throughout the course.

1.1 Data matrices

We make multiple measurements (p variables, say) on each of a set of individuals (n , say). Let x_{ri} be the measurement of the i th variable on the r th individual ($i = 1, \dots, p, r = 1, \dots, n$). In general, i, j, k will index the variables and r, s, t will index the individuals.

Arrange the data as an $n \times p$ matrix $X = (x_{ri})$ called the *data matrix*. Each column $\mathbf{x}_{(i)}$ represents a univariate vector of observations on just one variable: the i th. Hence, we have that

$$X = \left(\mathbf{x}_{(1)} \dots \mathbf{x}_{(p)} \right)$$

Each row $\mathbf{x}_r$ represents the collection of measurements on the r th individual.

$$X = \begin{pmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_n^T \end{pmatrix}.$$

We denote the sample mean vector by $\bar{\mathbf{x}} = (\bar{x}_1 \dots \bar{x}_p)^T$ where $\bar{x}_i = \frac{1}{n} \sum_{r=1}^n x_{ri}$ is the sample mean of the i th variable. Similarly, the sample covariance matrix is defined as the symmetric matrix S with elements

$$s_{ij} = \frac{1}{n-1} \sum_{r=1}^n (x_{ri} - \bar{x}_i)(x_{rj} - \bar{x}_j).$$

Note that, we may find that we run into issues of *identifiability* if $p > n$. Informally, this means that we have more parameters to find than we have data points, which causes issues if you think back to the idea of solving a set of linear equations. We will discuss this in more detail later.

We will be using the following notation throughout the course to describe the sets of data that we are working with:

$$\mathcal{Z} = \mathcal{X} \times \mathcal{Y},$$

where $\mathcal{Z}$ is the set of all data points, $\mathcal{X}$ is the set of all input data points (with individual elements usually encoded in a matrix), and $\mathcal{Y}$ is the set of all output data points (with individual elements encoded in a vector). Like anything in mathematics, there are a number of other conventions for when we refer to elements of $\mathcal{X}$ and $\mathcal{Y}$. For instance, we might call them “predictors and responses” or “features and labels” or “independent and dependent variables” instead of “inputs and outputs”.

1.1.1 Summaries of variability

Sometimes, it is useful to summarise the variability in a dataset with a single number. When we have multivariate data, we can move from a variance matrix to single number summaries such as the trace of the variance matrix or the determinant of the variance matrix. The trace is a proxy for the total variability in the data (in an additive sense), while the square root of the determinant is a proxy for the volume of the data cloud (the square of this is called the *generalised* variance).

Here’s a two-dimensional example to illustrate these ideas. We have a random sample drawn from:

$$\begin{aligned} X_1 &\sim N(0, 1), \\ U &\sim N(0, 1), \\ X_2 &\sim 0.5X_1 + U. \end{aligned}$$

We also create a second dataset that is simply both dimensions of the random sample multiplied by 0.5.

The two following plots are scatter plots of the two datasets. The first plot shows the data cloud for the first dataset, while the second plot shows the data cloud for the second dataset. The ellipses are the 90% confidence ellipses for the data clouds.

For the first dataset, we have a total variance of 1.93 and an approximate volume of 0.88. For the second dataset, we have a total variance of 0.48 and an approximate volume of 0.22, which accords with the transformation used.

1.1.2 Linear combinations

Sometimes it is useful to reduce the dimensionality of a data set by taking linear combinations of the variables. Using matrix notation, we may write

$$Y = XA^T$$

where X is our original data matrix and A is the transformation matrix. The columns of A are the linear combinations of the variables.

We can find the sample mean vector of Y as follows:

$$\bar{\mathbf{y}} = \bar{\mathbf{x}}A^T.$$

Similarly, we can find the sample variance matrix of Y as follows:

$$S_Y = AS_XA^T.$$

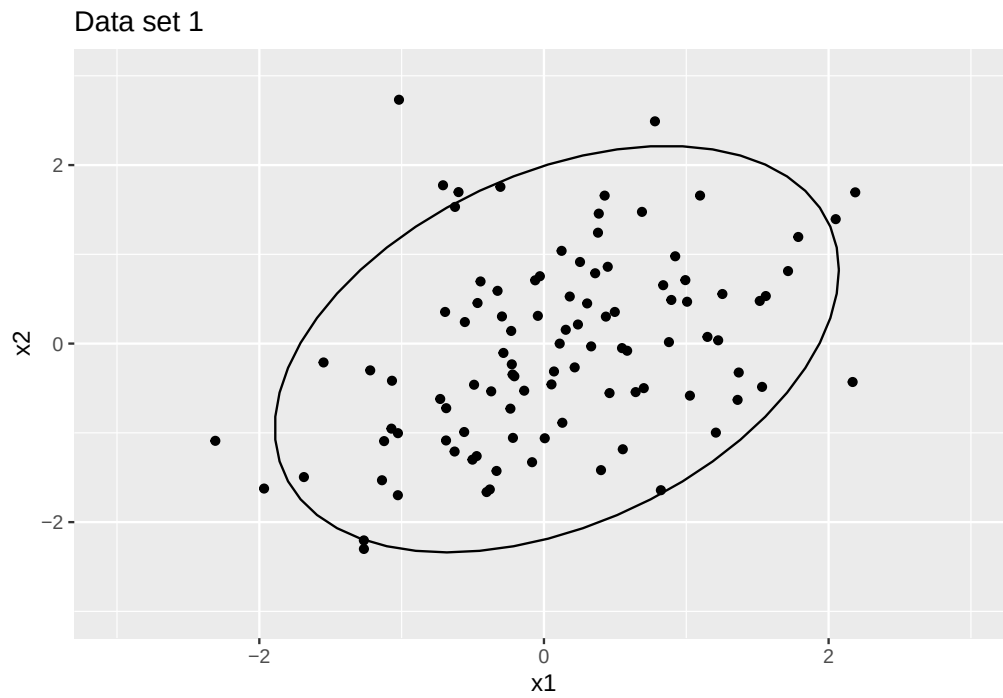


Figure 1.1: Data set 1 with 90% confidence ellipses

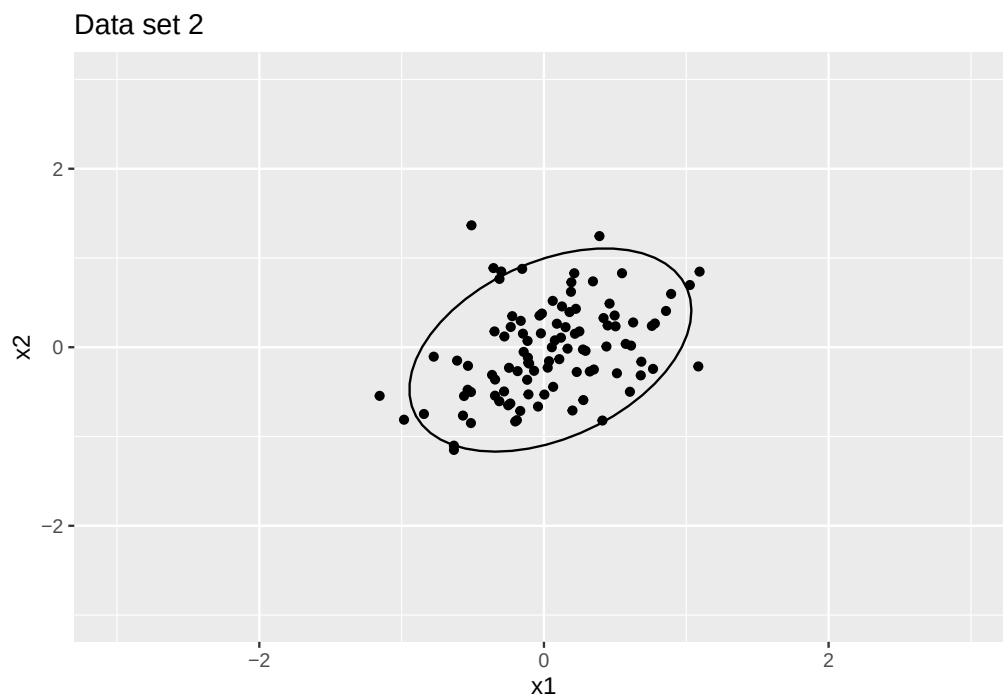


Figure 1.2: Data set 2 with 90% confidence ellipses

1.1.3 Eigen decomposition

The eigenvectors and eigenvalues of a matrix are important because they can be used to decompose the matrix into simpler components. Recall from linear algebra that the eigenvalues and eigenvectors of a matrix S are defined by the equation

$$S\mathbf{v} = \lambda\mathbf{v},$$

where $\mathbf{v}$ is an eigenvector and λ is the corresponding eigenvalue. The eigenvectors of a matrix are orthogonal to each other, and the eigenvalues are real numbers.

Using the definition, we say that the eigen decomposition of a matrix S is given by

$$S = V\Lambda V^T,$$

where V is the matrix of eigenvectors and Λ is the diagonal matrix of corresponding eigenvalues. Loosely speaking, the eigenvectors are the directions in which the data varies the most, while the eigenvalues are the amount of variance in each of these directions.

We then use a linear transformation to rotate the data so that the new variables are uncorrelated. This is useful because it allows us to reduce the dimensionality of the data and to identify the most important variables. We have the following transformation

$$Y = (X - \mathbf{1}_n \bar{\mathbf{x}}^T) V^T.$$

1.2 Distances between individuals

Let $\mathbf{x}_r$ and $\mathbf{x}_s$ be the rows of the data matrix X for individuals r and s . We might want a measure of the “distance” between these individuals. Some possibilities that you might have come across are:

- Euclidean distance,
- Pearson distance,
- Mahalanobis distance,
- Manhattan distance.

A distance measure is a type of dissimilarity measure that satisfies the following properties:

1. Non-negativity: $d(\mathbf{x}_r, \mathbf{x}_s) \geq 0$ for all r and s .
2. Identity: $d(\mathbf{x}_r, \mathbf{x}_s) = 0$ if and only if $\mathbf{x}_r = \mathbf{x}_s$.
3. Symmetry: $d(\mathbf{x}_r, \mathbf{x}_s) = d(\mathbf{x}_s, \mathbf{x}_r)$.
4. Triangle inequality: $d(\mathbf{x}_r, \mathbf{x}_s) + d(\mathbf{x}_s, \mathbf{x}_t) \geq d(\mathbf{x}_r, \mathbf{x}_t)$.

Using these properties, we can broaden the sets of objects that we can define distances between. For example, we can define distances between sets, between distributions and between functions.

Distances are important in machine learning because they are used in many algorithms and in the evaluation of models. The flexibility for us to choose which distance metric to use is important because it allows us to tailor the distance metric to the problem at hand and create richer sets of models.

1.2.1 Euclidean distance

The Euclidean distance between individuals r and s is given by

$$d(\mathbf{x}_r, \mathbf{x}_s) = \sqrt{\sum_{i=1}^p (x_{ri} - x_{si})^2}.$$

Sometimes, I will write this in vector form using “norm” notation:

$$d(\mathbf{x}_r, \mathbf{x}_s) = \|\mathbf{x}_r - \mathbf{x}_s\| = \sqrt{(\mathbf{x}_r - \mathbf{x}_s)^T (\mathbf{x}_r - \mathbf{x}_s)}.$$

1.2.2 Pearson distance

The Pearson distance between individuals r and s is given by

$$d(\mathbf{x}_r, \mathbf{x}_s) = \sqrt{\sum_{i=1}^p \left(\frac{x_{ri} - \bar{x}_i}{\sqrt{s_{ii}}} - \frac{x_{si} - \bar{x}_i}{\sqrt{s_{ii}}} \right)^2}.$$

Therefore, the Pearson distance is the Euclidean distance between the centred and scaled data. The centring is a bit of a red herring because it does not affect the distance. The scaling is important because it ensures that each variable contributes equally to the distance. This is important because if one variable has a much larger variance than the others, it will dominate the distance.

Again, in vector form, we can write this as

$$d(\mathbf{x}_r, \mathbf{x}_s) = \sqrt{\frac{1}{s_{ii}} (\mathbf{x}_r - \mathbf{x}_s)^T (\mathbf{x}_r - \mathbf{x}_s)}.$$

1.2.3 Mahalanobis distance

The Mahalanobis distance between individuals r and s is given by

$$d(\mathbf{x}_r, \mathbf{x}_s) = \sqrt{(\mathbf{x}_r - \mathbf{x}_s)^T S^{-1} (\mathbf{x}_r - \mathbf{x}_s)}.$$

The Mahalanobis distance goes one step further than the Pearson distance by taking into account the correlation between the variables. You can imagine the multiplication by S^{-1} as a rotation of the data so that we have a new set of variables that are uncorrelated.

1.2.4 Manhattan distance

The Manhattan (or taxi cab) distance between individuals r and s is given by

$$d(\mathbf{x}_r, \mathbf{x}_s) = \sum_{i=1}^p |x_{ri} - x_{si}|.$$

The Manhattan distance is the sum of the absolute differences between the variables.

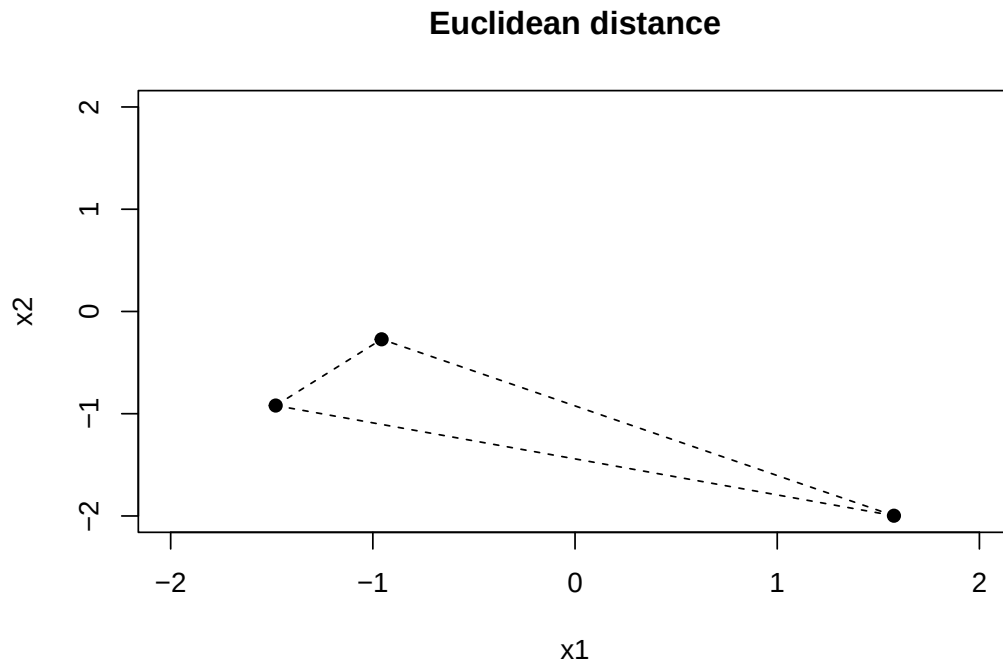


Figure 1.3: Euclidean distance metric for a two-dimensional example

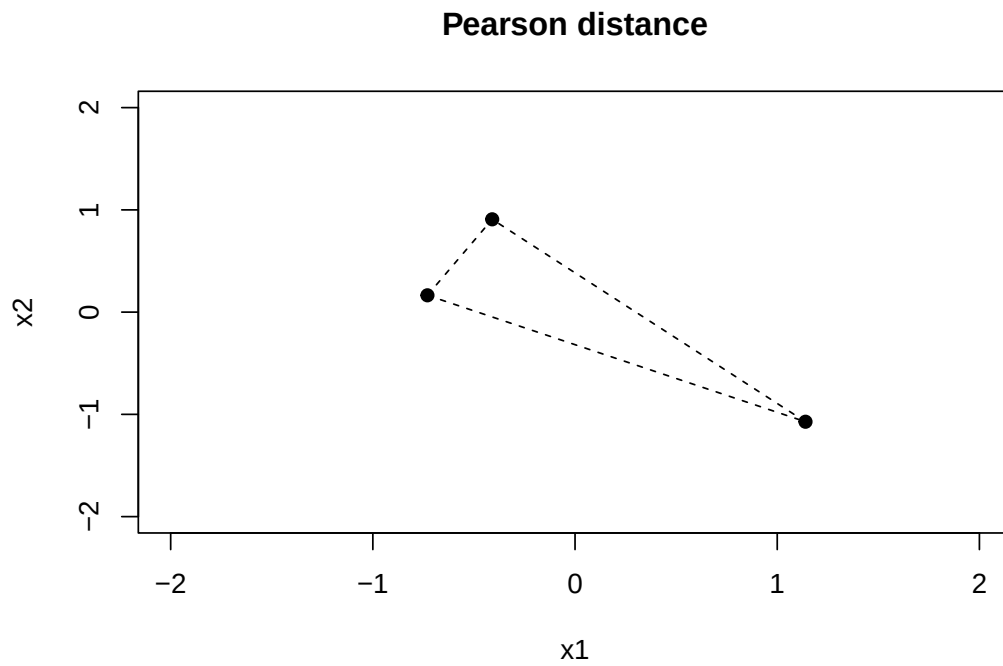


Figure 1.4: Pearson distance metric for a two-dimensional example

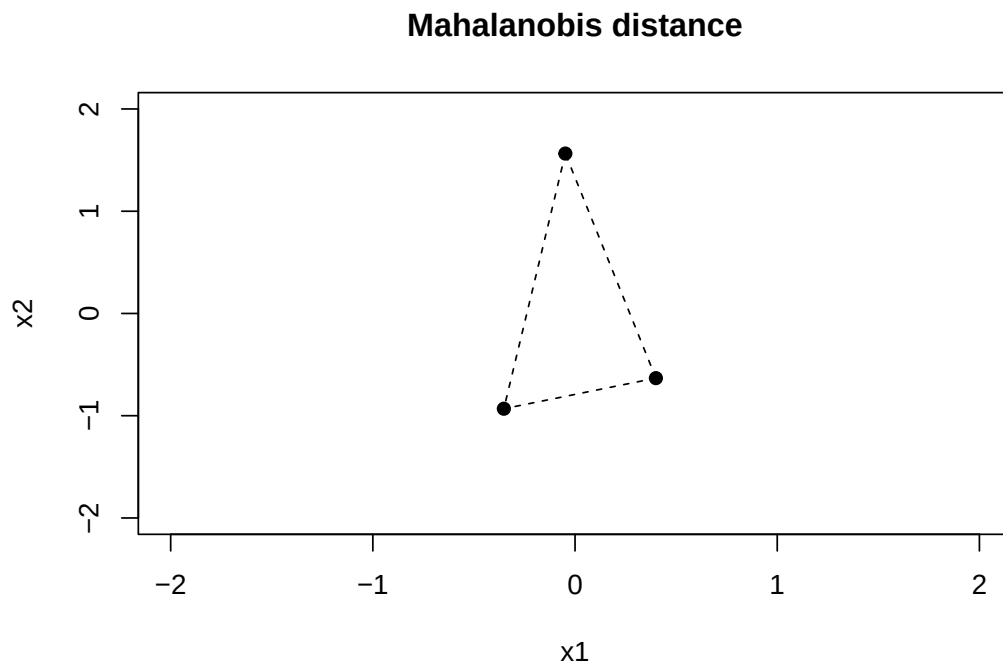


Figure 1.5: Mahalanobis distance metric for a two-dimensional example

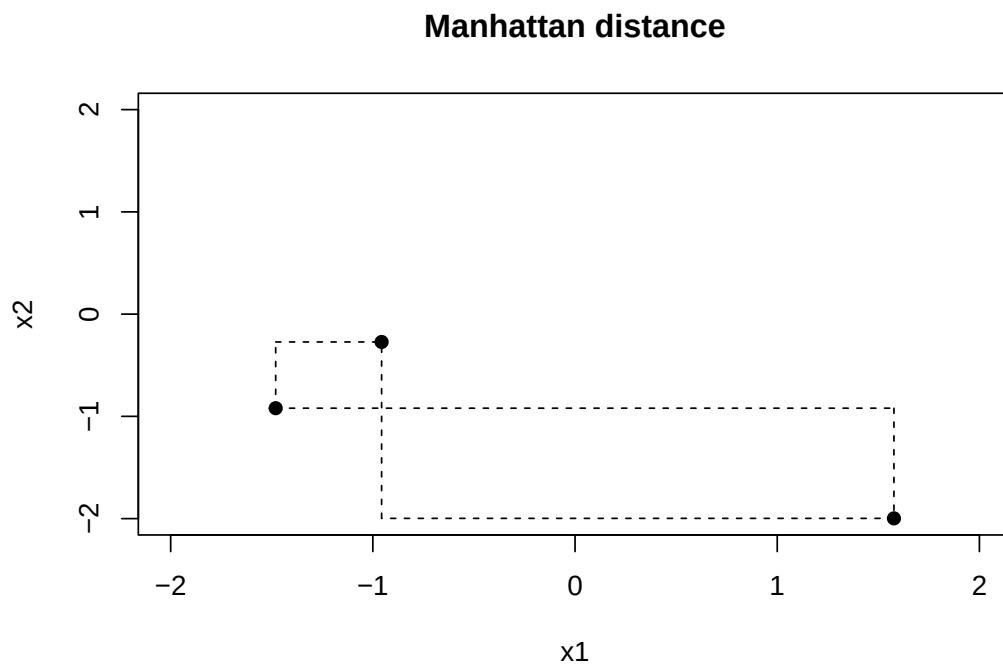


Figure 1.6: Manhattan distance metric for a two-dimensional example

Example

We can visualise the different distance metrics using a simple example. Let's consider a two-dimensional example with three individuals.

A related distance is the Chebyshev (or chessboard) distance, which is the maximum of the absolute differences between the variables:

$$d(\mathbf{x}_r, \mathbf{x}_s) = \max_i |x_{ri} - x_{si}|.$$

This distance is also known as the L_∞ distance, and it is related to the Manhattan and Euclidean distances. This can be seen via the definition of the unfortunately named L_p metric (given that we have the convention of having p variables), which is given by

$$d(\mathbf{x}_r, \mathbf{x}_s) = \left(\sum_{i=1}^p |x_{ri} - x_{si}|^k \right)^{1/k}.$$

The Chebyshev distance is the limit of the L_p metric as $k \rightarrow \infty$. We can visualise the different distances in the context of the chessboard in Figure 1.7.

1.2.5 Distances for categorical data

The distance metrics that we have discussed so far are for continuous data. However, we can also define distance metrics for categorical data. The most common distance metric for categorical data is the Hamming distance. In its original form, the Hamming distance is the number of positions at which the corresponding elements are different. For example, the Hamming distance between “2021” and “2001” is 1. For us, it is the number of categorical variables that are different between two individuals. The challenge with this distance metric is that each variable is treated as equally important, and there is no notion of relative similarity for each variable. Also, if we have many categorical variables, the Hamming distance can become very large.

We can solve this latter problem by effectively normalising the Hamming distance. We can do this by dividing the Hamming distance by the number of categorical variables. This gives us a distance metric that ranges from 0 to 1. This is related to the idea of the matching coefficient, which is the proportion of variables that are the same between two individuals: our distance is therefore $1 - \text{matching coefficient}$.

A popular method for handling the distance between categorical variables involves one hot encoding. This is where we convert the categorical variables into a set of binary variables. For example, if we have a categorical variable with three levels, we can convert this into three binary variables. This allows us to use the Euclidean distance (or whatever distance metric for numerical data we like) between the binary variables. This is a very useful technique because it allows us to use the same distance metric for both numerical and categorical data.

A final metric to consider is the Gower distance, which allows us to weight the variables differently. This is useful because it allows us to account for the fact that some variables are more important than others. The Gower distance is a generalisation of the Hamming distance and the Euclidean distance in that

$$d(\mathbf{x}_r, \mathbf{x}_s) = \frac{\sum_{i=1}^p w_i d_i(\mathbf{x}_{ri}, \mathbf{x}_{si})}{\sum_{i=1}^p w_i},$$

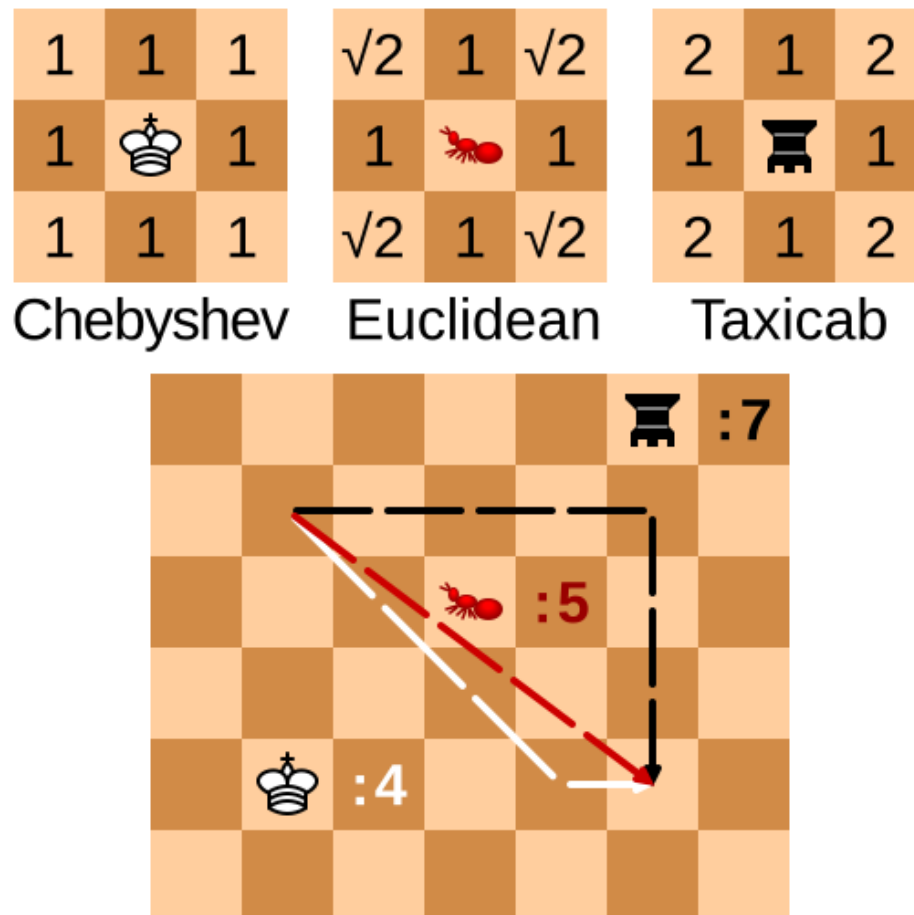


Figure 1.7: Manhattan (or taxi-cab), Euclidean and Chebychev distances marked out on a chessboard.
 <I>Cmglee, CC BY-SA 4.0 via Wikimedia Commons</I>

where w_i is the weight for variable i and d_i is the distance metric for variable i . In the case where we have all categorical variables, it is usual to use the Hamming distance for d_i . The general form here allows us to use any distance metric for the variables, and opens up possibilities for defining distances between observations that mix numerical and categorical variables.

Example

Let's consider a simple example with two individuals and two categorical variables. The first individual has the values "A" and "B" for the two variables, while the second individual has the values "A" and "A". The Hamming distance between the two individuals is 1, while the normalised Hamming distance is 0.5.

If we have a third individual with the values "C" and "C" for the two variables, the Hamming distance between the first and third individuals is 2, while the normalised Hamming distance is 1.

Alternatively, we can consider one-hot encoding the variables. The first individual is encoded as (1, 0, 0, 0, 1, 0), while the second individual is encoded as (1, 0, 0, 1, 0, 0). The Euclidean distance between the two individuals is $\sqrt{2}$. Similarly, the Euclidean distance between the first and third individuals is 2.

1.3 Linear regression

The multiple linear regression model with p predictor variables can be expressed in matrix form as:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

where:

- $\mathbf{y}$ is an $n \times 1$ vector of the response variable
- $\mathbf{X}$ is an $n \times (p + 1)$ design matrix with a column of ones for the intercept term
- β is a $(p + 1) \times 1$ vector of coefficients including the intercept
- ϵ is an $n \times 1$ vector of error terms

More explicitly, the model can be written as:

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

The least-squares estimator of β is given by:

$$\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

This estimator is obtained by minimising the sum of squared residuals with respect to β .

Aside: deriving the least squares estimator

The least squares estimator is derived by minimising the sum of squared residuals. The sum of squared residuals is given by

$$\sum_{i=1}^n (y_i - \hat{y}_i)^2 = \sum_{i=1}^n (y_i - \mathbf{x}_i^T \beta)^2 = (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta).$$

To find the β vector that minimises this expression, we differentiate with respect to β and set the result to zero. The derivative is

$$\nabla_{\beta} (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta) = -2\mathbf{X}^T (\mathbf{y} - \mathbf{X}\beta).$$

Setting this to zero gives

$$\begin{aligned} -2\mathbf{X}^T (\mathbf{y} - \mathbf{X}\hat{\beta}) &= 0, \\ \mathbf{X}^T \mathbf{y} &= \mathbf{X}^T \mathbf{X} \hat{\beta}, \\ \hat{\beta} &= (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}. \end{aligned}$$

This is subject to the technical condition that $\mathbf{X}^T \mathbf{X}$ is invertible. If we have colinearity in the explanatory variables, this matrix will not be invertible.

We might notice that this estimator is related to the covariance matrix of the centred data: recall that $(n-1)S = X^T H X$. This estimator gives rise to residuals that have some important properties (as we will see later): first, we have

$$\hat{\epsilon}^T \mathbf{1}_n = 0,$$

and, secondly, we have

$$\mathbf{X}^T \hat{\epsilon} = 0,$$

where $\mathbf{1}_n$ is a column vector of 1's, and $\hat{\epsilon} = \mathbf{y} - \mathbf{X}\hat{\beta}$ is the vector of residuals.

The first property speaks to a balance in the residuals, while the second property speaks to the residuals being orthogonal to the columns of $\mathbf{X}$. This is important because it means that the residuals are uncorrelated with the predictors.

Example

Let's consider the Anscombe dataset. The Anscombe dataset consists of four sets of data points, each with two variables. The goal is to fit a linear regression model to each set of data points and compare the results.

```
# Load the Anscombe dataset
data(anscombe)

# Plot the Anscombe dataset
par(mfrow = c(2, 2))
for (i in 5:8) {
  plot(anscombe[,x1], anscombe[[i]],
       xlab = "x1", ylab = paste0("y", i), pch = 19)
  abline(lm(anscombe[[i]] ~ anscombe[,x1]),
        col = "red")
}
```

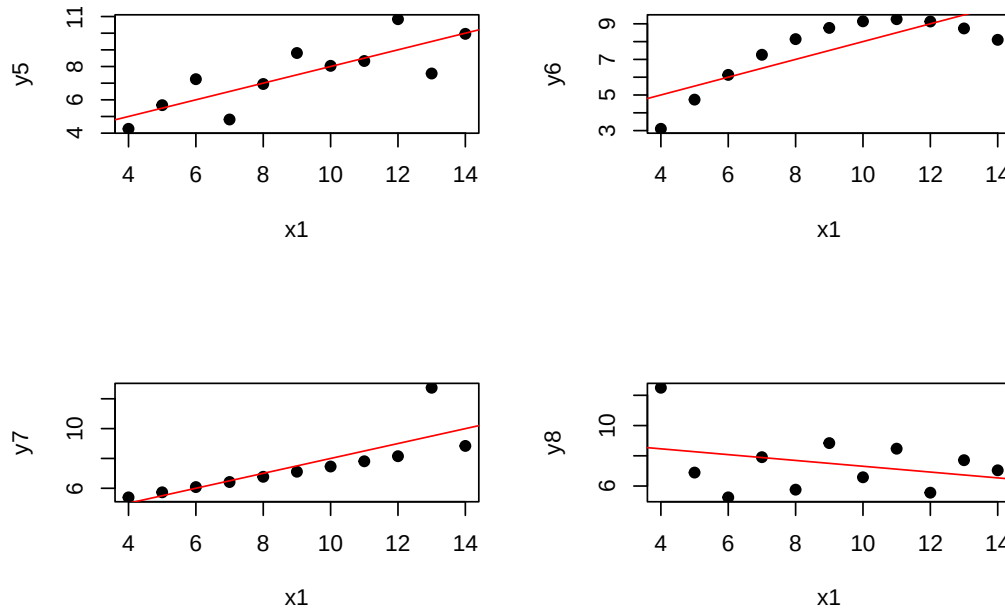


Figure 1.8: Linear fits for the Anscombe dataset

We can see that the linear regression model fits the data well for the first set of data points in that there is an upward trend plus some noise. However, the linear regression model does not fit the data well for the other sets of data points. For the third and fourth data sets, there is a problem with a single outlier. For the second data set, the relationship between the variables is non-linear, which is much more problematic.

However, linear regression does not just mean fitting a straight line to the data. We can also fit a polynomial to the data. For example, we can fit a quadratic model to the second set of data points.

```
# Fit a quadratic model to the second set of data points
quadratic_model <- lm(y2 ~ poly(x1, 2), data = anscombe)
# summary(quadratic_model)

# Plot the quadratic model
plot(anscombe$`x1`, anscombe$y2,
      xlab = "x1", ylab = "y2", pch = 19)
lines(sort(anscombe$`x1`),
      predict(quadratic_model, data.frame(x1 = sort(anscombe$`x1`)),
            type = "response"),
      col = "red")
```

This model is still linear in the sense that we have a linear combination of the various explanatory variables. Here, the expanded form of the model is

$$y_2 = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \epsilon,$$

and x_1^2 is just another variable that we have created. (This is a general idea in statistics and machine learning: we can create new variables from the existing variables to capture more complex relationships.) We should worry a little about colinearity between the variables: essentially, when

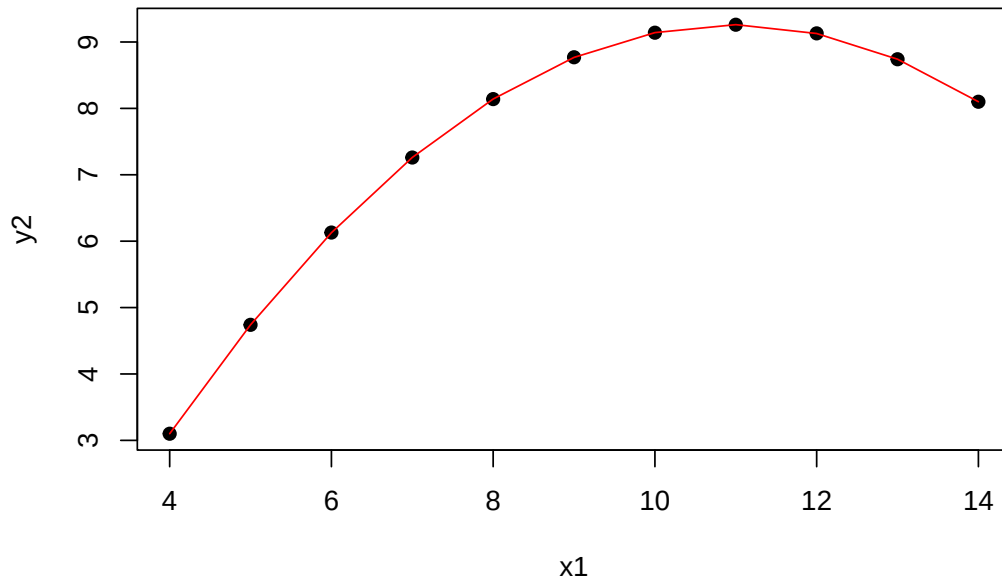


Figure 1.9: Quadratic model for the second set of data points

we have two variables that are highly correlated, we can have issues with the estimation of the coefficients because we do not know which variable deserves the credit for the prediction.

In this case, we calculate the correlation between x_1 and x_1^2 to be 0.99. This is a high correlation, which is not surprising because we have created x_1^2 from x_1 and the near-straight line relationship between x_1 and x_1^2 over the range of interest that can be seen in the following plot.

```
# Plot the relationship between x1 and x1^2
plot(anscombe$x1, anscombe$x1^2,
     xlab = expression(x[1]), ylab = expression(x[1]^2),
     pch = 19)

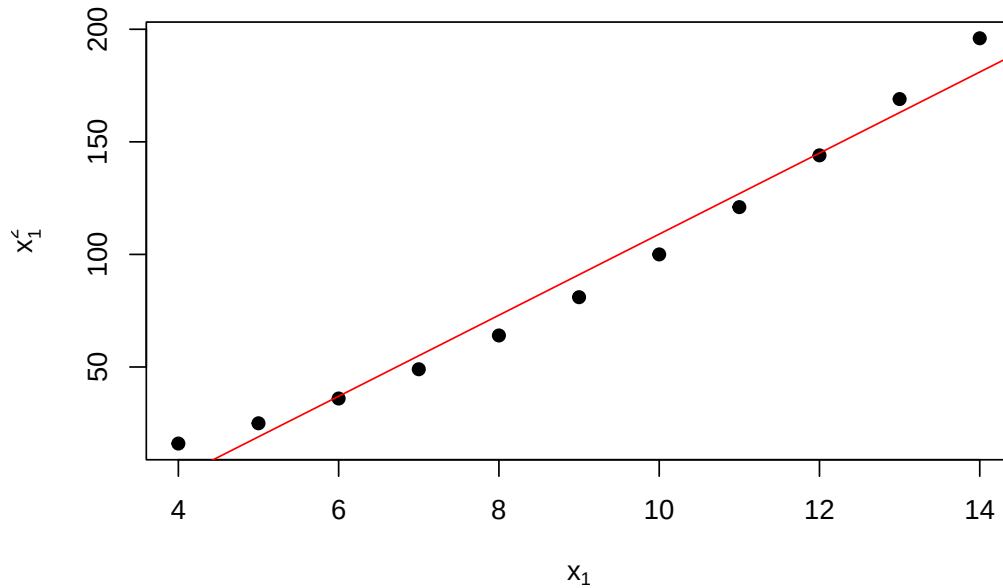
# Add a straight line fit
abline(lm(x1^2 ~ x1,
         data = anscombe),
       col = "red")
```

1.3.1 Using different distance metrics

The fitting of the linear model is usually based on the least squares criterion. This criterion is equivalent to minimising the Euclidean distance between the observed and predicted values. However, we can use different distance metrics to fit the model. For example, we can use the Manhattan distance to fit the model.

```
# Fit the linear model using the Manhattan distance

# Define a function that calculates the MAE
# between the observed and predicted values
mae <- function(y, y_hat) {
  return(sum(abs(y - y_hat)))
}
```

Figure 1.10: Correlation between x_1 and x_1^2

```

}

# Define a function that returns MAE for
# a vector of coefficients
mae_coefficients <- function(beta) {
  y_hat <- anscombe$`x1` * beta[2] + beta[1]
  return(mae(anscombe$y4, y_hat))
}

# Minimise the MAE
mae_model <- optim(c(0, 0), mae_coefficients)
mae_model$par

```

```
## [1] 6.80666450 0.01666688
```

1.3.2 Partitioning of variance

In linear regression, the total variance in the response variable y can be partitioned into components associated with the predictor variables and the residual variance. This partitioning is known as the analysis of variance (ANOVA) for the regression model.

Let SSE denote the sum of squared errors (or residuals):

$$\text{SSE} = \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$$

where $\hat{y}_i$ is the fitted value of the response variable for observation i .

Furthermore, let SST denote the total sum of squares:

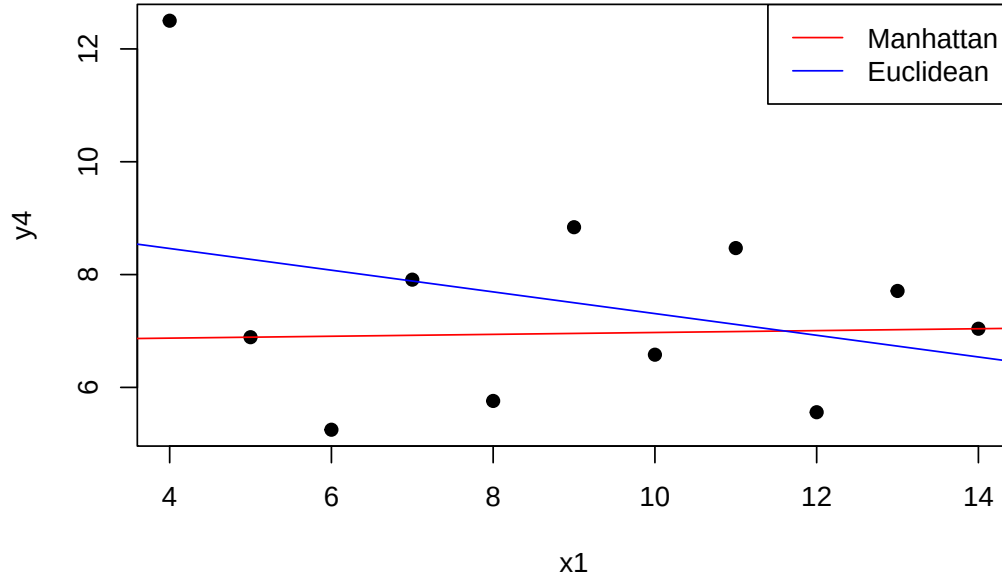


Figure 1.11: Manhattan vs Euclidean distance for the linear model

$$\text{SST} = \sum_{i=1}^n (y_i - \bar{y})^2 = \|\mathbf{y} - \bar{\mathbf{y}}\|_2^2$$

where $\bar{y}$ is the mean of the response variable.

The total sum of squares SST can be partitioned into two components:

$$\text{SST} = \text{SSR} + \text{SSE}$$

where SSR is the sum of squares due to the regression model (or the explained sum of squares):

$$\text{SSR} = \sum_{i=1}^n (\hat{y}_i - \bar{y})^2 = \|\hat{\mathbf{y}} - \bar{\mathbf{y}}\|_2^2.$$

Some texts use SSR for the residual sum of squares, or use names such as SSM or ESS for the explained component. In this course, we use $\text{SST} = \text{SSR} + \text{SSE}$, with SSR denoting the regression sum of squares and SSE denoting the error sum of squares.

This partitioning allows us to quantify the proportion of variance in the response variable that is explained by the predictor variables in the regression model.

The coefficient of determination, R^2 , is defined as:

$$R^2 = \frac{\text{SSR}}{\text{SST}} = 1 - \frac{\text{SSE}}{\text{SST}}$$

R^2 represents the proportion of the total variance in the response variable that is explained by the regression model. It ranges from 0 to 1, with higher values indicating a better fit of the model to the data.

These ideas are closely linked to the bias-variance trade-off. Often we use the difference between the actual values and predicted values to evaluate the model, such prediction error can in fact be decomposed into three parts:

1. **Bias:** The difference between the average prediction of our model and the correct value which we are trying to predict. A model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to high error on training and test data.
2. **Variance:** Variance is the variability of model prediction for a given data point or a value, which tells us how uncertain our model is. Model with high variance pays a lot of attention to training data and does not generalise on the data which it hasn't seen before. As a result, such models perform very well on training data but have high error rates on test data.
3. **Noise:** Irreducible error that we cannot eliminate.

The bias-variance trade-off explains the situation where we are adding just noise by adding complexity to the model. The training error goes down as the model complexity increases, but the test error is at its minimum when the model complexity is at the right level. We can create a graphical visualisation of bias and variance using a bull's-eye diagram. Imagine that the centre of the target is a model that perfectly predicts the correct values. As we move away from the bulls-eye, our predictions get poorer.

We can plot four different cases representing combinations of both high and low bias and variance.

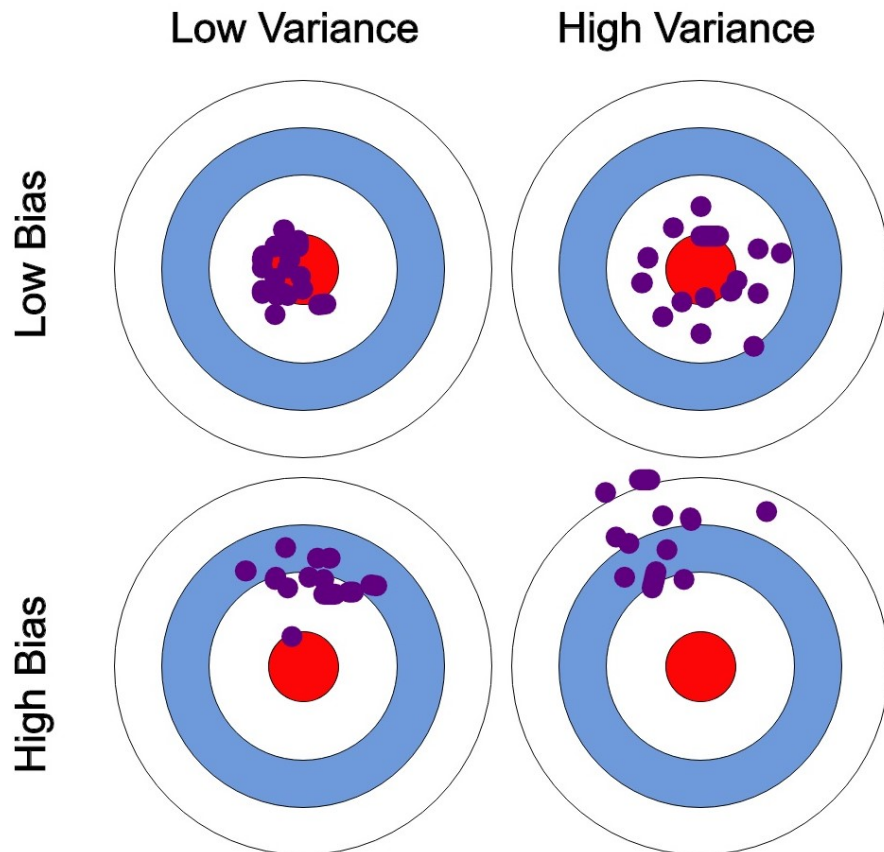


Figure 1.12: Bias-variance trade-off

1.4 Linear discriminant analysis

Linear discriminant analysis (LDA) is a method used in statistics, pattern recognition and machine learning to find a linear combination of features that characterises or separates two or more classes of objects or events. The resulting combination may be used as a linear classifier, or, more commonly, for dimensionality reduction before later classification. LDA was originally proposed by R.A. Fisher in 1936, but the generalisation to more than two classes was not published until 1948 and is attributed to C.R. Rao.

In LDA, we want to find the linear combination of the variables that maximises the ratio of the between-class variance to the within-class variance. The between-class variance is the variance between the classes, while the within-class variance is the variance within each class. The ratio of these two variances is maximised when the classes are well separated.

The LDA method can be broken down into the following steps:

1. Compute the p -dimensional mean vector for each of the classes from the dataset.
2. Compute the variance matrices (within-class and between-class matrices).
3. Compute the eigenvectors and corresponding eigenvalues for the matrices.
4. Sort the eigenvectors by decreasing eigenvalues and choose k eigenvectors with the largest eigenvalues to form a $p \times k$ dimensional matrix W (where every column represents an eigenvector).
5. Use this $p \times k$ eigenvector matrix to transform the samples onto the new subspace. This can be summarised by the matrix multiplication: $U = X \times W$ (where X is a $n \times p$ -dimensional matrix representing the n samples and U are the transformed $n \times k$ -dimensional samples in the new subspace).

There are strong assumptions that the data is normally distributed and that the classes have identical covariance matrices. If these assumptions are not met, LDA may not be the best method to use.

We define between-class variance as

$$S_{\text{between}} = \sum_{i=1}^k n_i (\bar{\mathbf{x}}_i - \bar{\mathbf{x}}) (\bar{\mathbf{x}}_i - \bar{\mathbf{x}})^T,$$

where $\bar{\mathbf{x}}_i$ is the mean of the i th class and $\bar{\mathbf{x}}$ is the overall mean of the data. The pooled within-class covariance matrix may be written directly in terms of the observations as

$$S_{\text{within}} = \frac{1}{n - k} \sum_{i=1}^k \sum_{j=1}^{n_i} (\mathbf{x}_{ij} - \bar{\mathbf{x}}_i) (\mathbf{x}_{ij} - \bar{\mathbf{x}}_i)^T.$$

Equivalently, if the covariance matrix within class i is defined by

$$S_i = \frac{1}{n_i - 1} \sum_{j=1}^{n_i} (\mathbf{x}_{ij} - \bar{\mathbf{x}}_i) (\mathbf{x}_{ij} - \bar{\mathbf{x}}_i)^T,$$

then the pooled within-class covariance matrix can be written as

$$S_{\text{within}} = \frac{1}{n - k} \sum_{i=1}^k (n_i - 1) S_i.$$

We want a linear combination of the variables that maximises the ratio of the between-class variance to the within-class variance. This is equivalent to maximising the general quadratic form:

$$\frac{AS_{\text{between}}A^T}{AS_{\text{within}}A^T}.$$

This maximisation needs an additional constraint to avoid the trivial solution of $A = 0$. The constraint is that $AS_{\text{within}}A^T = 1$.

The classification rule can be derived using the probabilities of belonging to each class and, as we assume that the data is normally distributed, this is easy to calculate. However, as there is the further assumption that the classes have identical covariance matrices, the classification rule is linear in the variables. The discriminant function for class k is given by:

$$\delta_k(\mathbf{x}) = \mathbf{x}^T S_{\text{within}}^{-1} \bar{\mathbf{x}}_k - \frac{1}{2} \bar{\mathbf{x}}_k^T S_{\text{within}}^{-1} \bar{\mathbf{x}}_k.$$

This leads to the classification rule of assigning an observation $\mathbf{x}$ to class k if $\delta_k(\mathbf{x}) > \delta_j(\mathbf{x})$ for all $j \neq k$.

Example

Let's consider the Iris dataset. The Iris dataset consists of 150 observations of iris flowers, with four features (sepal length, sepal width, petal length, petal width) and three classes (setosa, versicolor, virginica). The goal is to classify the flowers into one of the three classes based on the four features. The `iris` data set is widely used in many applications, such as pattern recognition and machine learning.

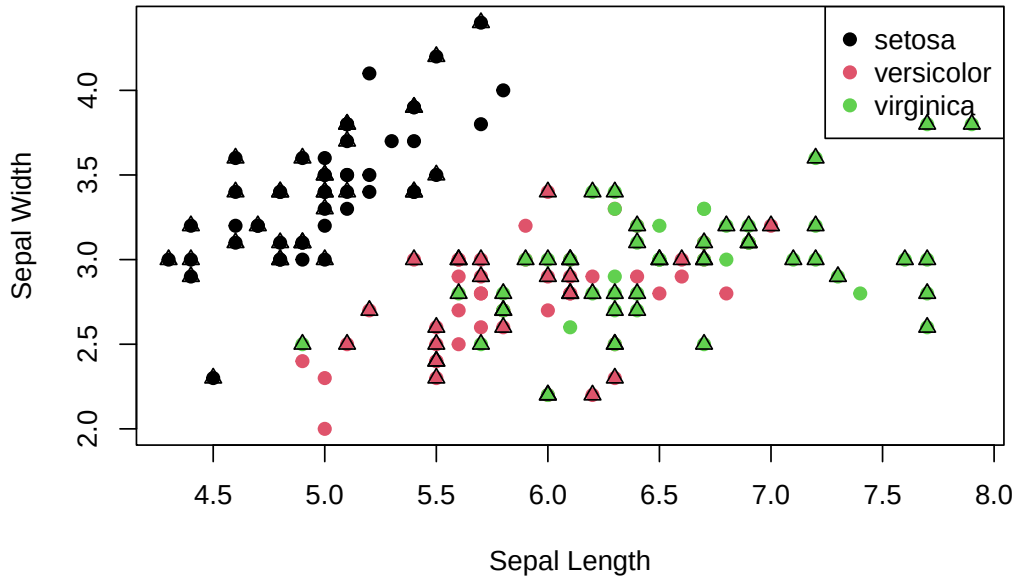


Figure 1.13: Iris dataset with different shapes for test and training and different colours for each class

Using linear discriminant analysis, we can classify the iris flowers into one of the three classes based on the four features.

```
# Linear discriminant analysis
library(MASS)
lda_results <- lda(Species ~ Sepal.Length + Sepal.Width,
                   data = iris)
lda_results

## Call:
## lda(Species ~ Sepal.Length + Sepal.Width, data = iris)
##
## Prior probabilities of groups:
##      setosa versicolor virginica
## 0.3333333 0.3333333 0.3333333
##
## Group means:
##           Sepal.Length Sepal.Width
## setosa              5.006         3.428
## versicolor          5.936         2.770
## virginica           6.588         2.974
##
## Coefficients of linear discriminants:
##           LD1          LD2
## Sepal.Length -2.141178 -0.8152721
## Sepal.Width  2.768109 -2.0960764
##
## Proportion of trace:
##      LD1      LD2
## 0.9628 0.0372
```

The coefficients of the linear discriminant function give us the coefficients of the linear combination of the variables that maximises the ratio of the between-class variance to the within-class variance. Essentially, the data points are being projected into a lower-dimensional space where the classes are well separated.

```
# Plot the predicted classes by changing the shapes of the points
Predicted <- predict(lda_results)$class
ggplot(data = iris) +
  geom_point(aes(x = Sepal.Length, y = Sepal.Width, color = Species,
                 shape = Predicted))

library(klaR)
partimat(Species ~ Sepal.Length + Sepal.Width,
         data = iris, method = "lda")
```

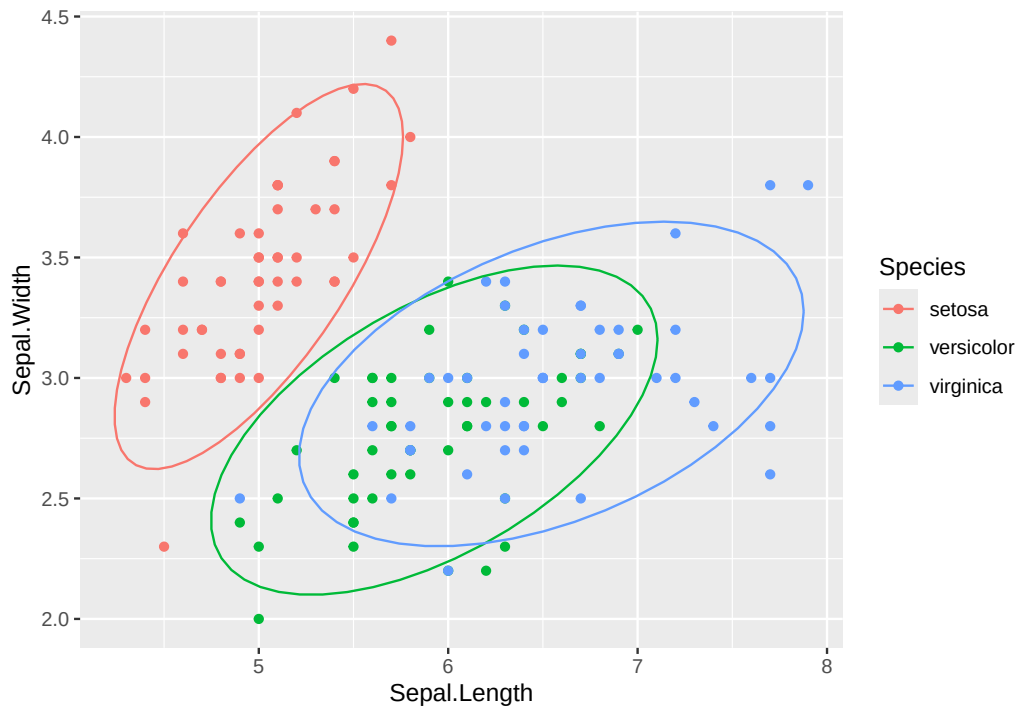


Figure 1.14: LDA results for the Iris dataset with ellipses representing the covariance for each class

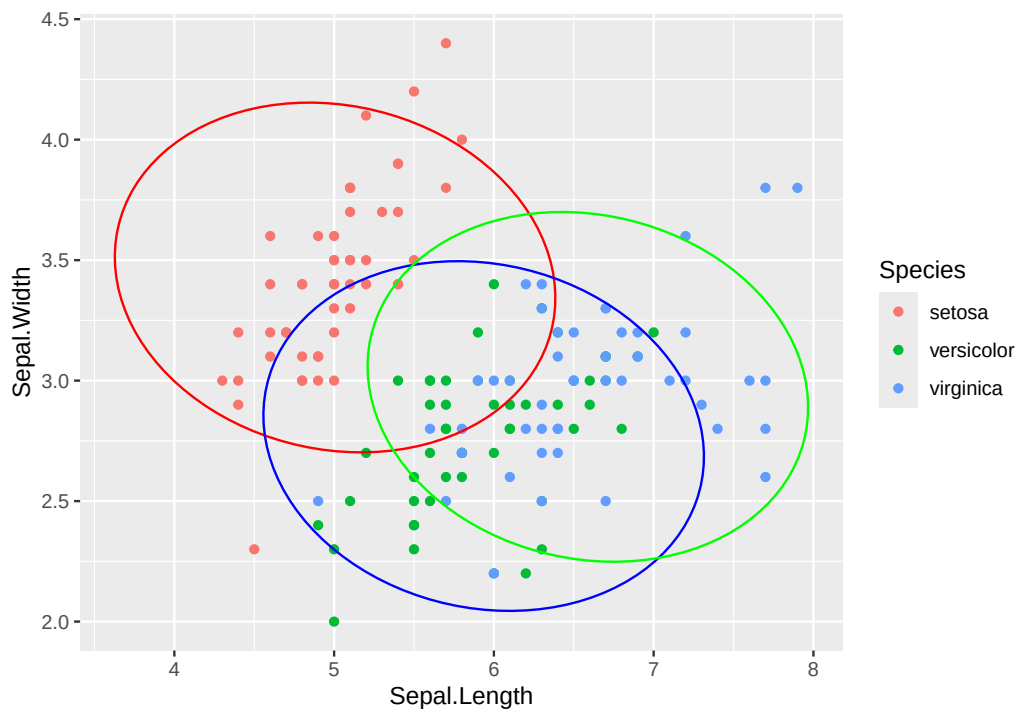


Figure 1.15: LDA results for the Iris dataset with ellipses representing the pooled covariance utilised

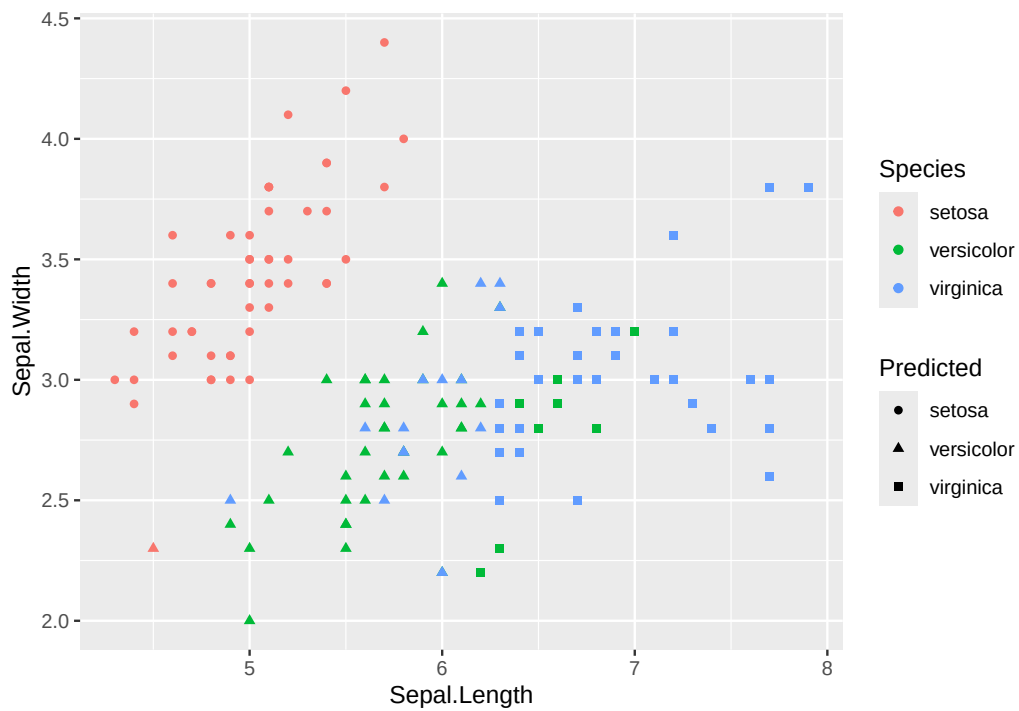


Figure 1.16: LDA results for the Iris dataset with respect to the discriminant boundaries

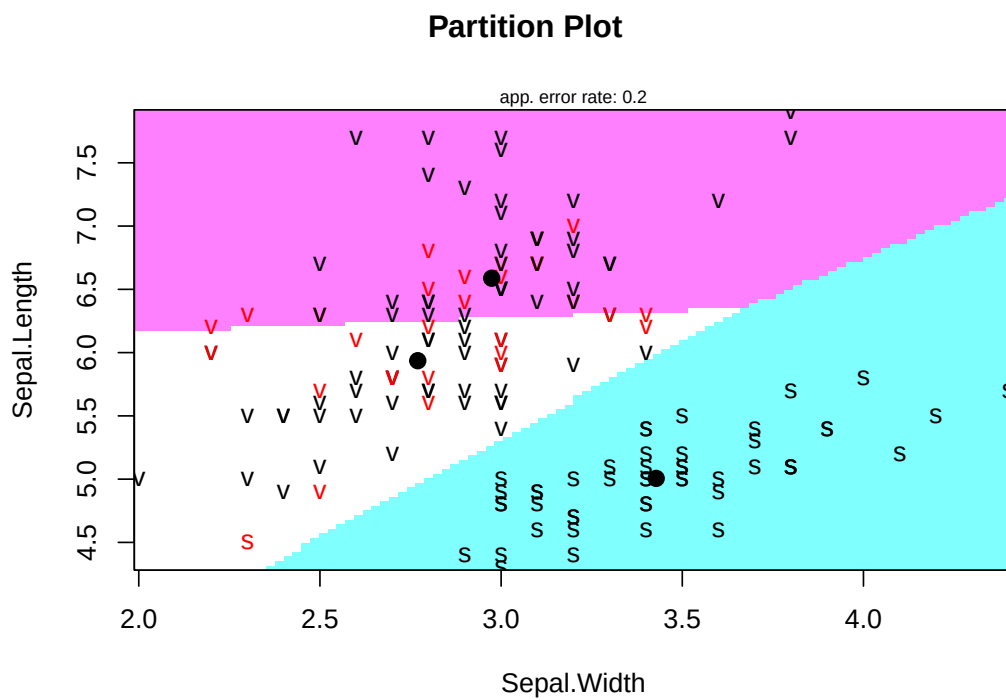


Figure 1.17: LDA results for the Iris dataset with respect to the discriminant boundaries via klaR

Chapter 2

Fundamental concepts

2.1 Types of learning

“Statistical learning, with its roots firmly in probability and statistics, seeks to unveil the hidden structure within data. By building interpretable models, it allows us to not only make predictions but also understand the relationships between variables and even draw causal inferences. Think of it this way: statistical learning, like a skilled detective, meticulously examines the data to uncover the underlying story.”

“Machine learning (ML), however, takes a more pragmatic approach. Borrowing heavily from computer science, it prioritises the creation of algorithms that excel at making accurate predictions on entirely new data. While interpretability might be sacrificed in the process, the raw predictive power gained can be immense. Complex algorithms like neural networks exemplify this philosophy – they may be opaque in their inner workings, but their ability to learn intricate patterns from data allows them to make impressive predictions.”

This seems a bit prejudiced against ML in my opinion, but this text was generated by Google’s LLM, Gemini. . .

I asked ChatGPT, *“which is best: statistical learning or machine learning?”*.

The response was

“Statistical learning focuses on understanding data properties and making inferences, while machine learning prioritises predictive accuracy and scalability. Use statistical learning for hypothesis testing and uncovering relationships, and machine learning for accurate predictions .”

I then asked two follow-up questions: *“Don’t statistical methods produce accurate predictions?”* and *“Can’t ML methods be used to interpret complex data?”*.

The response was

“Certainly! I apologise if my previous response caused any confusion. . . ”

The figures below show that ML and AI started entering the lexicon at around the middle of the last century.

I believe that the lines between them are blurred, but ML journals and applications do seem to be more geared towards predictive performance.

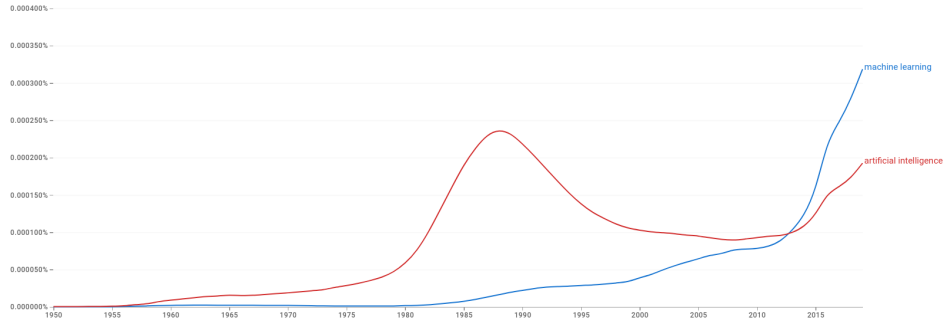


Figure 2.1: Google Books Ngram Viewer: machine learning and artificial intelligence



Figure 2.2: Google Books Ngram Viewer: machine learning, artificial intelligence and statistics

In this half of the module, you will be introduced to both supervised and unsupervised learning. Supervised learning is a type of machine learning where the model is trained on a labelled dataset. The model learns to map input data to output labels by minimising the difference between the predicted and actual labels. Unsupervised learning is a type of machine learning where the model is trained on an unlabelled dataset. The model learns to find patterns and relationships in the data without being given explicit labels. Unsupervised learning is used for tasks such as clustering and dimensionality reduction.

2.2 Training basics

Throughout, we need to keep in mind the distinction between the training and test datasets. The training dataset is used to train the model, while the test dataset is used to evaluate the model's performance. The test dataset is used to assess how well the model generalises to new, unseen data. The test dataset should be separate from the training dataset to avoid overfitting, which is when the model learns the training data too well and performs poorly on new data. These two are also called in-sample and out-of-sample data: in-sample data are training data, while out-of-sample data are used to evaluate the model's performance.

There are also situations when it is advantageous to split the data into three parts: training, validation, and test. The validation dataset is used to tune the hyperparameters of the model (which we could argue is just model training...). Hyperparameters are parameters that are set before the model is trained, such as the learning rate in a neural network or the distance tolerance in DBSCAN. The validation dataset is used to assess the model's performance on new, unseen data and to tune the hyperparameters to improve the model's performance. The test dataset is then used to evaluate

the model’s performance on unseen data.

It is difficult to give a general rule on how to choose the number of observations in each of the three parts. A typical split might be 50% for training, and 25% each for validation and testing. The difficulty in determining the split is that the more data we use for training, the better the model will be. However, the more data we use for testing, the better the estimate of the model’s performance will be. There is a trade-off between the two and it is challenging to come up with theoretical guidelines given the complexity of the data and the model.

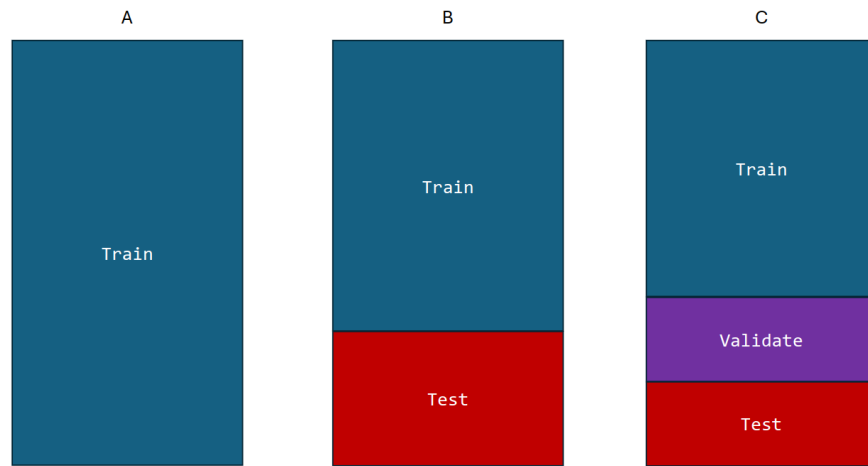


Figure 2.3: Train, validation, and test dataset split

In machine learning literature, you may also come across the terms “training epoch” and “training drift”.

A training epoch is one complete pass through the training dataset. The model is trained on the entire training dataset for a certain number of epochs, which is a hyperparameter that can be tuned. The number of epochs is typically set to a large value, such as 100 or 1000, and the model is trained for that many epochs. The model’s performance is evaluated on the validation dataset after each epoch, and the best model is selected based on the validation performance. A single epoch is rarely enough for a model to learn the underlying patterns in the data effectively. By training for multiple epochs, the model has the opportunity to see the same data points several times, which allows it to refine its understanding and improve its ability to generalise to new, unseen data.

Training drift refers to the phenomenon where the model’s performance on the training dataset improves over time, but its performance on the validation dataset does not improve or even deteriorates. This can happen when the model is overfitting to the training data, meaning that it is learning the noise in the data rather than the underlying patterns. Training drift can be mitigated by using techniques such as early stopping, regularisation, and cross-validation.

Related to this is the underlying assumption that the training and test datasets are drawn from the same distribution. If the training and test datasets are not drawn from the same distribution, the model’s performance on the test dataset may not be representative of its performance on new, unseen data. This is known as distribution shift or concept drift. It can occur when the data changes over time or when the data is collected from different sources. To mitigate this issue, it is important to ensure that the training and test datasets are representative of the same underlying distribution.

There are statistical tests that can be used to assess whether the training and test datasets are drawn from the same distribution. For example, the Kolmogorov-Smirnov test can be used to compare the distributions of two datasets. If the test indicates that the distributions are significantly different, it may be necessary to retrain the model on a new training dataset that is more representative of the test dataset.

2.3 Model performance

There are infinitely many ways to assess the performance of a model. The choice of metric depends on the type of data and the goals of the analysis. In this section, we will discuss some common metrics used to assess the performance of classification and regression models.

To assess model performance, we want to define a loss function that measures the difference between the predicted and actual values. The loss function is used to train the model by adjusting the model's parameters to minimise the loss. The choice of loss function depends on the type of data and the goals of the analysis. For example, the mean squared error is a common loss function for regression models, while the cross-entropy loss is a common loss function for classification models. In the setting of predictions, we write a loss function as $L(y^*, \hat{y})$ where y^* is the actual value and $\hat{y}$ is the predicted value.

Example

Consider the following example of a loss function:

$$L(y, \hat{y}) = (y - \hat{y})^2.$$

How would you interpret this loss function? What does it mean if the loss is zero? What does it mean if the loss is large?

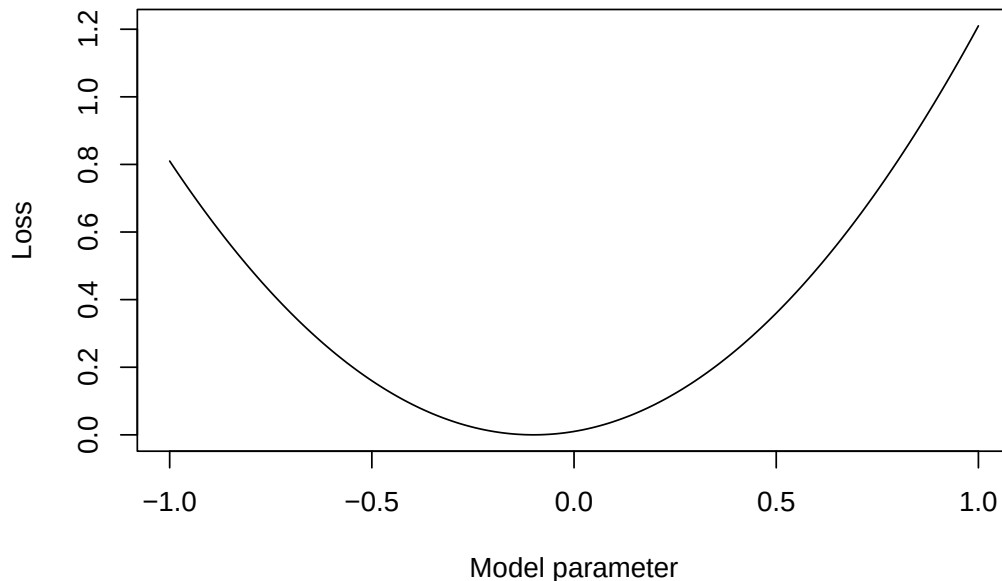


Figure 2.4: 1d plot of the loss function vs a model parameter

Loss functions are often chosen for mathematical convenience. The mean squared error is a common loss function for regression models because it is differentiable and convex, making it easy to optimise. The cross-entropy loss is a common loss function for classification models because it is differentiable and convex, again, making it easier to optimise. If you have chance to look more into decision theory, you will see that the loss function is a key part of the decision-making process: the loss function is used to quantify the cost of making a wrong decision and can be used to guide the model's learning process.

2.3.1 Classification performance

The performance of a classification model can be assessed using a confusion matrix. This is a table that shows the number of true positives, true negatives, false positives and false negatives. Throughout this section, let us consider a binary classification problem where we are trying to predict whether a patient has a disease (positive class) or not (negative class). We have a dataset with 100 patients, of which 75 do not have the disease and 25 do have the disease. We build a classification model that predicts whether a patient has the disease based on some covariates. The model makes the following predictions:

- 80 patients are predicted to not have the disease, of which 70 do not have the disease and 10 do have the disease.
- 20 patients are predicted to have the disease, of which 5 do not have the disease and 15 do have the disease.

2.3.1.1 Confusion matrix

The confusion matrix is a table that shows the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) produced by a classification model.

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

For our disease example, the confusion matrix is:

	Predicted Positive	Predicted Negative
Actual Positive	15	10
Actual Negative	5	70

Roughly speaking, the higher the numbers on the diagonal, the better the model is performing. The off-diagonal elements are the *misclassifications*. The counts encoded in the confusion matrix can be used to calculate various performance metrics.

2.3.1.2 Precision

What proportion of times are we correct when we predict a positive?

Precision is the proportion of true positive predictions out of all positive predictions made by the model. It is calculated as the number of true positive predictions divided by the sum of true positive and false positive predictions.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Precision is a useful metric when the cost of false positives is high. For example, in a medical diagnosis scenario, we want to be sure that a patient diagnosed with a disease actually has the disease before we start treatment. Class imbalance can affect the precision score.

For our running example, the precision is:

$$\text{Precision} = \frac{15}{15 + 5} = \frac{15}{20} = 0.75$$

2.3.1.3 Recall

What proportion of times do we spot an actual positive?

Recall is the proportion of true positive predictions out of all actual positive instances. It is calculated as the number of true positive predictions divided by the sum of true positive and false negative predictions.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Recall is a useful metric when the cost of false negatives is high. For example, in a medical diagnosis scenario, we want to be sure that all patients with a disease are correctly diagnosed. Again, class imbalance can affect the recall score. For example, if we have a dataset with 95% class A and 5% class B, a model that predicts all data points as class A will have a recall of 100%.

For our running example, the recall is:

$$\text{Recall} = \frac{15}{15 + 10} = \frac{15}{25} = 0.6$$

Note that recall is the same as the true positive rate (TPR), which we will meet soon, and sensitivity, which was met in first-year statistics.

2.3.1.4 Accuracy

What proportion of times are we correct?

Accuracy is the proportion of correct predictions made by the model. It is calculated as the number of correct predictions divided by the total number of predictions.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

The accuracy measure is not always the best metric to use, especially when the classes are imbalanced. For example, if 95% of the data points belong to class A and 5% belong to class B, a model that predicts all data points as class A will have an accuracy of 95%, even though it is not making any correct predictions for class B. Later, we will discuss how we can artificially balance the classes to improve the performance of the model.

For our running example, the accuracy is:

$$\text{Accuracy} = \frac{70 + 15}{70 + 5 + 10 + 15} = \frac{85}{100} = 0.85$$

Example

To highlight the potential issues caused by class imbalance, consider the following example of a classification model with three classes. We have a dataset with 300 data points distributed across the three classes as follows:

- Class A: 250 data points
- Class B: 40 data points
- Class C: 10 data points

Our classification model makes the following predictions as captured in this confusion matrix:

	Predicted A	Predicted B	Predicted C
Actual A	240	5	5
Actual B	10	30	0
Actual C	0	5	5

The precision, recall, and accuracy for each class are:

Class	Precision	Recall	Accuracy
A	0.96	0.96	0.933
B	0.75	0.75	0.933
C	0.5	0.5	0.967

It is clear to see from the derivation that the “240” true A class predictions are dominating the calculations. Therefore, any optimisation of the model will be focused on improving the predictions for class A. This is not necessarily a bad thing, but it is important to be aware of the potential biases in the model.

Taking this to an extreme, consider this example again but with class A being predicted perfectly and classes B and C always being mistaken for each other. The precision, recall, and accuracy for each class are:

Class	Precision	Recall	Accuracy
A	1	1	1
B	0	0	0.833

Class	Precision	Recall	Accuracy
C	0	0	0.833

And, now to an even stranger extreme, where everything is predicted as class A:

Class	Precision	Recall	Accuracy
A	0.833	1	0.833
B	NaN	0	0.867
C	NaN	0	0.967

2.3.1.5 Cohen's κ

For binary classification problems, Cohen's κ is given by

$$\kappa = \frac{2(TP \times TN - FP \times FN)}{(TP + FP)(FP + TN) + (TN + FN)(FN + TP)}.$$

This metric is often evaluated as part of many R packages for classification. It is preferred over accuracy when the classes are imbalanced, as it takes into account the agreement between the predicted and actual classes, while also accounting for the chance agreement. The value of κ ranges from -1 to 1, where 1 indicates perfect agreement, 0 indicates no agreement beyond chance, and negative values indicate worse than chance agreement.

We can see the reweighting in action if we rewrite the denominator in terms of the number of data points in the positive class and negative class, N_P and N_N , respectively. The denominator can be rewritten as

$$(TP + FP)(FP + TN) + (TN + FN)(FN + TP) = (TP + FP)N_N + (TN + FN)N_P.$$

We are taking the number of positives and weighting them by the number of negatives, and vice versa.

The numerator is the difference between the observed agreement and the expected agreement by chance. The expected agreement by chance is calculated as the product of the probabilities of each class, which is why we multiply the true positives and true negatives, and subtract the product of the false positives and false negatives.

Example

Let's consider the following example of a classification model with two classes. We have a dataset with 300 data points distributed across the classes as follows: - Class A: 250 data points - Class B: 50 data points

Our classification model makes the following predictions as captured in this confusion matrix:

	Predicted A	Predicted B
Actual A	240	10
Actual B	25	25

The confusion matrix can be used to calculate the accuracy as well as Cohen's κ .

The accuracy is:

$$\text{Accuracy} = \frac{240 + 25}{240 + 10 + 25 + 25} = \frac{265}{300} = 0.88\dot{3}.$$

This accuracy score is driven mostly by the performance within class A.

Cohen's κ is calculated as follows:

$$\kappa = \frac{2(240 \times 25 - 10 \times 25)}{(240 + 10)(10 + 25) + (25 + 25)(25 + 240)} = \frac{11500}{22000} = 0.5227.$$

This κ score indicates moderate agreement between the predicted and actual classes, taking into account the class imbalance.

Example

Let's consider an example that gives rise to a negative value for Cohen's κ . We have a dataset with 300 data points again distributed across the classes as follows: - Class A: 250 data points - Class B: 50 data points

Our classification model makes the following predictions as captured in this confusion matrix:

	Predicted A	Predicted B
Actual A	10	240
Actual B	25	25

The confusion matrix can be used to calculate the accuracy as well as Cohen's κ .

The accuracy is:

$$\text{Accuracy} = \frac{10 + 25}{10 + 240 + 25 + 25} = \frac{35}{300} = 0.11\dot{6}.$$

Cohen's κ is calculated as follows:

$$\kappa = \frac{2(10 \times 25 - 240 \times 25)}{(10 + 240)(240 + 25) + (25 + 25)(25 + 10)} = \frac{-11500}{68000} = -0.1691.$$

2.3.1.6 Precision-recall curve

The precision-recall curve is a graphical representation of the trade-off between precision and recall of a classification model. It is created by plotting precision against recall at various threshold values. The area under the precision-recall curve is a measure of the model's performance. A model with an AUC of 1 is perfect, while a model with an AUC of 0.5 is no better than random guessing.

Let's extend our disease example so that we have a measurement threshold for which we will classify the patient as having the disease if the value is above the threshold. We can then calculate the precision and recall at different threshold values and plot the precision-recall curve. We have 1,000 patients in total, of which 750 do not have the disease and 250 do have the disease. The measurement values are normally distributed with a mean of 37 for patients without the disease and a mean of 39.5 for patients with the disease. We vary the threshold from 30 to 50 in steps of 0.01.

```

data <- data.frame(actual = c(rep(0, 750),
                             rep(1, 250)),
                  measurement = c(rnorm(750,
                                       mean = 37,
                                       sd = 1),
                                 rnorm(250,
                                       mean = 39.5,
                                       sd = 1)))

thresholds <- seq(30, 50, by = 0.01)

# Define a classifier function
classifier <- function(threshold) {
  predicted_positive <- ifelse(data$measurement > threshold,
                              1, 0)

  return(predicted_positive)
}

# Apply the classifier for each threshold
prediction_list <- lapply(thresholds,
                         classifier)

# Calculate precision and recall for each threshold
prec <- sapply(prediction_list,
               function(prediction) {
                 TP <- sum(data$actual == 1 & prediction == 1)
                 FP <- sum(data$actual == 0 & prediction == 1)

                 return(TP / (TP + FP))
               })
recall <- sapply(prediction_list,
                 function(prediction) {
                   TP <- sum(data$actual == 1 & prediction == 1)
                   FN <- sum(data$actual == 1 & prediction == 0)

                   return(TP / (TP + FN))
                 })

# Plot the precision-recall curve
plot(recall,
     prec,
     type = "l",
     xlab = "Recall",
     ylab = "Precision")

```

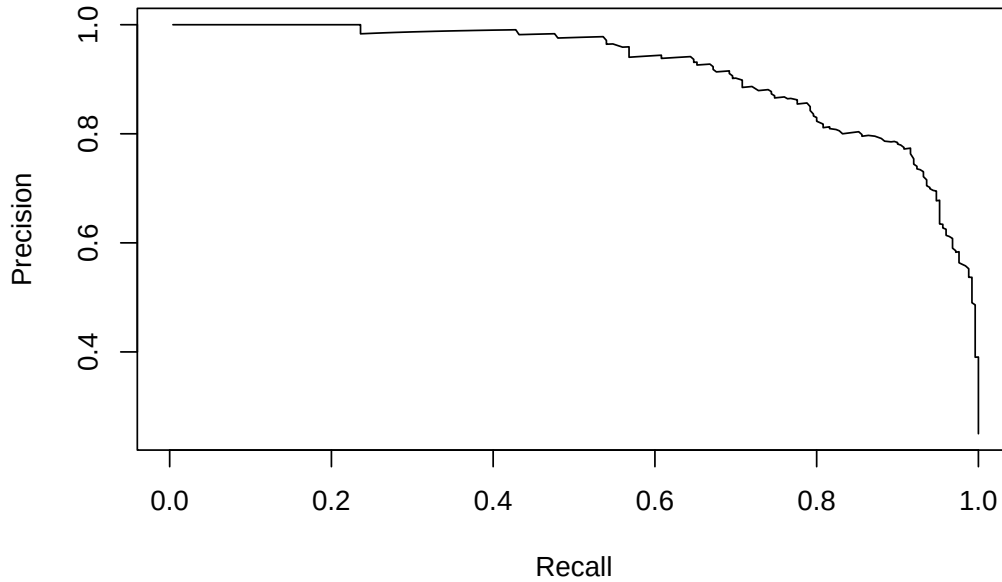


Figure 2.5: Precision-recall curve for the disease example

2.3.1.7 Brier score

The Brier score is a measure of the accuracy of *probabilistic* predictions made by a classification model. Note that there is no room for uncertainty in the metrics that we have discussed so far. We may say that they are making deterministic predictions. I believe that probabilistic predictions are more useful in practice because they allow us to quantify the uncertainty in the model’s predictions. The Brier score is a proper scoring rule that incentivises the model to make accurate probabilistic predictions. It is calculated as the mean squared difference between the predicted probability and the actual outcome. A lower Brier score indicates better performance. It can be easily calculated using the following formula:

$$\text{Brier score} = \frac{1}{N} \sum_{i=1}^N (p_i - o_i)^2.$$

where N is the number of data points, p_i is the predicted probability of the positive class for data point i , and o_i is the actual outcome (1 for positive, 0 for negative) for data point i . The Brier score is an example of a proper scoring rule, which is a measure of the accuracy of probabilistic predictions. It is “proper” in the sense that it incentivises the model to make accurate predictions in such a way that the expected score is minimised.

There is a challenge in utilising such measures with ML classifiers as they often do not give probabilistic predictions. However, there are ways to convert the output of a ML regression model into a probability, such as using the logistic function or the softmax function, which is very similar to the move between linear regression and logistic regression.

Example

Consider the Brier scores for two people predicting the weather over a seven day period. The following table gives their probabilities of rain on each day alongside the actual outcomes:

Day	Actual	Person A	Person B
1	1	0.8	0.6
2	0	0.2	0.4
3	1	0.7	0.7
4	0	0.1	0.3
5	1	0.9	0.8
6	0	0.3	0.3
7	1	0.6	0.5

```

person_A <- c(0.8, 0.2, 0.7, 0.1, 0.9, 0.3, 0.6)
person_B <- c(0.6, 0.4, 0.7, 0.3, 0.8, 0.3, 0.5)
actual <- c(1, 0, 1, 0, 1, 0, 1)

```

```

Brier_score_A <- mean((person_A - actual)^2)
Brier_score_B <- mean((person_B - actual)^2)

```

The Brier scores for person A and B respectively are 0.063 and 0.126.

Example

Let's consider the following example of a classification model in R, which utilises a logistic regression model:

```

# Fit a logistic regression model
model <- glm(actual ~ measurement,
             data = data,
             family = binomial)

# Add the model line to the plot by creating new data
new_data <- data.frame(measurement = seq(30, 50, by = 0.1))
new_data$predicted <- predict(model,
                             newdata = new_data,
                             type = "response")

# Plot the data
plot(data$measurement,
     data$actual,
     xlab = "Measurement",
     ylab = "Actual",
     pch = 19, cex = 0.2)
lines(new_data$measurement,
     new_data$predicted,
     col = "red", lwd = 3)

# Make probabilistic predictions
probabilities <- predict(model,
                        type = "response")

# Calculate the Brier score

```

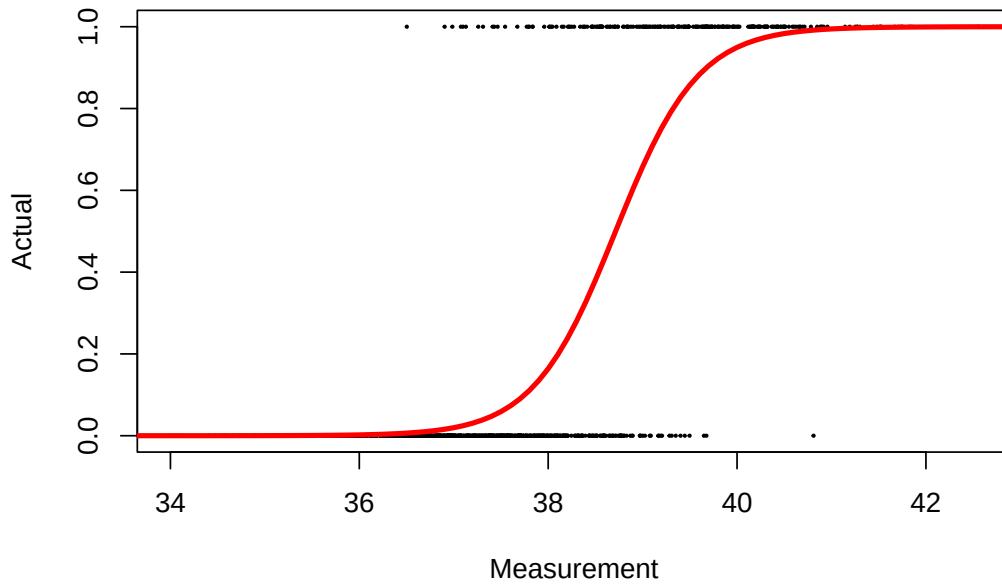


Figure 2.6: Logistic regression model for binary classification

```
log_Brier <- mean((probabilities - data$actual)^2)
```

For this model, the Brier score is 0.067.

We can easily convert the probabilistic predictions into deterministic predictions by setting a threshold. For example, if the predicted probability is greater than 0.5, we can predict the positive class, otherwise, we can predict the negative class. The Brier score for the deterministic predictions is 0.098.

2.3.1.8 Cross-entropy loss

The cross-entropy loss, like the Brier score, is a measure of the difference between the predicted and actual class probabilities. It is calculated as the negative log-likelihood of the predicted class probabilities given the actual class labels. The formula is

$$\text{Cross entropy loss} = -\frac{1}{N} \sum_{i=1}^N o_i \log(p_i) + (1 - o_i) \log(1 - p_i),$$

where N is the number of data points, o_i is the actual outcome (1 for positive, 0 for negative) for data point i , and p_i is the predicted probability of the positive class for data point i . The cross-entropy loss is a proper scoring rule. It is related to the Brier score but is more sensitive to the model's performance at the extremes of the probability range. The cross-entropy loss is commonly used in classification models because it is differentiable and convex, making it easy to optimise.

Example

Consider the cross-entropy loss for the same predictions from the two people predicting the weather over a seven day period.

Person	Brier score	Cross-entropy loss
A	0.063	0.269
B	0.126	0.43

2.3.2 Regression performance

Regression performance needs to take into account more than whether we are wrong or not. As we know, for these continuous models, there is zero chance of us being exactly right. We need to consider how far off we are from the actual value. In this section, we will discuss some common metrics used to assess the performance of regression models.

To aid exposition, let's consider the following simple example of regression models: a straight line, a constant, and a high degree polynomial. We will generate some data from a straight line model and fit the three models to the data.

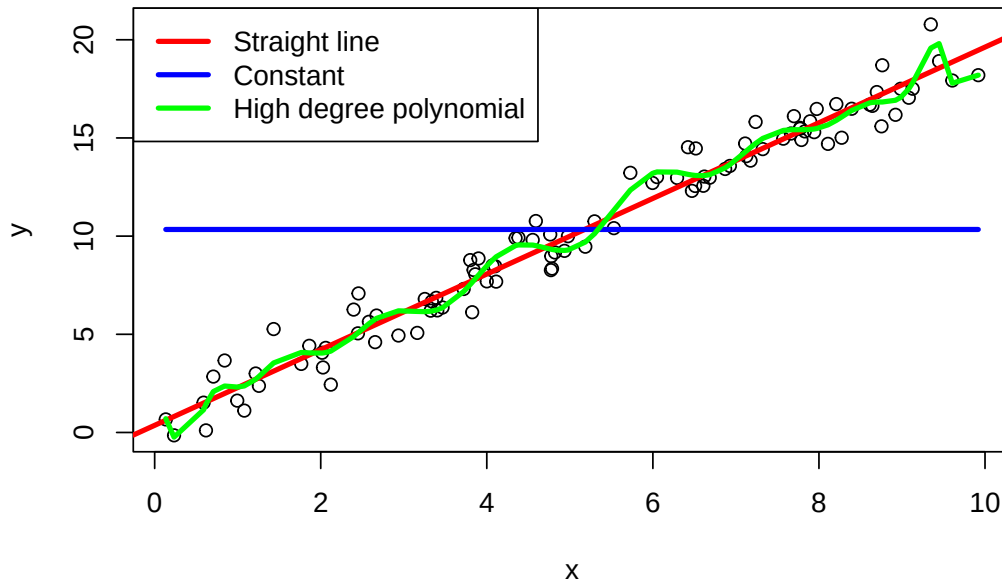


Figure 2.7: Regression models for a straight line, a constant, and a high degree polynomial

2.3.2.1 Mean squared error

The mean squared error (MSE) is a measure of the average squared difference between the predicted and actual values. It is calculated as the sum of the squared differences divided by the number of data points.

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2.$$

where N is the number of data points, y_i is the actual value for data point i , and $\hat{y}_i$ is the predicted value for data point i .

In R, it is easy to set a function to calculate the MSE:

```
MSE <- function(y, y_hat) {
  return(mean((y - y_hat)^2))
}
```

Whilst considering the MSE, it is worth highlighting the bias-variance trade-off. The bias-variance trade-off is a fundamental concept in machine learning that describes the trade-off between the bias and variance of a model. The bias of a model is the error introduced by approximating a real-world problem with a simpler model. The variance of a model is the error introduced by the model's sensitivity to the training data.

We may say that our prediction goal is to replicate the true function that maps the explanatory variables to the response variable, $f(x)$ say. Hence, we have a prediction model, $\hat{f}(x)$, that we hope is as close to $f(x)$ as possible. However, reality is not that simple: we will have a noisy observation of the true function, $y = f(x) + \epsilon$, where ϵ is a random error term (which has a mean of zero and is independent of $f(\cdot)$ and other errors).

We can decompose the squared-error between a predicted and actual value at for an unseen set of explanatory variables as follows:

$$\text{Expected squared error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible error}.$$

The expected squared error over possible training data for an unseen set of explanatory variables is

$$\text{E} \left[(y - \hat{y})^2 \right].$$

The other elements can be defined as

$$\text{Bias} = \text{E} \left[\hat{f}(x) - f(x) \right],$$

which is the average difference between the predicted and actual values,

$$\text{Variance} = \text{E} \left[\hat{f}(x) - \text{E}[\hat{f}(x)] \right]^2,$$

which is the variability of the predicted values around their mean with respect to uncertainty in the inputs, and

$$\text{Irreducible error} = \text{Var}(\epsilon).$$

This partition is useful to keep in mind as it highlights the trade-off between the bias and variance of a model whilst reminding us that there is a limit to how well we can predict the response variable. The presence of the irreducible error term is a reminder that we cannot predict the response variable perfectly.

2.3.2.2 Root mean squared error

The root mean squared error (RMSE) is the square root of the mean squared error. It is a measure of the average difference between the predicted and actual values.

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2}.$$

The RMSE is usually preferred to the MSE because it is in the same units as the target variable, making it easier to interpret.

In R, we have:

```
RMSE <- function(y, y_hat) {
  return(sqrt(mean((y - y_hat)^2)))
}
```

2.3.2.3 Mean absolute error

The mean absolute error (MAE) is a measure of the average absolute difference between the predicted and actual values. It is calculated as the sum of the absolute differences divided by the number of data points.

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|.$$

The MAE is less sensitive to outliers than the MSE and RMSE because it does not square the differences. However, it is not differentiable at zero, which can make it harder to use in optimisation algorithms.

The R code to calculate the MAE is:

```
MAE <- function(y, y_hat) {
  return(mean(abs(y - y_hat)))
}
```

The MAE is of the same form as the RMSE. In fact, they are both examples of the q -norm, which is defined as:

$$\|x\|_q = \left(\sum_{i=1}^N |x_i|^q \right)^{1/q}.$$

The MAE is the q -norm with $q = 1$ and the RMSE is the q -norm with $q = 2$. The q -norm is a generalisation of the Euclidean norm, which is the q -norm with $q = 2$. The q -norm is a measure of the size of a vector in N -dimensional space. The Euclidean norm is the most commonly used norm in machine learning because it is differentiable and convex, making it easy to optimise.

This has also been extended to the infinity norm, which is the maximum absolute value of the vector. The infinity norm is useful when we want to measure the maximum deviation of a vector from zero. It is defined as:

$$\|x\|_{\infty} = \max_i |x_i|.$$

We may see that the infinity norm is the limit of the q -norm as q approaches infinity. The infinity norm is useful when we want to measure the maximum deviation of a vector from zero. It is often used in robust statistics to measure the spread of a distribution.

2.3.2.4 R-squared

The R-squared value is a measure of how well the regression model fits the data. It is calculated as the proportion of the variance in the target variable that is explained by the model. An R-squared value of 1 indicates that the model explains all of the variance in the target variable, while a value of 0 indicates that the model explains none of the variance.

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

where N is the number of data points, y_i is the actual value for data point i , $\hat{y}_i$ is the predicted value for data point i , and $\bar{y}$ is the mean of the actual values.

The R-squared value is automatically calculated for some models in R. If we were to code it ourselves, we would have something like this:

```
R_squared <- function(y, y_hat) {
  return(1 - sum((y - y_hat)^2) / sum((y - mean(y))^2))
}
```

In the context of linear regression, we sometimes see the R-squared value written as:

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}} = \frac{\text{SSR}}{\text{SST}},$$

where SSE is the sum of the squared errors (or residuals), SST is the total sum of squares, and SSR is the sum of the squared differences between the predicted values and the mean of the actual values. With this convention, $\text{SST} = \text{SSR} + \text{SSE}$. You may see other conventions in which SSR denotes the residual sum of squares, but throughout this course SSR denotes the regression (explained) sum of squares. The R-squared value is the proportion of the total sum of squares that is explained by the model and is returned automatically by the `summary()` function in R.

2.3.2.5 Mean squared logarithmic error

The mean squared logarithmic error (MSLE) is a measure of the average squared difference between the logarithm of the predicted and actual values. It is calculated as the sum of the squared differences of the logarithms divided by the number of data points.

$$\text{MSLE} = \frac{1}{N} \sum_{i=1}^N (\log(y_i + 1) - \log(\hat{y}_i + 1))^2.$$

where N is the number of data points, y_i is the actual value for data point i , and $\hat{y}_i$ is the predicted value for data point i . Here, we add 1 to the logarithms to avoid taking the logarithm of zero. The

MSLE penalises underestimates more than overestimates, making it useful when we want to avoid underestimation.

The MSLE can be calculated in R using:

```
MSLE <- function(y, y_hat) {
  return(mean((log(y + 1) - log(y_hat + 1))^2))
}
```

2.3.2.6 Mean absolute percentage error

The mean absolute percentage error (MAPE) is a measure of the average absolute percentage difference between the predicted and actual values. It is calculated as the sum of the absolute percentage differences divided by the number of data points. Note that this metric only makes sense when the actual values are not zero so we tend to add the condition that the $y_i > 0$.

$$\text{MAPE} = \frac{1}{N} \sum_{i=1}^N \left| \frac{y_i - \hat{y}_i}{y_i} \right| \times 100.$$

The MAPE penalises overestimates more than underestimates, making it useful when we want to avoid overestimation.

The MAPE in R is:

```
MAPE <- function(y, y_hat) {
  return(mean(abs((y - y_hat) / y) * 100))
}
```

Example

Consider the three regression models from above. The MSE, RMSE, MAE, R-squared, MSLE, and MAPE for each model are:

Metric	Straight line	Constant	High degree polynomial
MSE	0.84	27.13	0.6
RMSE	0.91	5.21	0.78
MAE	0.71	4.54	0.61
R-squared	0.97	0	0.98
MSLE	0.03	0.44	0.02
MAPE	30.36	254.38	22.49

For the second model, we have a zero R-squared value, which indicates that the model is simply returning the mean of the target variable. We also have a much higher MSLE value than the first model with the MSLE being 15.1138415 times higher. We also have a higher MAPE value than the first model with the MAPE being 8.4 times higher. However, the MAPE is not as high as the MSLE because the MAPE penalises overestimates more than underestimates, and recall that model 2 underestimates the target variable by simply predicting 1 regardless of the input.

2.3.3 Overfitting

Overfitting is a common problem in machine learning (and modelling in general) where a model learns the training data too well and performs poorly on new, unseen data. This can happen when the model is too complex and captures noise in the training data rather than the underlying patterns. Overfitting can be detected by comparing the model's performance on the training and test datasets. If the model performs well on the training data but poorly on the test data, then it is likely overfitting.

There is an interesting philosophical question here about overfitting and whether we believe that the model is capturing reality. If we believe that the model is capturing reality, then we would expect the model to perform well on the test data. The idea that the model might be able to capture reality is the basis of the scientific method, where we use models to make predictions about the world and then test those predictions against new data. In machine learning, the idea that the model might accurately capture the mechanics of reality is tricky due to the complexity of the models and the data.

The third model from above is an example of overfitting in a polynomial regression model.

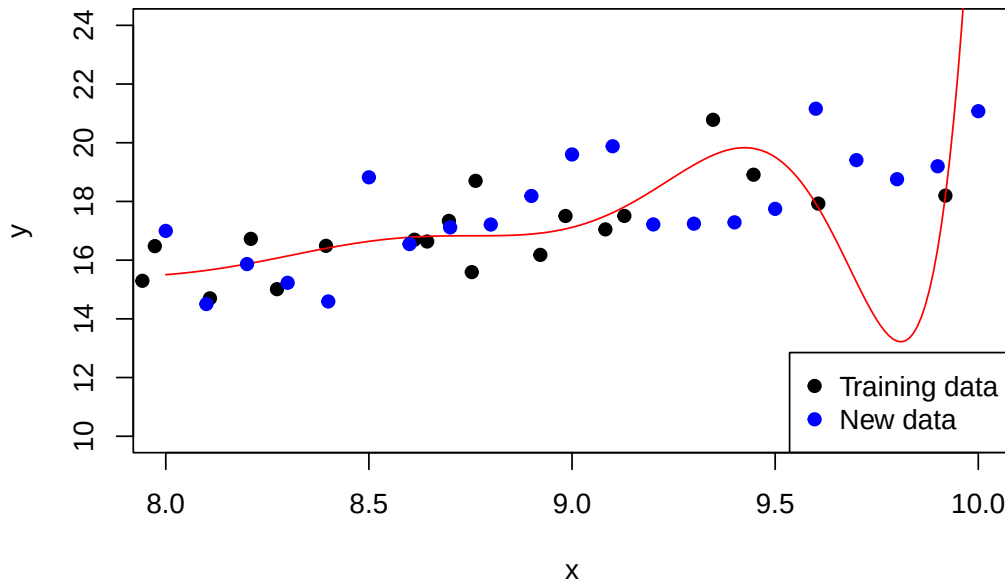


Figure 2.8: Overfitting in a polynomial regression model

Note how the model fits the training data very well but performs poorly on new data (look at the points either side of $x = 9.5$ for instance). This is a sign of overfitting.

There are several techniques to prevent overfitting, such as:

- **Regularisation:** Regularisation is a technique used to prevent overfitting by adding a penalty term to the model's loss function. This penalty term discourages the model from learning complex patterns that are not supported by the data. Common regularisation techniques include L1 regularisation (lasso) and L2 regularisation (ridge). (This naturally occurs in Bayesian inference as some of you will see.)
- **Cross-validation:** Cross-validation is a technique used to assess the performance of a machine learning model. It involves splitting the data into multiple subsets, training the model on some of the subsets, and testing it on the remaining subset. This process is repeated multiple times,

with different subsets used for training and testing each time. The results are then averaged to give an overall estimate of the model's performance. The benefit of using such an approach is that it helps to reduce the variance of the model's performance estimate and is less sensitive to the particular way the data is split.

- **Early stopping:** Early stopping is a technique used to prevent overfitting by stopping the training process when the model's performance on the validation data starts to decrease. This helps to prevent the model from learning the training data too well and generalising poorly to new data.

Earlier in this chapter, we discussed the R-squared value as a measure of how well the regression model fits the data. It is calculated as the proportion of the variance in the target variable that is explained by the model. As we add dummy variables to our model that have nothing to do with the target, the R-squared value will increase. This is because the model is learning the noise in the training data rather than the underlying patterns. This is an example of overfitting. The adjusted R-squared value is a modified version of the R-squared value that takes into account the number of predictors in the model (which makes it useful for regularisation). It is calculated as:

$$\text{Adjusted } R^2 = 1 - \frac{(1 - R^2)(N - 1)}{N - k - 1},$$

where N is the number of data points and k is the number of predictors in the model. The adjusted R-squared value penalises the R-squared value for the number of predictors in the model, making it a more reliable measure of the model's performance.

Example

Consider again the fitting of a polynomial regression model in R that we saw at the start of the overfitting section.

```
# We already have training and test data from the previous example
# Fit a polynomial regression model of degree 20
model <- lm(y ~ poly(x, 20))

# Calculate the R-squared and adjusted R-squared values
R_squared <- summary(model)$r.squared
adjusted_R_squared <- 1 - (1 - R_squared) *
  (length(y) - 1) / (length(y) - 20 - 1)

# Use the model to predict at the test locations
# and calculate the MSE
test_predictions <- predict(model,
                             newdata = data.frame(x = new_x))
MSE_test <- mean((new_y - test_predictions)^2)
```

Here, we have an R-squared value of 0.98 and an adjusted R-squared value of 0.97. The MSE on the test data is 6.35.

Now, let's consider the same example but with a much lower degree polynomial:

```

# Fit a polynomial regression model of degree 4
model <- lm(y ~ poly(x, 4))

# Calculate the R-squared and adjusted R-squared values
R_squared <- summary(model)$r.squared
adjusted_R_squared <- 1 - (1 - R_squared) *
  (length(y) - 1) / (length(y) - 4 - 1)

# Use the model to predict at the test
# locations and calculate the MSE
test_predictions <- predict(model,
                             newdata = data.frame(x = new_x))
MSE_test <- mean((new_y - test_predictions)^2)

```

In Figure 2.11, we can see start to judge the performance of the lower degree polynomial.

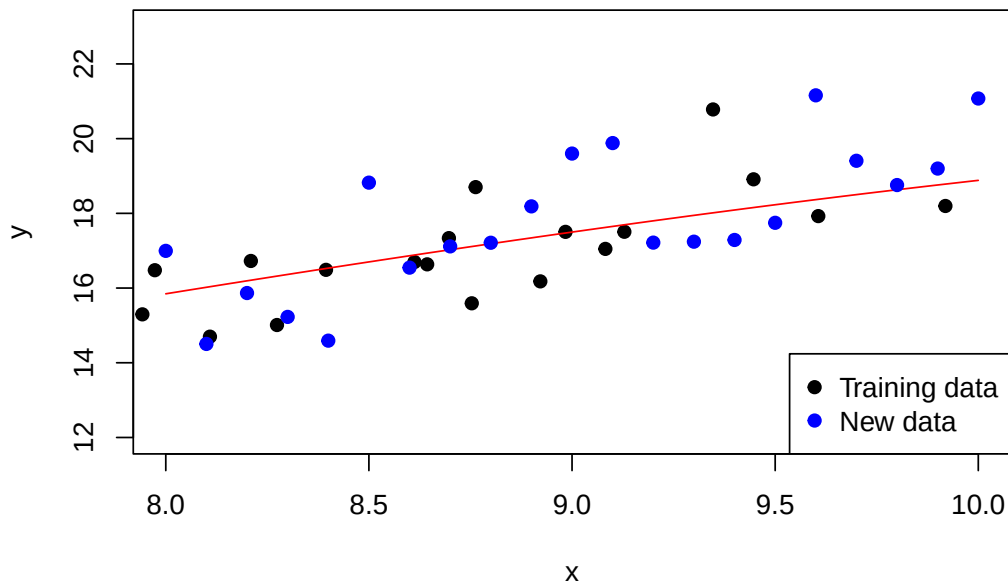


Figure 2.9: Fitting a polynomial regression model with a lower degree

With this set-up, we have an R-squared value of 0.97 and an adjusted R-squared value of 0.97. The MSE on the test data is 1.38.

2.3.4 Model selection criteria

There are several criteria that can be used to select the best model from a set of candidate models. These criteria are used to compare the performance of different models and choose the one that best fits the data. Some common model selection criteria include:

- **Akaike information criterion (AIC):** The AIC is a measure of the relative quality of a statistical model. It is calculated as:

$$\text{AIC} = -2 \log \left(l \left(\hat{\theta} \right) \right) + 2k,$$

where $l(\hat{\theta})$ is the likelihood of the model evaluated for the fitted parameters and k is the number of parameters in the model. The AIC penalises the model for the number of parameters, making it a useful measure of the model's complexity.

- **Bayesian information criterion (BIC):** The BIC is a measure of the relative quality of a statistical model. It is calculated as:

$$\text{BIC} = -2 \log(l(\hat{\theta})) + k \log(N),$$

where, additionally, N is the number of data points. The BIC penalises the model for the number of parameters and the sample size, making it a more conservative measure of the model's complexity. It is termed “Bayesian” because it is derived from a Bayesian perspective with a special form of prior over the model space.

In both these cases, the lower the value, the better the model. Note that these criteria are based on the likelihood of the model, which is the probability of the observed data given the model. Such likelihood calculations are often impossible or entirely nonsensical in the context of many machine learning methods because they are not based on probabilistic models. However, if we consider Gaussian errors, we again see why MSE is often a preferred performance metric.

Example

Consider the polynomial regression models that we have been fitting. In this case, the log-likelihood under Gaussian noise is

$$\begin{aligned} \log(l(\hat{\theta})) &= -\frac{N}{2} \log(2\pi) - \frac{N}{2} \log(\hat{\sigma}^2) - \frac{1}{2\hat{\sigma}^2} \sum_{i=1}^N (y_i - \hat{y}_i)^2 \\ &= -\frac{N}{2} \log(2\pi) - \frac{N}{2} \log\left(\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2\right) - \frac{N}{2} \\ &= -\frac{N}{2} [1 + \log(\text{MSE}) + \log(2\pi)]. \end{aligned}$$

So, we have that the AIC is given by

$$\begin{aligned} \text{AIC} &= N [1 + \log(\text{MSE}) + \log(2\pi)] + 2k \\ &= N \log(\text{MSE}) + 2k + \text{constant}, \end{aligned}$$

and BIC by

$$\begin{aligned} \text{BIC} &= N [1 + \log(\text{MSE}) + \log(2\pi)] + k \log(N) \\ &= N \log(\text{MSE}) + k \log(N) + \text{constant}. \end{aligned}$$

2.4 Cross-validation

Cross-validation is a technique used to assess the performance of a machine learning model. It involves splitting the data into multiple subsets, training the model on some of the subsets, and testing it on the remaining subset. This process is repeated multiple times, with different subsets

used for training and testing each time. The results are then averaged to give an overall estimate of the model's performance. The benefit of using such an approach is that it helps to reduce the variance of the model's performance estimate and is less sensitive to the particular way the data is split.

There are several types of cross-validation, including:

- *k*-fold cross-validation: In *k*-fold cross-validation, the data is split into *k* subsets, or folds. The model is trained on *k*-1 folds and tested on the remaining fold. This process is repeated *k* times, with each fold used as the test set once. The results are then averaged to give an overall estimate of the model's performance.
- Leave-one-out cross-validation: In leave-one-out cross-validation, each data point is used as the test set once, with the remaining data points used for training. This process is repeated *n* times, where *n* is the number of data points. The results are then averaged to give an overall estimate of the model's performance.
- Stratified cross-validation: In stratified cross-validation, the data is split into *k* folds such that each fold contains roughly the same proportion of each class. This helps to ensure that the model is trained and tested on a representative sample of the data.

Here is an example of *k*-fold cross-validation in R utilising the `caret` package for our polynomial example:

```
library(caret)

# Explicitly transform the data to a data frame
# and add polynomial terms
data <- data.frame(x = x, y = y)
data$x2 <- x^2
data$x3 <- x^3
data$x4 <- x^4

# Use cross-validation to fit a polynomial
# regression model of degree 4
model <- train(y ~ .,
               data = data.frame(x = x, y = y),
               method = "lm",
               trControl = trainControl(method = "cv",
                                       number = 10))
```

Recall that without cross-validation, the MSE on the test data was 1.38. With cross-validation, the MSE is 1.32.

If we wanted to use leave-one-out cross-validation, we could set `number = nrow(data)` in the `trainControl` function.

When we have *k* folds, we can calculate the average performance of the model across the different training subsets. This can be useful for comparing different model set-ups. We can also calculate the standard deviation of the performance across the folds to get an idea of the variability of the model's performance. Any model performance metric that we are estimating, θ say, can be calculated using an average across the *k* models:

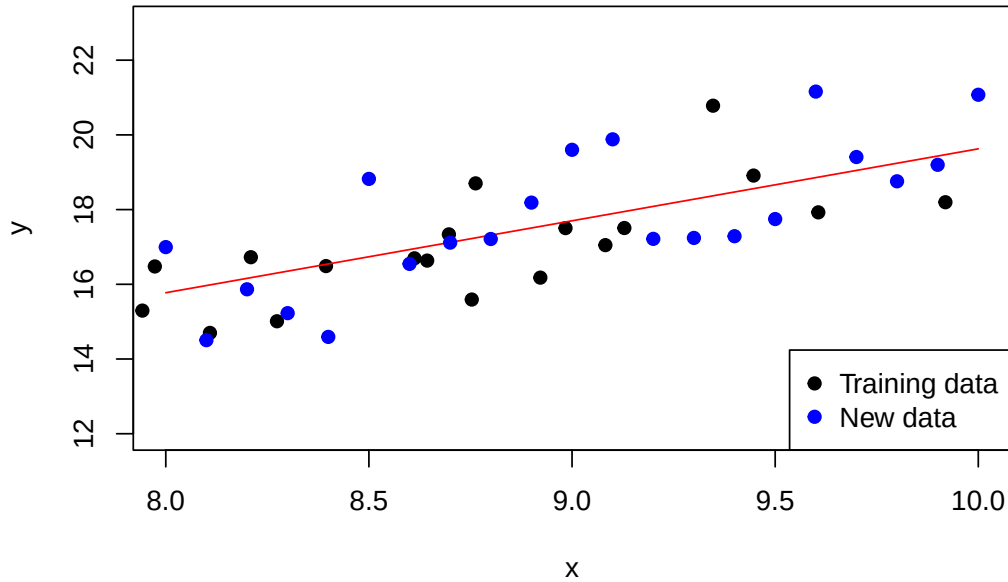


Figure 2.10: Polynomial regression model with 10-fold cross-validation

$$\hat{\theta} = \frac{1}{k} \sum_{i=1}^k \hat{\theta}_i.$$

However, it is worth looking for outliers in the performance metrics across the folds. This can be done by calculating the standard deviation of the performance metrics across the folds. If the standard deviation is large, then it is likely that there are outliers in the performance metrics. This can be a sign that the model is not generalising well to new data.

As you should expect, we get nothing for free. Cross-validation is an example of multiple-counting in statistical estimation, and the more we use the data, the more we are prone to overfitting. This may seem like an odd thing to say given that cross-validation is often used to prevent overfitting. However, the point is that we may do better in terms of producing a robust estimator within the context of the currently observed data, but we may do worse in terms of generalisation to new data. *(Side point: it is also in conflict with the Bayesian approach to inference that would only use each data point once.)*

2.5 Missing or few data

As you have seen, machine learning methods generally depend on high volumes of data with very many recorded variables. As such, missing data is a common problem. There are several ways to deal with missing data, including:

- **Removing missing data:** This is the simplest approach, where we simply remove any rows with missing data. This can be effective if the amount of missing data is small, but it can lead to biased estimates if the missing data is not random.
- **Imputing missing data:** This is a more sophisticated approach, where we fill in the missing data with estimates based on the available data. There are several methods for imputing missing data, including mean imputation, median imputation, and regression imputation.

- **Using models that can handle missing data:** Some machine learning models can handle missing data directly, such as decision trees and random forests. These models can use the available data to make predictions without requiring imputation.

The chosen method will typically have a significant impact on the performance of the model. It is important to consider the amount of missing data and the reason for the missing data when choosing a method. If the missing data is not random, then it is likely that the estimates will be biased. In such cases, it may be better to use a model that can handle missing data directly.

The other (partially-related) issue is that of too few data covering certain categories or parts of the input space. Again, this is a common problem in machine learning, especially when dealing with imbalanced datasets. In extreme cases, we may have no examples of a category in the training data, but it may come up in the test data. This can lead to poor performance of the model, as it has not seen any examples of that category during training. We will look more at this at the end of the next chapter in the section on “Balancing Classes”.

Chapter 3

Supervised Learning — Classification

3.1 Historical notes

Classification applications have a long history in statistics and machine learning. The first ML classification algorithm was the *perceptron*, which we will meet later, invented in 1943 by Warren McCulloch and Walter Pitts and developed for practical use by Frank Rosenblatt in 1957. In statistics, classification has a much longer history with examples dating back to the 19th century. The most famous example is Fisher’s **iris** dataset, which was used to develop Fisher’s linear discriminant analysis (as we have seen). Fisher’s LDA is still used today in many applications, such as facial recognition and handwriting recognition. Another common technique in statistics is logistic regression, which is a linear classifier that uses a logistic function to classify inputs into two categories. With origins in the 1800s and turned into the statistical tool it is today in the middle of the 1900s, logistic regression is widely used in many applications, such as credit scoring and medical diagnosis.

The general idea behind classification is to learn a function that maps inputs to categories. The function is learnt from a training dataset, which consists of input-output pairs. The function is then used to classify new inputs into one of the output categories. The training data need to be *labelled*; that is, the output categories need to be known.

3.2 Running examples

To aid explanations, we will use two well-known data sets throughout this chapter.

The first is the aforementioned **iris** data set. The **iris** data set consists of 150 observations of iris flowers, with four covariates (sepal length, sepal width, petal length, petal width) and three classes (setosa, versicolor, virginica). The goal is to classify the flowers into one of the three classes based on the four variables.

Using linear discriminant analysis, we can classify the iris flowers into one of the three classes based on the four variables (as seen in Chapter 1).

The confusion matrix in this example is

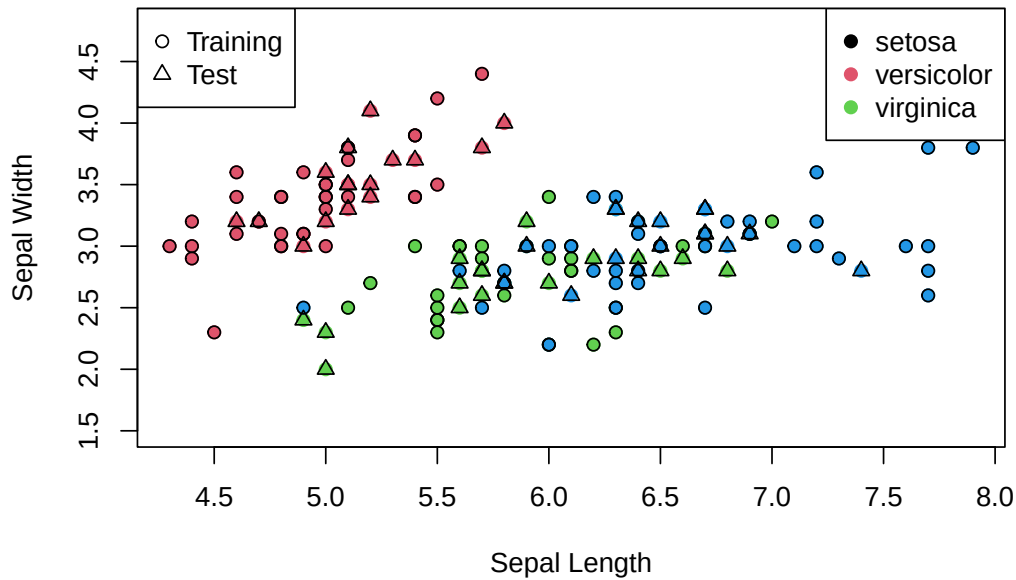


Figure 3.1: The ‘iris’ data set on the sepal length and width axes with different shapes for test and training and different colours for each class.

	setosa	versicolor	virginica
setosa	16	0	0
versicolor	0	19	2
virginica	0	0	13

For later comparison, we can also calculate the accuracy of the linear discriminant analysis model: 0.96.

Our second running example will be the often-cited handwriting recognition data set. Our subset of the handwriting recognition data set consists of thousands of observations of handwritten digits, with 784 covariates (28x28 pixels) and ten classes (0-9). The goal is to classify the digits into one of the ten classes based on the 784 covariates.

```
# Load the handwriting datasets
library(readr)
web_url <- "https://www.maths.dur.ac.uk/users/john.p.gosling/MATH3431_practicals/"
MNIST_test <- read_csv(paste0(web_url,
                              "mnist_test_small.csv"))
MNIST_train <- read_csv(paste0(web_url,
                              "mnist_train_small.csv"))

# We don't need all these data for our examples
MNIST_test <- MNIST_test[1:1000, ]
MNIST_train <- MNIST_train[1:2000, ]
```

The following table shows us the coverage of the digits in the training and test sets.

Digit	Training	Test
0	191	85
1	220	126
2	198	116
3	191	107
4	214	110
5	180	87
6	200	87
7	224	99
8	172	89
9	210	94

Figure 4.2 shows the first digit in the test set.

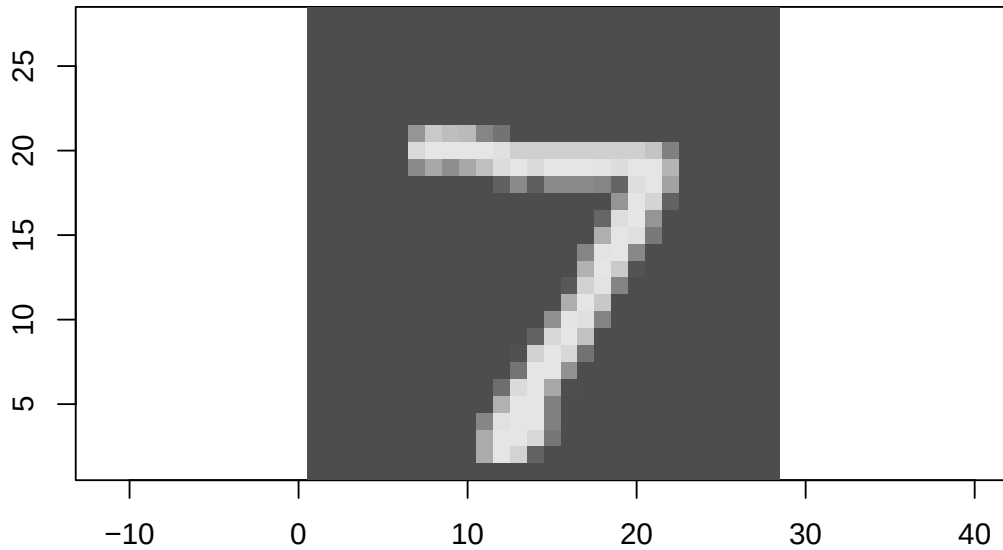


Figure 3.2: The first digit in the test set.

3.3 k -nearest neighbours

The k -nearest neighbours algorithm is a simple and intuitive classification algorithm that is widely used in many applications. The algorithm works by finding the nearest neighbours of a new input and classifying the input based on the majority class of the neighbours. The algorithm is non-parametric because it does not make any assumptions about the underlying data distribution. The algorithm is also *lazy* that means that it does not really learn a function from the training data, but instead stores the training data and uses it to classify new inputs.

3.3.1 Basic algorithm

The k -nearest neighbours algorithm works as follows:

1. Compute the distance between the new input and each training input.
2. Sort the distances in ascending order.

3. Select the k nearest neighbours of the new input.
4. Classify the new input based on the majority class of the neighbours.

The distance metric used to compute the distances can be any metric, such as Euclidean distance, Manhattan distance, or Minkowski distance. The value of k can be any positive integer, but is typically chosen to be a number greater than the number of classes to help avoid ties.

The computational cost of this algorithm is in the calculation of all the distances and sorting. In practice, implementations of the algorithm use efficient data structures and algorithms to speed up the computation. In particular, the k -*d tree* data structure is often used to speed up the computation of the distances.

Example

For this illustration, we will just use two variables from the `iris` data set: `Sepal.Length` and `Sepal.Width`.

```
# k-nearest neighbours
library(class)

# Compute the distance between the new input
# and each training input
knn_results <- knn(train = train_data[, 1:2],
                   test = test_data[, 1:2],
                   cl = train_data$Species,
                   k = 5)

# Create the confusion matrix
confusion_matrix <- table(test_data$Species, knn_results)
confusion_matrix

##           knn_results
##           setosa versicolor virginica
## setosa           16           0           0
## versicolor        0          10          11
## virginica         0           4           9

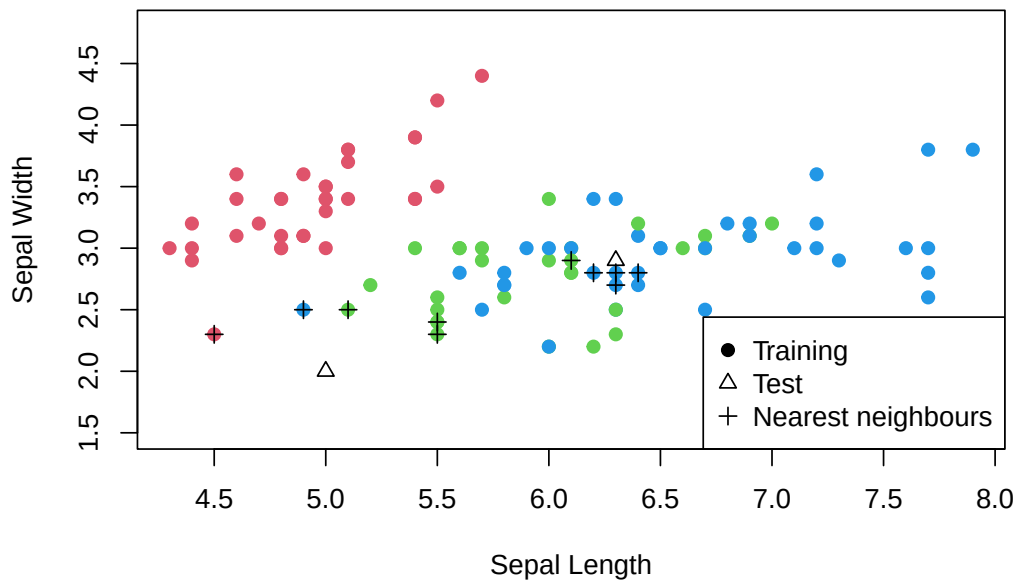
# Calculate the accuracy
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
```

The accuracy here is 0.7. The following plot highlights what the algorithm is doing for a couple of the test points.

As noted, we are free to utilise any distance metric we wish. Consider the same example as above, but using the Manhattan distance metric.

```
# k-nearest neighbours with Manhattan distance
knn_results <- NULL

# For each test point
for (i in 1:nrow(test_data)) {
  # Compute the Manhattan distance between the test point
```

Figure 3.3: The *k*-nearest neighbours algorithm in action for the ‘iris’ data set.

```
# and each training point
distances <- apply(train_data[, 1:2], 1,
  function(x) sum(abs(x - test_data[i, 1:2])))
# Find the k nearest neighbours, checking for ties
nearest_neighbours <- train_data[distances <= sort(distances)[5], ]

# Classify the test point based on the majority
# class of the neighbours again checking for ties
candidates <- names(table(nearest_neighbours$Species)[
  table(nearest_neighbours$Species)>=sort(table(nearest_neighbours$Species),
    decreasing = TRUE)[1]])
knn_results[i] <- sample(candidates)[1]
}

# Confusion matrix
confusion_matrix <- table(test_data$Species, knn_results)
confusion_matrix

##          knn_results
##          setosa versicolor virginica
## setosa      16         0         0
## versicolor   0         8        13
## virginica    0         4         9

# Accuracy
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
```

This is getting worse: the accuracy now is 0.66. Looking back at the plot of the data, we can see that the data is more spread out along the x-axis than the y-axis. This means that the Manhattan

distance metric is not a good choice for this data set. The Euclidean distance metric is a better choice because it takes into account the spread of the data along both axes. Also, in this implementation, it is clear that care is needed when deciding how to deal with ties. If there are ties in the distances, then the algorithm will allow all the data points to vote on the classification of the test point. If there are ties in the class vote, then the algorithm will randomly select a class from the tied classes.

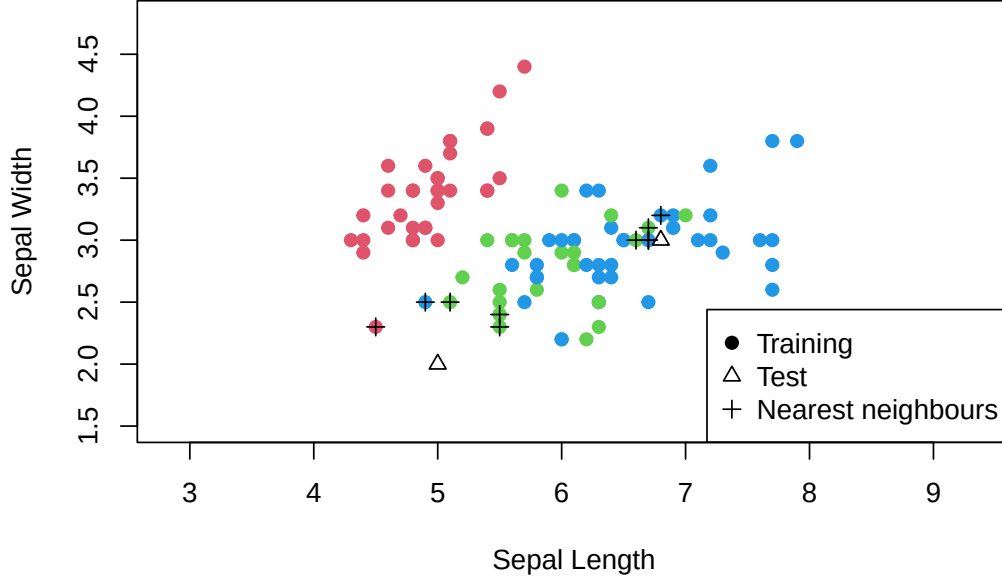


Figure 3.4: The k -nearest neighbours algorithm in action for the ‘iris’ data set using the Manhattan distance.

3.4 Naive Bayes classifier

The naive Bayes classifier is another efficient classification algorithm that is widely used in many applications. The algorithm is based on Bayes’s theorem, which is a fundamental theorem in probability theory. The algorithm is called “naive” because it makes the naive assumption that the variables are conditionally independent given the class. This assumption simplifies the computation of the class probabilities and makes the algorithm computationally efficient. The probabilities underpinning the naive Bayes classifier can be written as:

$$\Pr(y|x_1, x_2, \dots, x_n) = \frac{\Pr(y) \prod_{i=1}^n \Pr(x_i|y)}{\Pr(x_1, x_2, \dots, x_n)},$$

where y is the class label, $x_1, x_2, \dots, x_n$ are the covariates, and $\Pr(y)$, $\Pr(x_i|y)$, and $\Pr(x_1, x_2, \dots, x_n)$ are the class prior, the covariate likelihood, and the evidence, respectively. Now, the *non-naive* version would look like this:

$$\Pr(y|x_1, x_2, \dots, x_n) = \frac{\Pr(y)\Pr(x_1, x_2, \dots, x_n|y)}{\Pr(x_1, x_2, \dots, x_n)}.$$

The $\Pr(x_1, x_2, \dots, x_n|y)$ term is the joint likelihood of the covariates given the class, which is much more complex to compute than the product of the individual likelihoods.

In practice, the *prior* $\Pr(y)$ and the *likelihood* $\Pr(x_i|y)$ are estimated from the training data using maximum likelihood estimation. The *evidence* $\Pr(x_1, x_2, \dots, x_n)$ is a constant that does not affect the classification decision and can be ignored. The class label is then assigned to the input with the highest class probability.

More explicitly, the algorithm works as follows:

1. Estimate the class prior $\Pr(y)$ and the covariate likelihood $\Pr(x_i|y)$ from the training data.
2. Compute the class probability for each class using Bayes's theorem for each test set of predictors.
3. Assign the class label with the highest probability for each test set of predictors.

The estimate of the class prior is the proportion of the class labels in the training data, while the estimate of the variable likelihood depends on the variable type. For continuous variables, the likelihood is estimated using a normal distribution, and, for categorical variables, the likelihood is estimated using a categorical distribution.

Example

Imagine we have a data set with two variables and two classes. We can use the naive Bayes classifier to classify the data into one of the two classes.

Here is our rather small data set.

x_1	x_2	y
1.5	A	Positive
1.7	B	Positive
1.3	A	Positive
1.9	B	Positive
2.1	A	Positive
2.3	B	Negative
2.5	A	Negative
2.7	B	Negative
1.9	A	Negative

We have estimated class prior probabilities of $\Pr(\text{Positive}) = 5/9$ and $\Pr(\text{Negative}) = 4/9$.

We have also estimated the likelihoods of the variables as follows:

- $x_1|\text{Positive} \sim N(1.7, 0.1)$,
- $x_1|\text{Negative} \sim N(2.3, 0.1)$,
- $x_2|\text{Positive} \sim \text{Categorical}(P_A = 0.6, P_B = 0.4)$,
- $x_2|\text{Negative} \sim \text{Categorical}(P_A = 0.5, P_B = 0.5)$.

Example

Here is the naive Bayes classifier in action for the MNIST data set.

```
# Naive Bayes
# This library is based on material from a course at TU Wien
library(e1071)
```

```

# Convert the digit column to a factor
MNIST_train$label <- as.factor(MNIST_train$label)

# Train the Naive Bayes classifier
naive_bayes_model <- naiveBayes(label ~ .,
                                data = MNIST_train)

# Predict the class labels for the test data
naive_bayes_predictions <- predict(naive_bayes_model,
                                   newdata = MNIST_test)

# Create the confusion matrix
confusion_matrix <- table(MNIST_test$label,
                           naive_bayes_predictions)
confusion_matrix

```

```

##      naive_bayes_predictions
##           0    1    2    3    4    5    6    7    8    9
## 0  74    1    0    0    0    1    2    0    7    0
## 1   0 125    0    0    0    0    1    0    0    0
## 2   6  22  20    5    0    4  18    0  41    0
## 3   9  20   1  26    0    5    7    2  32    5
## 4   5  10   0    0  18    0  12    1  26  38
## 5  10  13   1    0    2    1    4    0  54    2
## 6   4   5   1    0    0    0  72    0   5    0
## 7   2  24   0    1    2    1    1  29  13  26
## 8   1  19   1    1    2    2    0    0  60    3
## 9   1  13   0    0    1    0    1    0   5  73

```

It can be useful to visualise the confusion matrix as a heat map when there are so many categories.

3.4.1 Missing data in naive Bayes

As the classifier is trained in a marginal way, the missing values are not a problem. The algorithm can still estimate the variable likelihoods using the available data. The algorithm can also handle missing values in the test data by ignoring the missing values when making predictions. This means that the algorithm can still make predictions even when some of the input variables are missing. In this case, if the labels are missing, those data points have no value in the training process.

Example

Revisiting our simple example, we add some missing values to the data set.

x_1	x_2	y
1.5	A	Positive
1.7	B	Positive
1.3	A	Positive
?	?	Positive

x_1	x_2	y
2.1	A	Positive
2.3	B	Negative
2.5	A	Negative
?	?	Negative
1.9	A	Negative

As nothing has changed for y , we still have $\Pr(\text{Positive}) = 5/9$ and $\Pr(\text{Negative}) = 4/9$. The difference comes for the likelihoods of the variables. We can still estimate the likelihoods of the variables as follows:

- $x_1|\text{Positive} \sim N(1.7, 0.1)$,
- $x_1|\text{Negative} \sim N(2.3, 0.1)$,
- $x_2|\text{Positive} \sim \text{Categorical}(P_A = 0.75, P_B = 0.25)$,
- $x_2|\text{Negative} \sim \text{Categorical}(P_A = 0.67, P_B = 0.33)$.

3.5 Decision trees

Decision trees are a popular and powerful classification algorithm that is widely used in many applications. The algorithm works by recursively partitioning the covariate space into regions that are homogeneous with respect to the class labels. The algorithm is based on a tree structure, where each node represents a decision based on a covariate, and each leaf node represents a class label. The algorithm is simple and intuitive, and it can handle both numerical and categorical covariates.

The decision tree algorithm works as follows:

1. Select the best variable to split the data.
2. Split the data into two subsets based on the variable.
3. Repeat steps 1 and 2 for each subset until the data is homogeneous with respect to the class labels.
4. Assign the majority class label to each leaf node.

The algorithm can be used to classify new inputs by traversing the tree from the root node to a leaf node based on the variable values of the input. The class label of the leaf node is then assigned to the input.

What is meant here by the “best variable” or “homogeneous with respect to the class labels”? Essentially, we want the data in each region belongs to a single class so let’s pick the variable where things are most confused in terms of class. This is where the Gini impurity or entropy come in. The Gini impurity is a measure of the impurity of a region, with a value of 0 indicating that the region is pure and a value of 1 indicating that the region is impure. The entropy is a measure of the impurity of a region, with a value of 0 indicating that the region is pure and a value of 1 indicating that the region is impure. The formula for the Gini impurity is:

$$I_G(t) = 1 - \sum_{i=1}^c \Pr(i|t)^2,$$

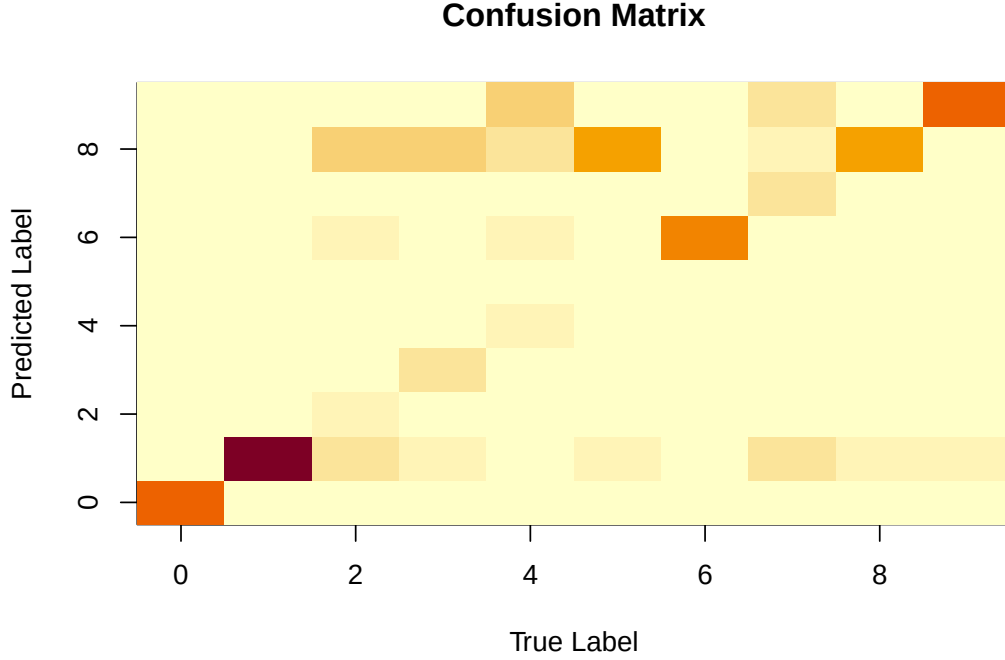


Figure 3.5: The confusion matrix for the MNIST data set from the naive Bayes classifier.

where $I_G(t)$ is the Gini impurity of node t , c is the number of classes, and $\Pr(i|t)$ is the probability of class i at node t . This form for the Gini impurity is the most common, but there are other forms that can be used:

$$\begin{aligned}
 I_G(t) &= \sum_{i=1}^c \left(\Pr(i|t) \sum_{j \neq i} \Pr(j|t) \right) \\
 &= \sum_{i=1}^c \Pr(i|t) (1 - \Pr(i|t)) \\
 &= \sum_{i=1}^c \Pr(i|t) - \sum_{i=1}^c \Pr(i|t)^2.
 \end{aligned}$$

The first form here lays out the motivation for the impurity: we want to find the class that is most likely to be confused with another class. Roughly speaking, how often would a randomly selected label be incorrect if it was randomly assigned according to the distribution of labels in the node? These probabilities are usually crudely estimated using the class labels of the data points in each region (just like in the naive Bayes classifier).

The second option in R is entropy:

$$I_H(t) = - \sum_{i=1}^c \Pr(i|t) \log_2 \Pr(i|t),$$

where $I_H(t)$ is the entropy of node t , c is the number of classes, and $\Pr(i|t)$ is the probability of class i at node t . The motivation for this formula can be found in information theory. The entropy is a measure of the uncertainty of a region, with a value of 0 indicating that the region is certain and a value of 1 indicating that the region is uncertain. The entropy is a more complex measure than the Gini impurity, but it can be more sensitive to the distribution of the class labels.

Both of these are special cases of Tsallis entropy. Tsallis entropy can be defined as:

$$I_q(t) = \frac{k}{q-1} \left(1 - \sum_{i=1}^c \text{Pr}(i|t)^q \right),$$

where $q > 1$ is a parameter that controls the sensitivity of the entropy to the distribution of the class labels and k is a constant. The Gini impurity is a special case of Tsallis entropy when $q = 2$, and the entropy is a special case of Tsallis entropy as $q \rightarrow 1$. We can derive entropy from this formula using L'Hôpital's rule.

When constructing the tree, we want to find the variable that minimises the impurity of the regions. To find this, we compute the impurity of the region before the split and the impurity post-split:

$$I_q(\{u, v\}) = \frac{n_u}{n} I_q(u) + \frac{n_v}{n} I_q(v),$$

where $I_q(\{u, v\})$ is the impurity of the split, n_u and n_v are the number of data points in the regions u and v , and n is the total number of data points. We then choose the variable that minimises the impurity of the split.

Example

Let's consider the digit recognition data set and use the decision tree algorithm to classify the digits into one of the ten classes based on the 784 covariates.

```
# Decision tree
library(rpart)

# Train the decision tree
decision_tree_model <- rpart(label ~ ., data = MNIST_train)

# Predict the class labels for the test data
decision_tree_predictions <- predict(decision_tree_model,
                                     newdata = MNIST_test,
                                     type = "class")

# Create the confusion matrix
confusion_matrix <- table(MNIST_test$label,
                          decision_tree_predictions)

# Visualise the confusion matrix
image(0:9, 0:9, confusion_matrix,
      xlab="True Label",
      ylab="Predicted Label",
      main="Confusion Matrix")
```

We can look at the tree to try to work out how the data are being classified.

If we squint, we can see that the decision tree is essentially asking if a pixel is blank or not. With surprisingly few splits, the tree is able to classify the digits with a high degree of accuracy.

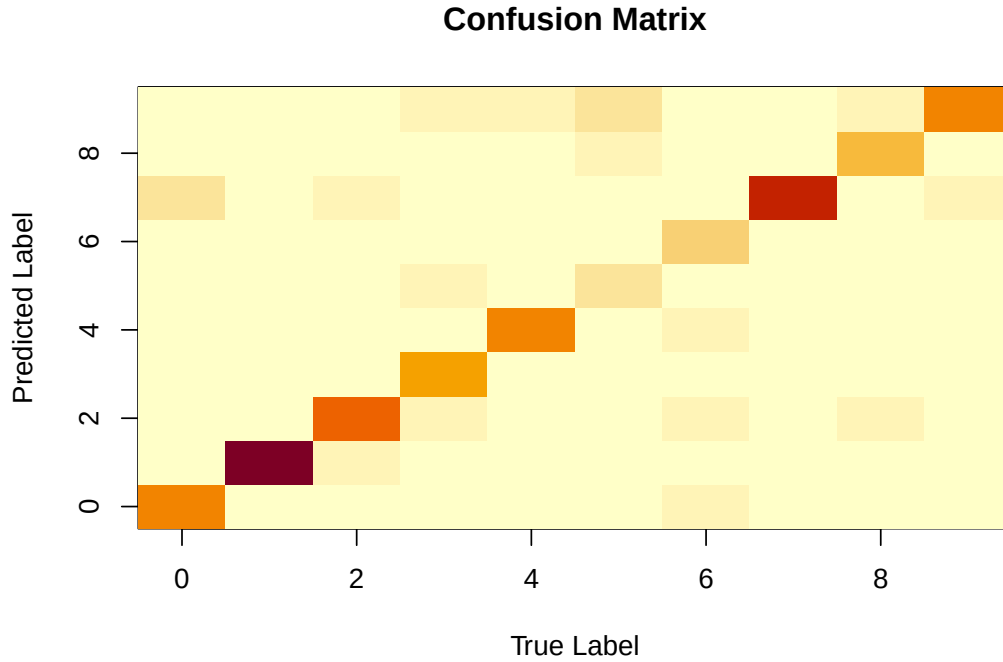


Figure 3.6: The confusion matrix for the MNIST data set from the decision tree classifier. (#fig:class-chunk-13, 3.mnist.dt)

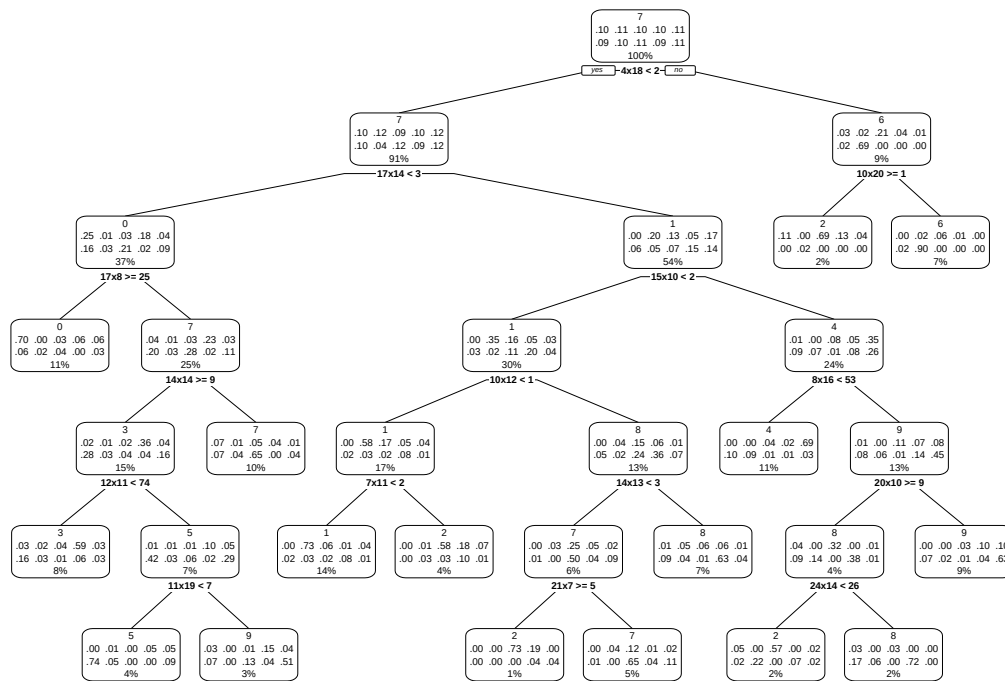


Figure 3.7: The decision tree for the MNIST data set.

3.5.1 Missing data in decision trees

A key advantage of decision trees is their inherent ability to manage missing values during the model training process. The primary method employed is the surrogate split. When a value for one of the input variables is missing, the decision tree algorithm can use a surrogate variable to make the split instead. This surrogate variable is chosen based on its ability to mimic the original variable's split as closely as possible. The surrogate split allows the decision tree to continue making predictions even when some input variables are missing, thus enhancing the model's robustness and flexibility.

If there are many missing values in the training data, the decision tree algorithm can still create a model that is able to make predictions on new data. The surrogate split is particularly useful in situations where the missing values are not random and are related to the class labels. In such cases, the surrogate split can help to improve the accuracy of the model by using the information from the surrogate variable to make the split.

If there are missing values in the test data, the decision tree algorithm can still make predictions by using the surrogate split. The surrogate split is used to make the prediction based on the available input variables. If the surrogate variable is not available, the decision tree algorithm could use the majority class label of the training data to make the prediction.

For example, imagine that we have the following decision tree with three variables: x_1 , x_2 and x_3 . The decision tree is trained on a data set with some missing values in all variables. The training algorithm is as follows: 1. If x_2 is missing, use x_1 to make the split. 2. If x_1 is also missing, use x_3 to make the split. 3. If all three variables are missing, use the majority class label of the training data to make the prediction.

3.5.2 Complexity parameter

When fitting a decision in R using `rpart` or `caret`, a complexity parameter (`cp`) is used to control the size of the decision tree and to prevent overfitting. The complexity parameter is a hyperparameter that controls the trade-off between the size of the tree and the accuracy of the model. A small value of `cp` will result in a large tree that may overfit the training data, while a large value of `cp` will result in a small tree that may underfit the training data.

Mathematically, the complexity parameter is used to prune the decision tree by removing nodes that do not improve the accuracy of the model. The pruning process is based on the following formula:

$$\text{Cost} = \text{Error} + \alpha \times \text{Size},$$

where Cost is the cost of the tree, Error is the error of the tree on the training data, Size is the number of nodes in the tree and α is the complexity parameter. The pruning process removes nodes that do not improve the accuracy of the model by minimising the cost of the tree.

3.6 Perceptron

The perceptron is a linear classifier that uses a threshold function to classify inputs into two categories. The perceptron was the first machine learning algorithm capable of learning from its mistakes, and it was used in early applications of machine learning, such as optical character recognition. The perceptron is a simple and efficient algorithm that is widely used in many applications.

We start with a set of weights assigned to each input variable. The algorithm works by updating the weights of the inputs based on the classification error. The weights are updated using the following

update rule:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(y_r - \hat{y}_{t,r})\mathbf{x}_r,$$

where $\mathbf{w}_t$ is the weight vector at time t , α is the learning rate, y_r is the true class for observation r (assumed to be -1 or 1) and $\hat{y}_{t,r}$ is the predicted class at time t . The algorithm continues to update the weights until the classification error is minimised. Given the weights, the prediction is made by computing the dot product of the weights and the input variables and applying a threshold function to the result:

$$\hat{y}_t = \text{sign}(\mathbf{w}_t^T \mathbf{x}),$$

where $\hat{y}$ is the predicted class label, w_i is the weight of covariate i , $\mathbf{x}$ is the input covariate vector, and $\text{sign}(\cdot)$ is the threshold function that returns 1 if the input is greater than zero and -1 otherwise. Note that, like in linear regression, it is conventional to add a constant term to the input variables.

We can prove that the perceptron algorithm converges to a solution if the data is linearly separable. The proof is based on the fact that the perceptron algorithm updates the weights in the direction of the true class label, which eventually leads to a solution that separates the data into two classes.

Here is the perceptron algorithm in pseudo-code including the update rule and tunable learning rate:

1. Initialise the weights to zero.
2. Set the learning rate.
3. Set the maximum number of iterations.
4. Repeat the following steps until the maximum number of iterations is reached:
5. For each input in the training data:
6. Compute the predicted class label.
7. Update the weights based on the classification error.
8. Return the weights.

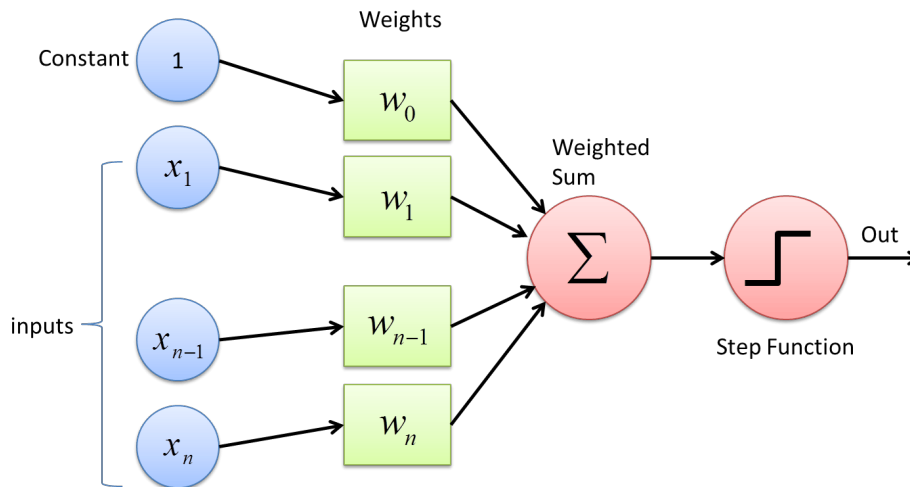


Figure 3.8: The perceptron algorithm

The learning rate comes into play when we update the weights. The learning rate is a hyperparameter that controls the size of the weight updates. A small learning rate will result in small weight updates, while a large learning rate will result in large weight updates. The learning rate is a critical hyperparameter that can affect the convergence of the algorithm. If the learning rate is too small,

the algorithm may take a long time to converge, while if the learning rate is too large, the algorithm may diverge.

The following code shows the perceptron algorithm in action for the `iris` data set focusing on just two of the species. Here, we use functions in base R to lay bare the algorithm.

```
# Perceptron
# Use only two species for simplicity
data(iris)
iris <- iris[iris$Species %in% c("setosa", "versicolor"), ]

# Convert the species column to a numeric -1 or 1
iris$Species <- ifelse(iris$Species == "setosa", -1, 1)

# Add in a constant term
iris <- cbind(1, iris)

# Split the iris data set into training and test sets
set.seed(123)
train_index <- sample(1:nrow(iris), 60)
train_data <- iris[train_index, ]
test_data <- iris[-train_index, ]

# Initialise the weights to zero
weights <- rep(0, ncol(iris) - 1)

# Set the learning rate
alpha <- 0.1

# Set the maximum number of iterations
max_iter <- 100

# Repeat the following steps until the maximum number of
# iterations is reached
for (i in 1:max_iter) {
  # For each input in the training data
  for (j in 1:nrow(train_data)) {
    # Compute the predicted class label
    predicted <- ifelse(sum(weights * train_data[j, 1:5]) > 0,
                        1, -1)

    # Update the weights based on the classification error
    weights <- weights + alpha * (train_data[j, 6] - predicted) *
      train_data[j, 1:5]
  }
}

# Display the fitted weights
weights
```

```
##      1 Sepal.Length Sepal.Width Petal.Length Petal.Width
## 31 -0.2      -0.16      -0.62      1.36      0.66

# Make predictions
predictions <- NULL
for (j in 1:nrow(test_data)) {
  predictions[j] <- ifelse(sum(weights[1,] * test_data[j, 1:5]) > 0,
                           1, -1)
}

# Create the confusion matrix
confusion_matrix <- table(test_data$Specie,
                          predictions)
confusion_matrix

##      predictions
##      -1  1
## -1 20  0
##  1  0 20
```

Here, we have the following formula to classify a new input:

$$\hat{y} = \text{sign}(1.36 \text{ Petal length} + 0.66 \text{ Petal width} \\ - 0.16 \text{ Sepal length} - 0.62 \text{ Sepal width} - 0.2),$$

where $\hat{y}$ is the predicted class label, w_i is the weight of covariate i , x_i is the input covariate, and $\text{sign}(\cdot)$ is the threshold function that returns 1 if the input is greater than zero and -1 otherwise.

The perceptron is a linear classifier, meaning that it can only reliably classify inputs that are linearly separable. In the second half of this module, you will be introduced to the multi-layer perceptron, which is a non-linear classifier that can classify inputs that are not linearly separable and the kernel trick, which is a technique that can be used to transform the input space into a higher-dimensional space where the inputs are linearly separable.

We should also note that we have set up the perceptron with a step *activation* function. This is a binary classifier, but we can extend this to a probabilistic classifier by choosing a different activation function. The most common choice is the sigmoid function, which maps the output to the range $[0, 1]$. (This is the basis of logistic regression.)

Another important extension is to consider multiple classes. The perceptron can be extended to handle multiple classes by using a one-vs-all approach, where a separate perceptron is trained for each class and the class with the highest “score” is selected as the predicted class. The score is computed as the dot product of the weights and the input covariates.

Example

Here is the perceptron in action for the full `iris` data set. This time, we will need to consider how to handle multiple classes.

```
# Perceptron
data(iris)
```

```

# Convert the species column to a numeric -1 or 1
# over three columns
iris$setosa <- ifelse(iris$Species == "setosa", 1, -1)
iris$versicolor <- ifelse(iris$Species == "versicolor", 1, -1)
iris$virginica <- ifelse(iris$Species == "virginica", 1, -1)

# Add in a constant term
iris <- cbind(1, iris)

# Split the iris data set into training and test sets
set.seed(123)
train_index <- sample(1:nrow(iris), 60)
train_data <- iris[train_index, ]
test_data <- iris[-train_index, ]

# Initialise the weights to zero
weights <- NULL
weights[[1]] <- rep(0, ncol(iris) - 4)
weights[[2]] <- rep(0, ncol(iris) - 4)
weights[[3]] <- rep(0, ncol(iris) - 4)

# Set the learning rate
alpha <- 0.1

# Set the maximum number of iterations
max_iter <- 100

predicted <- NULL
# Repeat the following steps until the maximum number of
# iterations is reached
for (i in 1:max_iter) {
  # For each input in the training data
  for (j in 1:nrow(train_data)) {
    # Compute the predicted class label
    for (k in 1:3) {
      predicted[k] <- ifelse(sum(weights[[k]] * train_data[j, 1:5]) > 0,
                             1, -1)
    }
    # Update the weights based on the classification error
    for (k in 1:3) {
      weights[[k]] <- weights[[k]] + alpha *
        (train_data[j, 6 + k] - predicted[k]) * train_data[j, 1:5]
    }
  }
}

# Calculate the scores from each class

```

```

scores <- matrix(0, nrow = nrow(test_data), ncol = 3)
for (j in 1:nrow(test_data)) {
  for (k in 1:3) {
    scores[j, k] <- sum(weights[[k]] * test_data[j, 1:5])
  }
}

# Predict class based on largest score
predictions <- apply(scores, 1, which.max)

# Convert predictions to class labels
class_labels <- c("setosa", "versicolor", "virginica")
predictions <- class_labels[predictions]

# Create the confusion matrix
confusion_matrix <- table(test_data$Species,
                          predictions)

confusion_matrix

```

```

##           predictions
##           setosa versicolor virginica
## setosa           32          0          0
## versicolor       15          3         12
## virginica         0          1         27

```

3.7 Balancing classes

In many classification problems, the classes are imbalanced, meaning that one class has more observations than the other classes. This can lead to biased models that favour the majority class and perform poorly on the minority class. To address this issue, we can balance the classes by using techniques such as oversampling, undersampling, and synthetic data generation.

Here, we will only consider oversampling. Oversampling is a technique that involves duplicating the minority class observations to balance the classes. This can be done by randomly sampling the minority class observations with replacement until the classes are balanced. The oversampling technique can help improve the performance of the model on the minority class and reduce the bias towards the majority class.

Example

Let's consider a synthetic data set with imbalanced classes and use the oversampling technique to balance the classes.

```

library(rpart)

# Synthetic data set with imbalanced classes
set.seed(123)
n <- 100
data <- data.frame(x = rnorm(n),

```

```

y = rnorm(n),
class = sample(c(0, 1), n,
               replace = TRUE,
               prob = c(0.9, 0.1)))

# Visualise the data set
plot(data$x, data$y,
     col = data$class + 1, pch = 19)

```

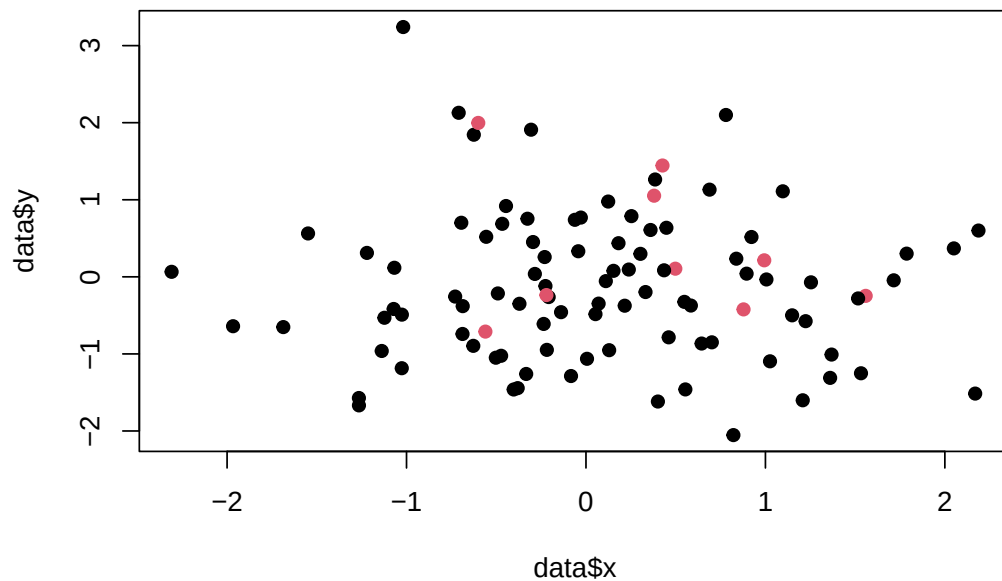


Figure 3.9: The synthetic data set with imbalanced classes.

```

# Fit a decision tree to the data set
decision_tree_model <- rpart(class ~ .,
                             data = data,
                             method = "class")

# Predict the class labels for the data set
predictions <- predict(decision_tree_model,
                       type = "class")

# Create the confusion matrix
confusion_matrix <- table(predictions,
                           data$class)

confusion_matrix

```

```

##
## predictions  0  1
##              0 91  9
##              1  0  0

```

The confusion matrix shows that the model is biased towards the majority class and performs poorly on the minority class. To address this, we can balance the classes using the oversampling technique.

```

# Oversampling
# Balance the classes using the oversampling technique
n_maj <- sum(data$class == 0)
n_min <- sum(data$class == 1)
data_minority <- data[data$class == 1, ]
data_oversampled <- rbind(data,
                          data_minority[sample(1:n_min,
                                                n_maj - n_min,
                                                replace = TRUE), ])

# Count the number in each class before and after
table(data$class)

```

```

##
##  0  1
## 91  9

```

```
table(data_oversampled$class)
```

```

##
##  0  1
## 91 91

```

Now, we can fit a decision tree to the oversampled data set and evaluate the model's performance.

```

# Fit a decision tree to the oversampled data set
decision_tree_model <- rpart(class ~ .,
                             data = data_oversampled,
                             method = "class")

# Predict the class labels for the data set
predictions <- predict(decision_tree_model,
                      newdata = data,
                      type = "class")

# Create the confusion matrix
confusion_matrix <- table(predictions,
                          data$class)

confusion_matrix

```

```

##
## predictions  0  1
##              0 74  1
##              1 17  8

```

Chapter 4

Supervised Learning — Regression

4.1 Historical notes

Regression is an important technique in statistics and machine learning. It was first introduced by Sir Francis Galton in the 19th century. He used it to study the relationship between the heights of fathers and their sons. The term “regression” was coined by Galton to describe the tendency of the heights of sons to “regress” towards the mean height of the population. For us, the general idea is to predict a continuous value from a set of input variables.

As we have already seen, metrics for regression are different from those for classification. Historically, machine learning techniques for regression have focused on getting as close as possible to the target variable with little regard for prediction uncertainty. This is changing as we move towards more complex models that can provide a measure of uncertainty in their predictions.

4.2 Running example

Like in the previous chapter, we will use two well-known data sets to illustrate regression techniques. The first is the Boston housing data set, which contains information about housing in the Boston area. The target variable is the median value of owner-occupied homes in thousands of dollars. The data set contains 506 observations on 14 variables. The second data set is the Auto MPG data set, which contains information about cars. The target variable is the miles per gallon (MPG) of the car. The data set contains 398 observations on 9 variables.

In both cases, it is trivial to set up a linear regression model in R.

```
# load in the data
library(MASS)
data(Boston)
data(mtcars)

# standardise the data to aid interpretation
Boston <- as.data.frame(scale(Boston))
mtcars <- as.data.frame(scale(mtcars))

# linear regression
```

```
lm.fit.Boston <- lm(medv ~ ., data = Boston)
summary(lm.fit.Boston)
```

```
##
## Call:
## lm(formula = medv ~ ., data = Boston)
##
## Residuals:
```

	Min	1Q	Median	3Q	Max
	-1.69559	-0.29680	-0.05633	0.19322	2.84864

```
##
## Coefficients:
```

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-2.833e-15	2.294e-02	0.000	1.000000
crim	-1.010e-01	3.074e-02	-3.287	0.001087 **
zn	1.177e-01	3.481e-02	3.382	0.000778 ***
indus	1.534e-02	4.587e-02	0.334	0.738288
chas	7.420e-02	2.379e-02	3.118	0.001925 **
nox	-2.238e-01	4.813e-02	-4.651	4.25e-06 ***
rm	2.911e-01	3.193e-02	9.116	< 2e-16 ***
age	2.119e-03	4.043e-02	0.052	0.958229
dis	-3.378e-01	4.567e-02	-7.398	6.01e-13 ***
rad	2.897e-01	6.281e-02	4.613	5.07e-06 ***
tax	-2.260e-01	6.891e-02	-3.280	0.001112 **
ptratio	-2.243e-01	3.080e-02	-7.283	1.31e-12 ***
black	9.243e-02	2.666e-02	3.467	0.000573 ***
lstat	-4.074e-01	3.938e-02	-10.347	< 2e-16 ***

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.516 on 492 degrees of freedom
## Multiple R-squared:  0.7406, Adjusted R-squared:  0.7338
## F-statistic: 108.1 on 13 and 492 DF,  p-value: < 2.2e-16
```

```
lm.fit.mpg <- lm(mpg ~ ., data = mtcars)
summary(lm.fit.mpg)
```

```
##
## Call:
## lm(formula = mpg ~ ., data = mtcars)
##
## Residuals:
```

	Min	1Q	Median	3Q	Max
	-0.57254	-0.26620	-0.01985	0.20230	0.76773

```
##
## Coefficients:
```

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	1.436e-15	7.773e-02	0.000	1.0000

```
## cyl      -3.302e-02  3.097e-01  -0.107   0.9161
## disp      2.742e-01  3.672e-01   0.747   0.4635
## hp       -2.444e-01  2.476e-01  -0.987   0.3350
## drat      6.983e-02  1.451e-01   0.481   0.6353
## wt       -6.032e-01  3.076e-01  -1.961   0.0633 .
## qsec      2.434e-01  2.167e-01   1.123   0.2739
## vs       2.657e-02  1.760e-01   0.151   0.8814
## am       2.087e-01  1.703e-01   1.225   0.2340
## gear      8.023e-02  1.828e-01   0.439   0.6652
## carb     -5.344e-02  2.221e-01  -0.241   0.8122
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4397 on 21 degrees of freedom
## Multiple R-squared:  0.869, Adjusted R-squared:  0.8066
## F-statistic: 13.93 on 10 and 21 DF,  p-value: 3.793e-07
```

The summaries here give us the adjusted R-squared, amongst other things. It is interesting to consider the model coefficients in these models and to look at which are significantly different from zero. We can also calculate the other performance measures that we met in the earlier chapter using the fitted models.

	Boston	Auto MPG
MSE	0.26	0.13
RMSE	0.51	0.36
MAE	0.36	0.29
MSLE	NaN	NaN
MAPE	2.51	0.64

You will note that some of these return NaN values. This is because the target variable contains zeros. We can remove these from the data set to avoid this issue or think more carefully about which metric is appropriate for the problem at hand. You will also note that, in isolation, these metrics are not particularly informative. They are useful for comparing models, but not for understanding the quality of the model.

4.3 Regularised Regression

As discussed in the section on overfitting, regularisation is a fundamental technique used to prevent models from becoming overly complex and generalising poorly to new data. The core idea behind regularisation is to add a penalty term to the loss function that discourages large parameter values, effectively trading off some training accuracy for better generalisation performance.

Before going into specific regularisation techniques, it is important to revisit and understand the bias-variance trade-off that underlies regularisation. When we fit a model:

- **Bias** refers to the error introduced by approximating a complex real-world problem with a simplified model
- **Variance** refers to the sensitivity of the model to small changes in the training data

- **Irreducible error** is the noise inherent in the problem that cannot be eliminated

Regularisation techniques deliberately introduce some bias to significantly reduce variance, often resulting in lower total error. This is particularly valuable when we have: - Limited training data, - Many variables relative to observations, - Multicollinearity among predictors.

4.3.1 Types of Regularisation

There are three main types of regularisation for linear regression: L1 (Lasso), L2 (Ridge), and Elastic Net (a combination of both). Each has distinct properties and use cases.

- **Ridge Regression:** Use when you believe most variables are relevant but want to prevent overfitting. Ridge keeps all variables but shrinks their coefficients toward zero.
- **Lasso Regression:** Use when you suspect many variables are irrelevant and want automatic Variable selection. Lasso can set coefficients exactly to zero.
- **Elastic Net:** Use when you have groups of correlated variables. Elastic Net tends to select groups of correlated variables together, unlike Lasso, which may arbitrarily select one from a correlated group.

4.3.2 Mathematical Foundation

Recall that ordinary least squares (OLS) regression finds coefficients by minimising the mean squared error:

$$\text{Loss}_{\text{OLS}} = \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \|\mathbf{y} - X\beta\|_2^2$$

The closed-form solution is:

$$\hat{\beta}_{\text{OLS}} = (X^T X)^{-1} X^T \mathbf{y}$$

When using regularised regression, it is essential to **standardise both the predictors and response variable** because regularisation penalties are sensitive to the scale of variables. Standardisation ensures that penalties are applied equally across all variables.

4.3.2.1 Ridge Regression (L2 Regularisation)

Ridge regression adds an L2 penalty term to the loss function that penalises the sum of squared coefficients:

$$\text{Loss}_{\text{Ridge}} = \|\mathbf{y} - X\beta\|_2^2 + \lambda \|\beta\|_2^2 = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^p \beta_j^2$$

The closed-form solution becomes:

$$\hat{\beta}_{\text{Ridge}} = (X^T X + \lambda I)^{-1} X^T \mathbf{y}$$

Ridge regression is particularly effective at handling multicollinearity, a common issue where two or more predictor variables are highly correlated. In such scenarios, the design matrix X becomes ill-conditioned, meaning the $X^T X$ matrix is close to singular and difficult to invert. This leads to

ordinary least squares (OLS) coefficient estimates that are highly sensitive to small changes in the data, resulting in large variances and instability. By adding the regularisation term λI to the $X^T X$ matrix, ridge regression ensures that the matrix $(X^T X + \lambda I)$ is invertible and well-conditioned, regardless of the correlation among the predictors. This process stabilises the estimates and reduces their variance, producing more reliable and robust coefficients than ordinary least squares in the presence of multicollinearity.

Key Properties: - Never sets coefficients exactly to zero - Shrinks coefficients of correlated variables toward each other - Computationally efficient (closed-form solution) - From a Bayesian perspective, equivalent to placing a Gaussian prior on coefficients

Geometric Interpretation: The L2 constraint creates a circular constraint region. The optimal solution occurs where the elliptical contours of the loss function touch this circular constraint.

4.3.2.2 Lasso Regression (L1 Regularisation)

Lasso (Least Absolute Shrinkage and Selection Operator) regression uses an L1 penalty that penalises the sum of absolute values of coefficients:

$$\text{Loss}_{\text{Lasso}} = \|\mathbf{y} - X\beta\|_2^2 + \lambda \|\beta\|_1 = \sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^p |\beta_j|$$

The optimisation problem is:

$$\hat{\beta}_{\text{Lasso}} = \underset{\beta}{\text{argmin}} \left(\|\mathbf{y} - X\beta\|_2^2 + \lambda \|\beta\|_1 \right)$$

Key Properties: - Can set coefficients exactly to zero (automatic Variable selection), - Tends to select one Variable from a group of correlated variables, - No closed-form solution; requires iterative algorithms (coordinate descent), - From a Bayesian perspective, equivalent to placing a Laplace prior on coefficients.

Geometric Interpretation: The L1 constraint creates a diamond-shaped constraint region. The corners of this diamond lie on the coordinate axes, which is why Lasso can produce exactly zero coefficients.

4.3.3 Elastic Net Regression

Elastic Net combines L1 and L2 penalties, providing a balance between Ridge and Lasso:

$$\text{Loss}_{\text{Elastic Net}} = \|\mathbf{y} - X\beta\|_2^2 + \lambda \left(\alpha \|\beta\|_1 + (1 - \alpha) \|\beta\|_2^2 \right)$$

where $\alpha \in [0, 1]$ controls the balance between L1 and L2 penalties: - $\alpha = 0$: Pure Ridge regression, - $\alpha = 1$: Pure Lasso regression, - $0 < \alpha < 1$: Combination of both penalties.

Key Properties: - Combines Variable selection (L1) with coefficient shrinkage (L2) - Handles groups of correlated variables better than Lasso - Particularly useful when $p > n$ (more variable than observations)

4.3.4 Hyperparameter Selection

The regularisation strength λ (and α for Elastic Net) are hyperparameters that must be chosen carefully. The most common approach is k -fold cross-validation:

1. Split training data into k folds.
2. For each candidate hyperparameter value:
 - Train on $k - 1$ folds,
 - Validate on the remaining fold,
 - Repeat for all folds.
3. Select hyperparameters that minimise average validation error.

Grid Search can be used to search over combinations of hyperparameters systematically. Of course, this can be computationally expensive, especially for large datasets or complex models.

Example

We will demonstrate the strengths and weaknesses of each regularisation method using a carefully constructed synthetic dataset.

```
# Load required libraries
library(glmnet)
library(caret)
library(ggplot2)
library(dplyr)
library(tidyr)
library(gridExtra)

# Set seed for reproducibility
set.seed(42)

# Create a synthetic dataset with known structure
n <- 200 # number of observations
p <- 50  # number of variables

# Generate correlated predictors
X <- matrix(rnorm(n * p), n, p)

# Add correlation structure: variables 1-5 are highly correlated
for(i in 2:5) {
  X[, i] <- 0.7 * X[, 1] + 0.3 * X[, i]
}

# variable 6-10 are moderately correlated
for(i in 7:10) {
  X[, i] <- 0.5 * X[, 6] + 0.5 * X[, i]
}

# True coefficient structure:
```

```

# - variable 1, 6, 11, 16, 21 have large effects (truly important)
# - variable 2-5, 7-10 have small effects (correlated groups)
# - Remaining variables have zero effect (noise)
true_beta <- rep(0, p)
true_beta[c(1, 6, 11, 16, 21)] <- c(3, 2.5, 2, 1.5, 1) # Important variable
true_beta[2:5] <- 0.5 # Correlated with Variable 1
true_beta[7:10] <- 0.3 # Correlated with Variable 6

# Generate response with noise
y <- X %*% true_beta + rnorm(n, 0, 0.5)

# Split into training and test sets
train_idx <- sample(1:n, 0.7 * n)
X_train <- X[train_idx, ]
y_train <- y[train_idx]
X_test <- X[-train_idx, ]
y_test <- y[-train_idx]

# Standardise the data (crucial for regularisation!)
X_train_scaled <- scale(X_train)
X_test_scaled <- scale(X_test,
                      center = attr(X_train_scaled, "scaled:center"),
                      scale = attr(X_train_scaled, "scaled:scale"))
y_train_scaled <- scale(y_train)[, 1]
y_test_scaled <- scale(y_test,
                      center = attr(scale(y_train), "scaled:center"),
                      scale = attr(scale(y_train), "scaled:scale"))[, 1]

# Need to standardise the beta coeffs for later comparison
true_beta_scaled <- true_beta / attr(scale(y_train), "scaled:scale")

# Create combined data frames for training and test sets
X_train_scaled <- as.data.frame(X_train_scaled)
X_test_scaled <- as.data.frame(X_test_scaled)
y_train_scaled <- as.data.frame(y_train_scaled)
y_test_scaled <- as.data.frame(y_test_scaled)
# Rename columns for clarity
colnames(X_train_scaled) <- paste0("X", 1:p)
colnames(X_test_scaled) <- paste0("X", 1:p)
# Combine predictors and response for training and test sets
train_data <- cbind(y_train_scaled, X_train_scaled)
test_data <- cbind(y_test_scaled, X_test_scaled)

cat("Dataset created with:\n")

```

```
## Dataset created with:
```

```

cat("- 5 truly important variable\n")

## - 5 truly important variable
cat("- 2 groups of correlated variable (4 variable each)\n")

## - 2 groups of correlated variable (4 variable each)
cat("- 37 noise variable\n")

## - 37 noise variable
cat("- Training set:", nrow(X_train), "observations\n")

## - Training set: 140 observations
cat("- Test set:", nrow(X_test), "observations\n")

## - Test set: 60 observations

```

For ordinary least squares (OLS) regression, we can use the `lm` function in R. For Ridge, Lasso, and Elastic Net regression, we will use the `glmnet` package. The `glmnet` package provides efficient implementations of these regularised regression techniques.

```

# 1. Ordinary Least Squares (OLS)
ols_model <- lm(y_train_scaled ~ .,
               data = train_data)
ols_pred <- predict(ols_model,
                   newdata = test_data)
ols_mse <- mean((y_test_scaled[,1] - ols_pred)^2)

# 2. Ridge Regression
ridge_cv <- cv.glmnet(x = as.matrix(X_train_scaled),
                    y = y_train_scaled[,1],
                    alpha = 0, nfolds = 10)
ridge_pred <- predict(ridge_cv, s = "lambda.min",
                    newx = as.matrix(X_test_scaled))
ridge_mse <- mean((y_test_scaled[,1] - ridge_pred)^2)
ridge_coef <- coef(ridge_cv, s = "lambda.min")[-1] # Remove intercept

# 3. Lasso Regression
lasso_cv <- cv.glmnet(x = as.matrix(X_train_scaled),
                    y = y_train_scaled[,1], alpha = 1, nfolds = 10)
lasso_pred <- predict(lasso_cv, s = "lambda.min",
                    newx = as.matrix(X_test_scaled))
lasso_mse <- mean((y_test_scaled[,1] - lasso_pred)^2)
lasso_coef <- coef(lasso_cv, s = "lambda.min")[-1] # Remove intercept

# 4. Elastic Net Regression
elastic_grid <- expand.grid(alpha = seq(0.1, 0.9, by = 0.1),
                          lambda = seq(0.001, 0.1, length = 20))

```

Table 4.2: Model Performance Comparison

Method	Test_MSE	Non_zero_coef	Lambda
OLS	0.0145	50	NA
Ridge	0.0222	50	0.0745
Lasso	0.0127	18	0.0103
Elastic Net	0.0127	19	0.0114

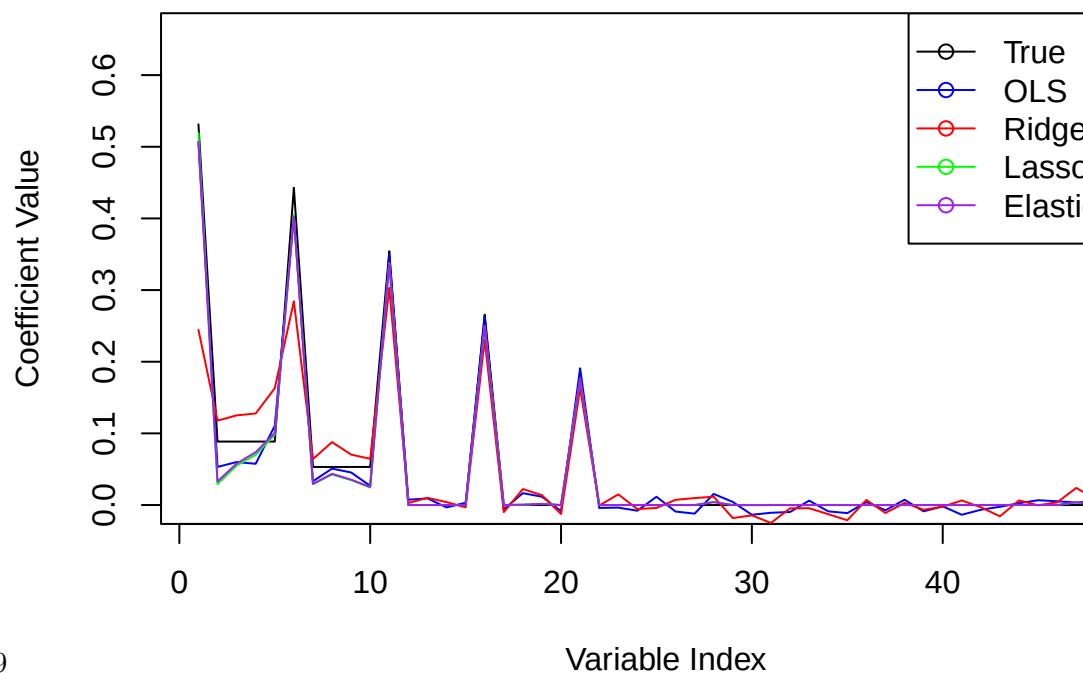
```

elastic_model <- train(x = as.matrix(X_train_scaled),
                      y = y_train_scaled[,1],
                      method = "glmnet",
                      tuneGrid = elastic_grid,
                      trControl = trainControl(method = "cv", number = 10),
                      standardise = FALSE) # Already standardised

elastic_pred <- predict(elastic_model, X_test_scaled)
elastic_mse <- mean((y_test_scaled[,1] - elastic_pred)^2)
elastic_coef <- coef(elastic_model$finalModel, s = elastic_model$bestTune$lambda)[-1]

```

Coefficient Comparison



Elastic Net optimal alpha: 0.9

With the ridge regression coefficients, we can see the effect of the regularisation being too strong. The coefficients are shrunk towards zero, but not set to zero. This is the expected behaviour of ridge regression. The lasso regression coefficients, on the other hand, show that some coefficients are set to zero, which is the expected behaviour of lasso regression. The elastic net coefficients show a combination of both ridge and lasso regression, with some coefficients set to zero and others shrunk towards zero.

Example

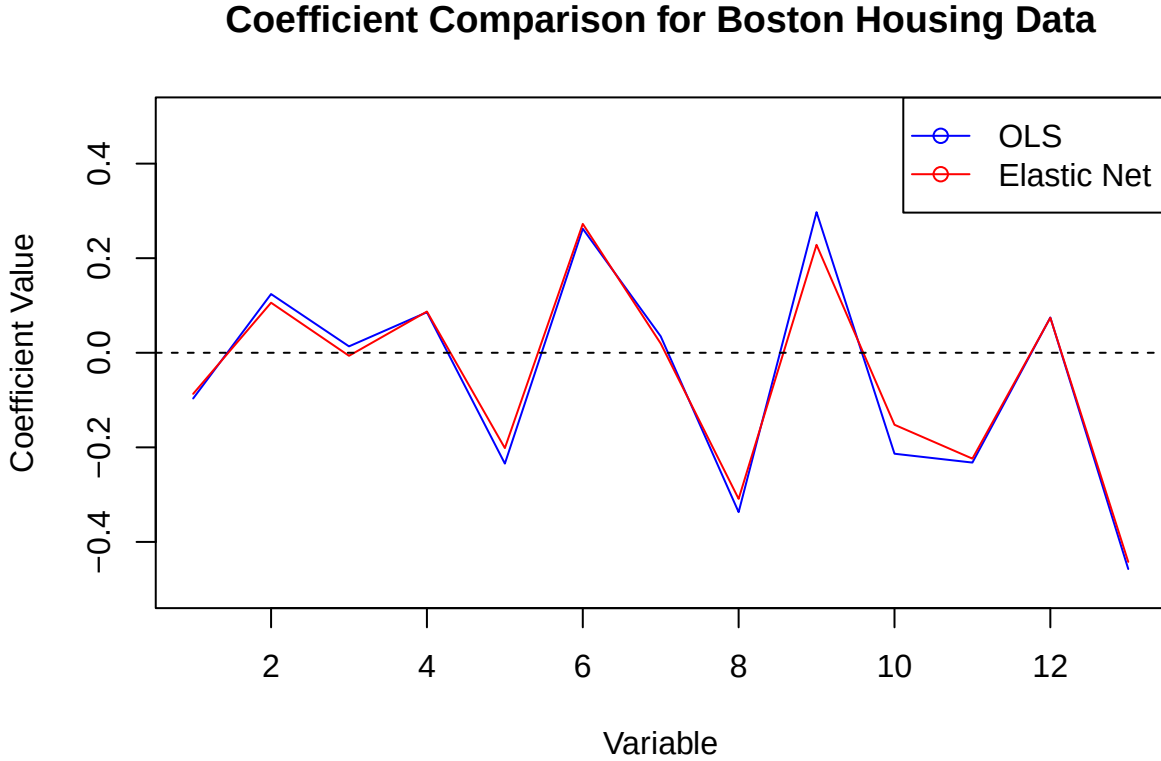
We will now fit a regularised regression model to the Boston housing data set again using the `glmnet` package.

```
# Load the Boston data set
library(MASS)
data(Boston)
# Standardise the data
Boston <- as.data.frame(scale(Boston))
# Set up train and test sets
set.seed(123)
trainIndex <- sample(1:nrow(Boston), 0.8 * nrow(Boston))
trainBoston <- Boston[trainIndex,]
testBoston <- Boston[-trainIndex,]

# OLS for coeff comparison
ols.fit.Boston <- lm(medv ~ ., data = trainBoston)
ols.pred <- predict(ols.fit.Boston, newdata = testBoston)
ols.MLE <- mean((ols.pred - testBoston$medv)^2)

# Elastic Net Regression
elastic_grid <- expand.grid(alpha = seq(0.1, 0.9, by = 0.1),
                           lambda = seq(0.001, 0.1, length = 20))
elastic.fit.Boston <- train(medv ~ .,
                           data = trainBoston,
                           method = "glmnet",
                           tuneGrid = elastic_grid,
                           trControl = trainControl(method = "cv", number = 10),
                           standardise = FALSE) # Already standardised
elastic.pred <- predict(elastic.fit.Boston, newdata = testBoston)
elastic.MSE <- mean((elastic.pred - testBoston$medv)^2)
```

In this case, we get a regularisation parameter of 0.0166 and an optimal alpha of 0.1. The MSE for the elastic net model is 0.28, which is slightly lower than the MSE for the OLS model of 0.28.



4.4 *k*-nearest neighbours

Like in the classification setting, we can use *k*-nearest neighbours for regression. The idea is exactly the same: we predict the target variable by averaging the values of the *k*-nearest neighbours. The updated algorithm is given by

1. Calculate the distance between the test point and all the training points.
2. Sort the distances and determine the *k*-nearest neighbours.
3. Average the target variable of the *k*-nearest neighbours.

Again, the choice of *k* is important in *k*-nearest neighbours. A small value of *k* will result in a model that is sensitive to noise, while a large value of *k* will result in a model that is biased. The choice of *k* will depend on the problem at hand and the amount of noise in the data. More formally, the mathematical representation of the *k*-nearest neighbours algorithm is

$$\hat{y} = \frac{1}{k} \sum_{i=1}^k y_i$$

where $\hat{y}$ is the predicted value, *k* is the number of neighbours, and y_i is the target variable of the *i*-th neighbour.

Step 2 of the algorithm can be computationally challenging with large training sets. The `knn` package in R uses a data structure called a *k-d tree* to speed up the calculation of the distances between the test point and the training points. The *k-d tree* is a binary tree that is used to partition the input space into regions. The *k-d tree* is built by recursively splitting the input space along the dimensions of the input variables. The *k-d tree* is used to find the *k*-nearest neighbours by traversing the tree and only considering the regions that contain the *k*-nearest neighbours.

The final predictions can be thought of as a piecewise constant function over the input space defined by the training data. The different prediction values sit on a Voronoi tessellation whose tiles are determined by the training data. The idea behind k -nearest neighbours is related to the idea of stacking that we will meet in the next section. Stacking uses the predictions of multiple models to make a final prediction. In the case of k -nearest neighbours, the prediction “models” are the k -nearest neighbours themselves.

Example

We fit a k -nearest neighbours model to the Boston data set using the `knn` package within `caret`.

```
set.seed(123)
# Set up train and test sets
trainIndex <- sample(1:nrow(Boston), 0.8 * nrow(Boston))
trainBoston <- Boston[trainIndex,]
testBoston <- Boston[-trainIndex,]

# Fit the model
knn.fit.Boston <- train(medv ~ .,
  data = Boston,
  method = "knn",
  trControl = trainControl(method = "cv",
    number = 10))

# Predictions
knn.pred <- predict(knn.fit.Boston, testBoston)
```

Figure 4.3 shows the model performance in terms of the actual and predicted values.

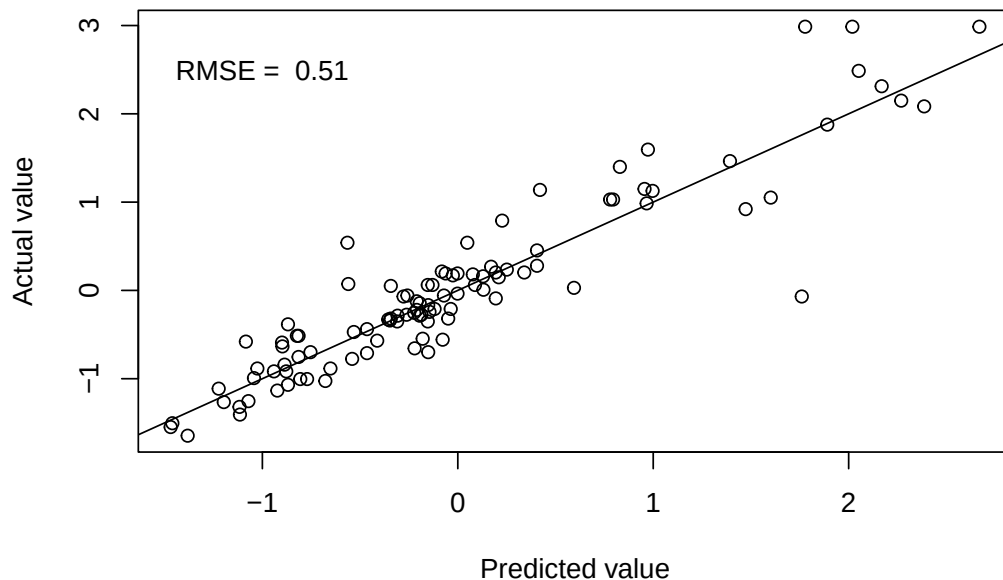


Figure 4.1: k -nearest neighbours predictions

We can extend the k -nearest neighbours model to include a weighting term in the prediction. The weighting term is a function of the distance between the test point and the training points. The

idea is to give more weight to the training points that are closer to the test point. The weighted k -nearest prediction is calculated using

$$\hat{y} = \frac{\sum_{i=1}^k w_i y_i}{\sum_{i=1}^k w_i}$$

where w_i is the weight of the i -th neighbour. The weights are typically calculated using the inverse of the distance between the test point and the training points. The weighted k -nearest neighbours model is implemented in the `kkn` package in R.

```
set.seed(123)
# Fit the model
kkn.fit.Boston <- train(medv ~ .,
  data = Boston,
  method = "kkn",
  trControl = trainControl(method = "cv",
    number = 10))

# Predictions
kkn.pred <- predict(kkn.fit.Boston, testBoston)
```

Figure 4.4 shows the model performance in terms of the actual and predicted values utilising the distance weights.

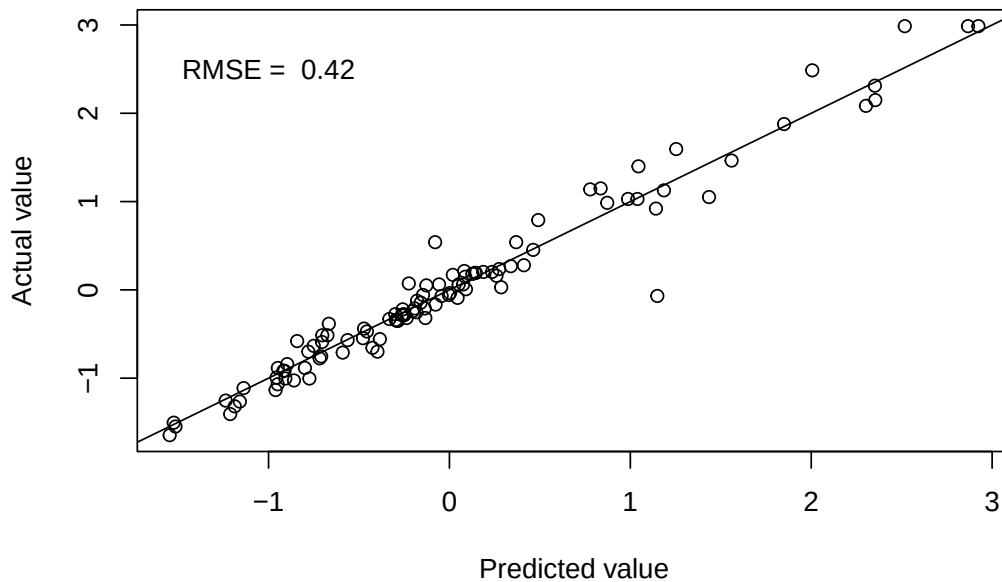


Figure 4.2: Weighted k -nearest neighbours predictions

4.5 Decision trees

Decision trees provide yet another non-parametric regression technique that is based on recursive partitioning of the input space that work the same way as in the classification setting. The idea is to split the input space into regions and fit a simple model to each region. The model is trained by recursively splitting the input space into regions until a stopping criterion is met. The stopping

criterion can be based on the number of observations in a region, the depth of the tree, or the impurity of the regions. The *impurity* of a region is a measure of how well the target variable is predicted by the model in that region. The impurity can be measured using the mean squared error, the mean absolute error, or the mean squared logarithmic error.

In the standard implementation, the model is simply the mean of the target variable in the region. The final model is therefore a multidimensional step function over the space of explanatory variables.

Example

Let's see it in action for the `mtcars` data set.

```
set.seed(123)
# Set up train and test sets
trainIndex <- createDataPartition(mtcars$mpg, p = 0.8,
list = FALSE)
trainMtcars <- mtcars[trainIndex,]
testMtcars <- mtcars[-trainIndex,]

# Fit the model
tree.fit.mtcars <- train(mpg ~ .,
data = mtcars,
method = "rpart",
trControl = trainControl(method = "cv",
number = 10))

# Predictions
tree.pred <- predict(tree.fit.mtcars,
testMtcars)
```

Figures 4.5 and 4.6 show the model performance in terms of the actual and predicted values and the fitted decision tree. The model here is extremely simple with only two splits. Essentially, the model provides just three values for the mean behaviour of the target variable. This is to be expected given the small number of observations in the data set.

```
# Plot the tree
library(rpart.plot)
rpart.plot(tree.fit.mtcars$finalModel)
```

4.6 Multivariate Adaptive Regression Splines

The Multivariate Adaptive Regression Spline (MARS) method is a non-parametric regression technique that is based on piecewise linear functions. MARS models are built by recursively partitioning the input space into regions and fitting a linear function to each region. This should sound familiar as it is the same idea as decision trees. The difference is that the MARS model is built from simple linear functions, while the decision tree model is built from constant functions.

We write the MARS model in terms of hinge functions, which are piecewise linear functions that are

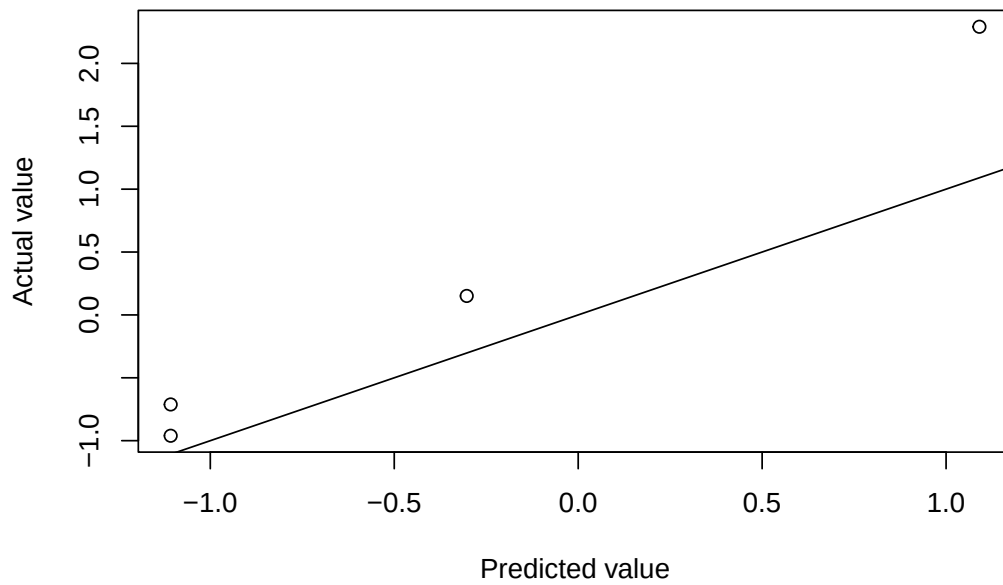


Figure 4.3: Decision tree predictions

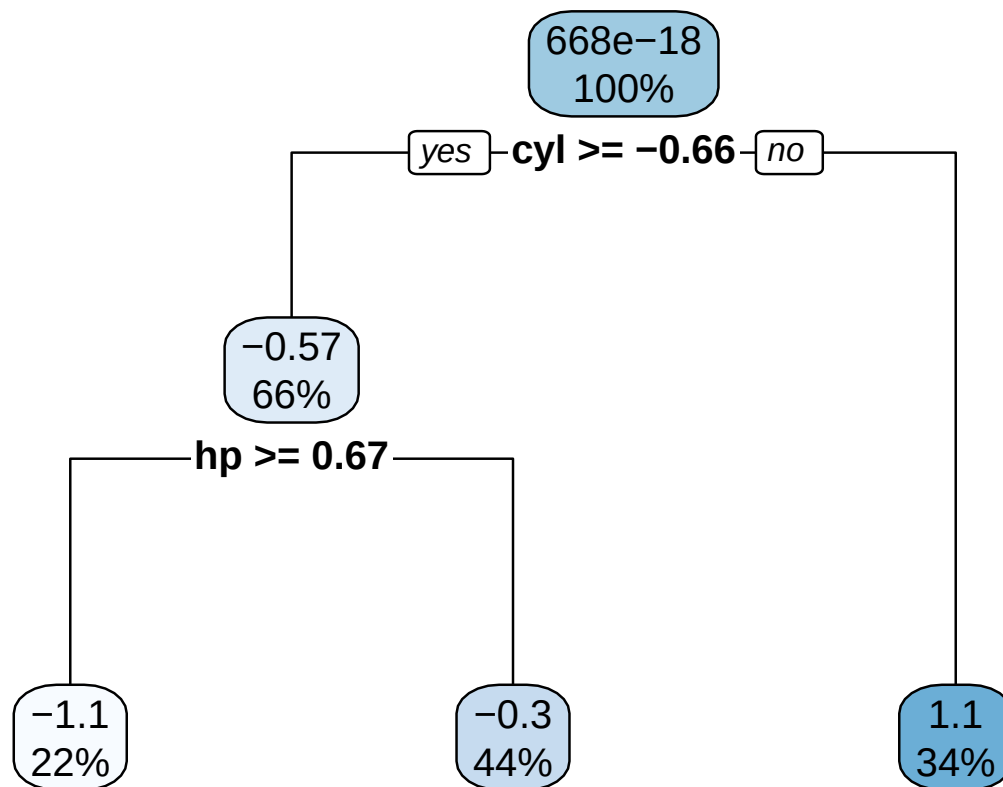


Figure 4.4: Fitted decision tree

used to create the piecewise linear functions in the MARS model:

$$f(\mathbf{x}) = \beta_0 + \sum_{j=1}^m \beta_j h_j(\mathbf{x}),$$

where the β_j are the model parameters and the $h_j(\mathbf{x})$ are the hinge functions. The hinge functions are defined as

$$h_j(\mathbf{x}) = \max(0, x_k - c_j) \quad \text{or} \quad \max(0, c_j - x_k),$$

where x_k is one of the input variables and c_j is the knot point. The “or” here allows us to have functions with negative or positive gradients. The MARS model is built by adding hinge functions to the model and selecting the most important hinge functions. This means that we need to estimate the location of the knot points and the coefficients of the hinge functions. In practice, the knot locations are determined in a manner akin to the decision tree algorithm.

The MARS model is often trained by considering reductions in RMSE. At each stage of the algorithm, the model is trained by adding a hinge function that reduces the RMSE the most. The algorithm is stopped when the RMSE is no longer reduced by adding a hinge function.

The MARS model has the advantage of being interpretable because the model is built from simple linear functions. Figure 4.7 shows an example of the MARS model in one dimension.

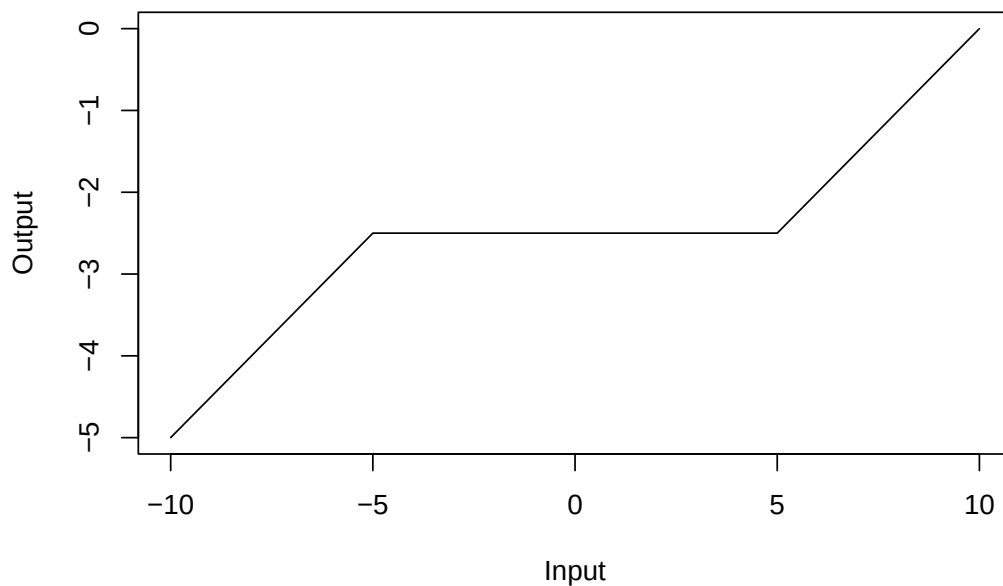


Figure 4.5: MARS model in one dimension

Example

We fit a MARS model to the Boston data set using the `earth` package within `caret`.

```
library(earth)

set.seed(123)
mars.fit.Boston <- train(medv ~ .,
  data = Boston,
```

```

method = "earth",
trControl = trainControl(method = "cv",
number = 10),
tuneGrid = expand.grid(degree = 1:2,
nprune = 2:10))

summary(mars.fit.Boston)

## Call: earth(x=matrix[506,13], y=c(0.1595,-0.101...), keepxy=TRUE, degree=2,
##          nprune=10)
##
##
##               coefficients
## (Intercept)          0.1549343
## h(rm-0.208315)        0.9102939
## h(-1.15797-dis)       15.7464951
## h(lstat- -0.914859)   -0.5632729
## h(1.36614-nox) * h(lstat- -0.914859)    0.1849062
## h(0.206892-rm) * h(dis- -1.15797)       -0.1600934
## h(rm-0.208315) * h(ptratio-0.0667298)   -1.3843533
## h(-0.612548-tax) * h(-0.914859-lstat)    3.8341243
## h(tax- -0.612548) * h(-0.914859-lstat)   2.8353130
##
## Selected 9 of 27 terms, and 6 of 13 predictors (nprune=10)
## Termination condition: Reached nk 27
## Importance: rm, lstat, ptratio, tax, dis, nox, crim-unused, zn-unused, ...
## Number of terms at each degree of interaction: 1 3 5
## GCV 0.137542    RSS 63.93938    GRSq 0.8627298    RSq 0.8733874

```

In the model summary, the `h()` functions are the hinge functions that are used to create the piecewise linear functions. For instance, `h(rm-0.208315)` with a coefficient of 0.91 should be interpreted as the hinge function $\max(0, rm - 0.208315)$, which can be visualised as

Similarly, `h(-1.15797-dis)` with a coefficient of 15.7 should be interpreted as the hinge function $\max(0, -1.15797 - dis)$:

The MARS model has an RMSE of 0.41. In the model summary, we can see that we have selected 9 of 27 terms, and 6 of the 13 predictors have been used. This means that the MARS model has selected 9 hinge functions for the final model that utilise 6 of the 13 explanatory variables. Like in the standard linear regression case, `rm`, `dis` and `lstat` are important predictors in the MARS model.

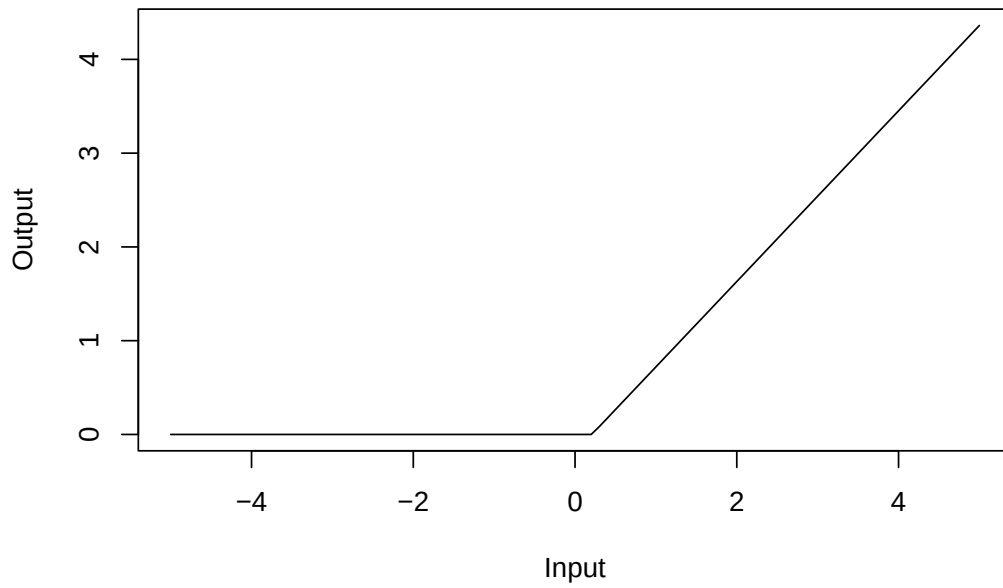


Figure 4.6: Hinge function for 'rm' variable

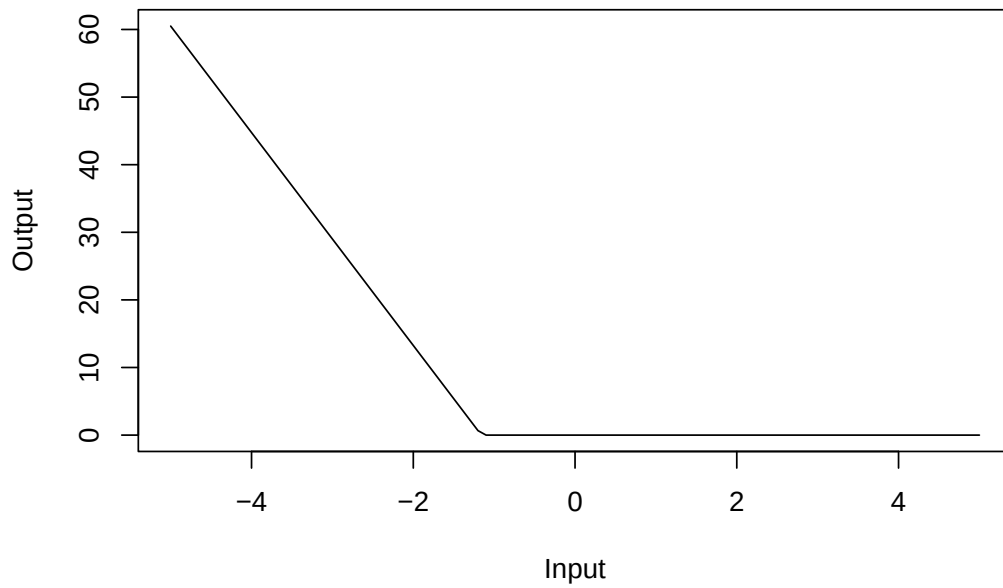


Figure 4.7: Hinge function for 'dis' variable

Chapter 5

Ensemble methods

It is often said that two heads are better than one. This principle also applies to machine learning, where combining multiple models can lead to better performance than any individual model. Ensemble methods leverage the diversity of multiple models to create a more robust and accurate prediction. In this chapter, we will explore some popular ensemble methods including stacking, bagging, random forests, and boosting. Again, we will be using the same running examples as from the previous chapters to illustrate these concepts.

Let's first consider some of the mathematics behind why ensemble methods work. The key idea is that combining multiple models can help reduce bias and variance, leading to better generalisation performance. This is known as the bias-variance trade-off, which we met earlier. Mathematically, the expected prediction error of a model can be decomposed into three components:

$$\begin{aligned}\text{Expected Prediction Error} &= \text{Bias}^2 + \text{Var} \\ &\quad + \text{Irreducible Error}.\end{aligned}$$

When we utilise two models f_1 and f_2 to make predictions, the expected prediction error of the ensemble can be expressed as:

$$\begin{aligned}\text{Expected Prediction Error} &= \text{Bias}^2 + \text{Var} + \text{Irreducible Error} \\ &\quad + \text{constant} * \text{Covariance}(f_1, f_2).\end{aligned}$$

The covariance term captures the relationship between the predictions of the two models. If the models are highly correlated, the covariance term will be positive, and the ensemble may not benefit much from combining them. However, if the models are uncorrelated or negatively correlated, the covariance term will act to reduce the variance in the prediction, and the ensemble can achieve a lower expected prediction error than any individual model.

Let's concentrate on the more statistical idea of model averaging. If our ensemble prediction is of the form:

$$f_{\text{Ensemble}}(X) = \alpha f_1(X) + (1 - \alpha) f_2(X),$$

where $\alpha \in [0, 1]$ is a weight that determines the contribution of each model. The variance of the ensemble prediction with respect to unknown explanatory variables can be expressed as:

$$\begin{aligned}\text{Var}(f_{\text{Ensemble}}(X)) &= \text{Var}(\alpha f_1(X) + (1 - \alpha)f_2(X)) \\ &= \alpha^2 \text{Var}(f_1(X)) + (1 - \alpha)^2 \text{Var}(f_2(X)) \\ &\quad + 2\alpha(1 - \alpha)\text{Cov}(f_1(X), f_2(X)).\end{aligned}$$

Here, if $f_1(x) = f_2(x)$, we return the variance in predictions from that single model. If we have two models that are uncorrelated, the covariance term will be zero, and the variance of the ensemble prediction will be lower than the variance of the individual models. This is the key idea behind ensemble methods: by combining multiple models that are diverse and uncorrelated, we can reduce the variance of the ensemble prediction and improve the overall (average) performance of the model. However, we limit the ability of the ensemble to capture more extreme and complex patterns in the data.

5.1 Weak and strong learners

In the context of ensemble learning, a *weak learner* is a model that performs slightly better than random guessing on a given task. Weak learners are typically simple models that have limited predictive power but can still provide some useful information. Examples of weak learners include decision stumps (decision trees with a single split), linear models, and shallow neural networks.

In contrast, a *strong learner* is a model that performs well above random guessing on a given task. Strong learners have high predictive power and can accurately capture complex patterns in the data. Examples of strong learners include deep neural networks, gradient-boosted trees, and support vector machines (which you will learn about next term).

5.2 Stacking

Stacking, also known as stacked generalisation, is a technique in machine learning that combines the predictions from multiple models to create a new, improved model. It is a type of ensemble method that leverages the strengths of various models to achieve better performance than any individual model could on its own.

Here's a breakdown of how stacking works:

1. Train multiple *base* models: You start by training several different machine learning models on the same dataset. These models can be of different types.
2. Make predictions with base models: Use each base model to make predictions on a separate hold-out validation set (data not used for training). This gives you multiple predictions for each data point in the validation set.
3. Train a meta-model: Now, you train a new model, called a meta-model or stacking model, using the predictions from the base models as features. The target variable for the meta-model is the actual value in the validation set.
4. Final predictions: The meta-model learns how to best combine the predictions from the base models to create a final prediction. This final prediction is often expected to be more accurate than the predictions from any of the individual base models.

Stacking essentially allows you to harness the strengths of different models and potentially address their weaknesses. By combining their insights, the final model can achieve a more robust and generalisable performance.

Figure 5.1 illustrates the concept of stacking from the point of view of making predictions.

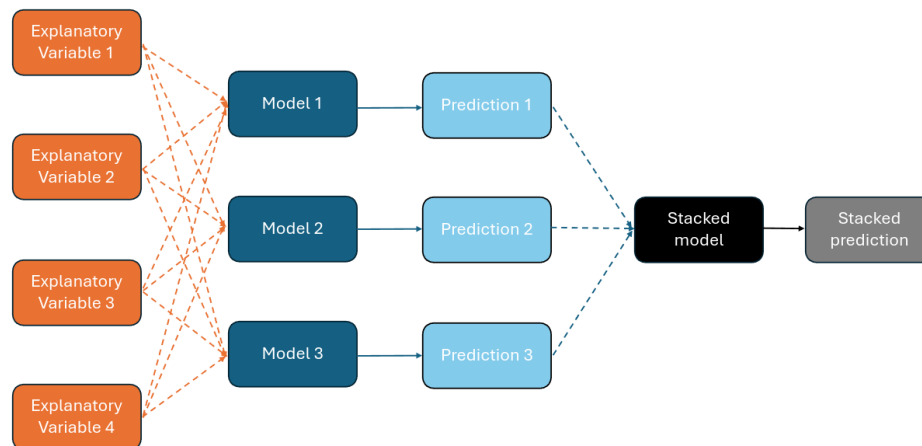


Figure 5.1: Combining models to make stacked predictions

5.2.1 Stacking of classifiers

In this example, we will demonstrate how to implement stacking in R. We will train multiple base models on the `iris` dataset and then use a meta-model to combine their predictions.

```

# Load the required libraries
library(caret)
library(e1071)

# Load the iris dataset
data(iris)

# Create a training and validation set using the sample function
set.seed(123)
train_index <- sample(1:nrow(iris), 0.8 * nrow(iris))
train <- iris[train_index, ]
test <- iris[-train_index, ]

# Define the base models
# (multinomial regression, kNN and the naive Bayes classifier)
model1 <- train(Species ~ ., data = train,
                 method = "multinom", trace = FALSE)
model2 <- train(Species ~ ., data = train,
                 method = "knn")
model3 <- train(Species ~ ., data = train,
                 method = "naive_bayes")

```

```

# Make predictions on the validation set
pred1 <- predict(model1, newdata = test)
pred2 <- predict(model2, newdata = test)
pred3 <- predict(model3, newdata = test)

# Combine the predictions into a data frame alongside the actual values
stacked_data <- data.frame(pred1, pred2, pred3,
                           Species = test$Species)

# Train a meta-model (decision trees) on the stacked data
meta_model <- train(Species ~ ., data = stacked_data,
                    method = "rpart")

# Make predictions with the meta-model
final_pred <- predict(meta_model,
                      newdata = stacked_data)

# Evaluate the performance of the meta-model
confusionMatrix(test$Species, final_pred)$table

```

```

##           Reference
## Prediction  setosa versicolor virginica
##   setosa      10         0         0
##   versicolor   1        14         0
##   virginica    5         0         0

```

As a contrast, we also show the confusion matrices from the three separate models:

```
confusionMatrix(test$Species, pred1)$table
```

```

##           Reference
## Prediction  setosa versicolor virginica
##   setosa      10         0         0
##   versicolor   0        14         1
##   virginica    0         0         5

```

```
confusionMatrix(test$Species, pred2)$table
```

```

##           Reference
## Prediction  setosa versicolor virginica
##   setosa      10         0         0
##   versicolor   0        14         1
##   virginica    0         0         5

```

```
confusionMatrix(test$Species, pred3)$table
```

```

##           Reference
## Prediction  setosa versicolor virginica
##   setosa      10         0         0
##   versicolor   0        14         1

```

```
## virginica      0      0      5
```

So, the k -nearest neighbours model is the best one and as good as the stacked model in its own right. There is also an issue with overfitting here again because we are training the meta model on the same data as we are testing it on. This is a common issue with stacking and can be addressed by using cross-validation to train the meta-model or by simply splitting the data into three parts: training, validation, and testing.

Let's rerun the analysis with a 60-20-20 split:

[illegible]

```
# Evaluate the performance of the meta-model
confusionMatrix(test$Species, final_pred)$table
```

```
##                Reference
## Prediction      setosa versicolor virginica
##   setosa         9         0         0
##   versicolor     11         0         1
##   virginica       0         0         9
```

Again, we can compare the performance of the stacked model with the individual models for the test data.

```
confusionMatrix(test$Species,
                 predict(model1, newdata = test))$table
```

```
##                Reference
## Prediction      setosa versicolor virginica
##   setosa         9         0         0
##   versicolor      0        11         1
##   virginica        0         0         9
```

```
confusionMatrix(test$Species,
                 predict(model2, newdata = test))$table
```

```
##                Reference
## Prediction      setosa versicolor virginica
##   setosa         9         0         0
##   versicolor      0        12         0
##   virginica        0         0         9
```

```
confusionMatrix(test$Species,
                 predict(model3, newdata = test))$table
```

```
##                Reference
## Prediction      setosa versicolor virginica
##   setosa         9         0         0
##   versicolor      0        12         0
##   virginica        0         1         8
```

The stacked model does not look so good this time because it is better to use the k -nearest neighbours model on its own. The plot of the fitted decision tree in the stacked model is shown below:

Interestingly, the second model is completely ignored in the decision tree. This is a common issue with stacking, where the meta-model can sometimes ignore the predictions from certain base models if they are not contributing to the overall performance. The reason for this can be seen in the confusion matrices for the validation set:

```
confusionMatrix(validation$Species, pred1)$table
```

```
##                Reference
## Prediction      setosa versicolor virginica
##   setosa         8         0         0
```

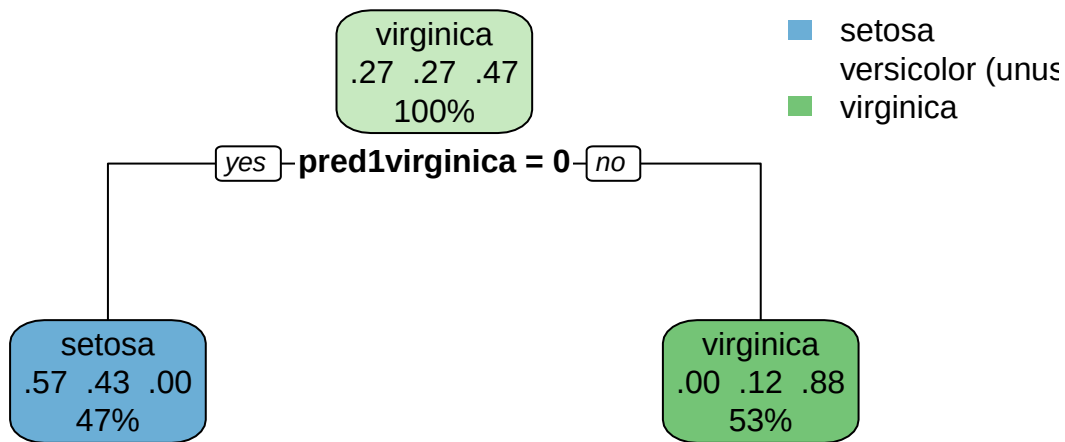


Figure 5.2: Fitted decision tree for the stacked model

```
##   versicolor      0      6      2
##   virginica      0      0     14
confusionMatrix(validation$Species, pred2)$table
```

```
##           Reference
## Prediction  setosa versicolor virginica
##   setosa      8      0      0
##   versicolor  0      6      2
##   virginica  0      0     14
```

```
confusionMatrix(validation$Species, pred3)$table
```

```
##           Reference
## Prediction  setosa versicolor virginica
##   setosa      8      0      0
##   versicolor  0      7      1
##   virginica  0      2     12
```

The second model is performing in precisely the same way as the first model, so the meta-model has no reason to use it. The third model is at least handling the `versicolor` class better than the first two models, so it is used in the meta-model.

5.3 Bagging

Bagging is an ensemble method that aims to reduce the variance of a machine learning model and improve its overall stability. It achieves this by training multiple models, called base learners, on different subsets of the data and then aggregating their predictions. The name “bagging” is short for “bootstrap aggregating,” as it uses bootstrapping to create the data subsets.

5.3.1 Bootstrapping

Bootstrapping is a resampling technique used in statistics. It is often mistaken for a method that generates psuedo-data-points, but it is actually a way to create multiple datasets from the original

data by sampling with replacement. This allows us to estimate the uncertainty of a statistic without the need for additional data collection. There is an assumption that the original data is representative of the population, and the bootstrapped samples are representative of the original data.

The bootstrapped sample is the same size as the original data. A bootstrapped estimate of θ say is

$$\bar{\theta} = \frac{1}{B} \sum_{b=1}^B \hat{\theta}_{(b)},$$

where $\hat{\theta}_{(b)}$ is the estimate of θ from the b th bootstrapped sample and B is the total number of bootstrapped samples.

As we are simply sampling with replacement, we can work out the expected proportion of data points that are not used in any bootstrapped sample. Let

$$I_i = 1(\text{Observation } i \text{ appears at least once in the bootstrapped sample}).$$

and

$$U = \sum_{i=1}^n I_i,$$

where n is the total number of observations in the original data. The expected value of U is

$$\begin{aligned} E(U) &= \sum_{i=1}^n E(I_i) = \sum_{i=1}^n P(I_i = 1) \\ &= \sum_{i=1}^n \left[1 - \left(1 - \frac{1}{n} \right)^n \right] \\ &= n \left[1 - \left(1 - \frac{1}{n} \right)^n \right]. \end{aligned}$$

Hence, the proportion of training observations missing from the bootstrapped sample is

$$\frac{n - E(U)}{n} = \left(1 - \frac{1}{n} \right)^n.$$

Now, as $n \rightarrow \infty$, we have

$$\lim_{n \rightarrow \infty} \left(1 - \frac{1}{n} \right)^n = e^{-1} \approx 0.368.$$

This means that, on average, about 36.8% of the original data points will not be included in any given bootstrapped sample. These excluded data points are often referred to as “out-of-bag” samples and can be used for model evaluation.

Bagging is a prediction version of this where multiple models are fitted; here’s how bagging works:

1. Create bootstrap samples: Bagging starts by generating multiple random samples (with replacement) from the original training data. These samples, called bootstrap replicates, are of the same size as the original data but may contain duplicates of some data points and exclude others.
2. Train base learners in parallel: Each bootstrap replicate is used to train a separate base learner independently. These base learners can be any type of model, but decision trees are commonly used with bagging.

3. Aggregate predictions: After training all the base learners, their predictions are combined. For regression tasks, the predictions are typically averaged. In classification tasks, a majority vote is often used, where the class with the most votes from the base learners becomes the final prediction.

By training on various data subsets and aggregating their predictions, bagging reduces the influence of any single data point on the final model. This helps to address overfitting and leads to a more generalisable model that performs well on unseen data. Bagging has a random component due to the bootstrapping process, which adds diversity to the base learners and improves the overall performance of the ensemble.

Example

Let's consider a bagged lasso regression model using the `boot` package in R. We will use the `Boston` dataset to demonstrate bagging with a lasso regression model.

```
# Load the Boston dataset
data(Boston)

# Create a training and test set using the sample function
set.seed(123)

train_index <- sample(1:nrow(Boston), 0.8 * nrow(Boston))
train <- Boston[train_index, ]
test <- Boston[-train_index, ]

# Define the base model (lasso regression)
library(glmnet)
base_model <- function(data, indices) {
  fit <- glmnet(x = as.matrix(data[indices, -14]), y = data$medv[indices],
               family = "gaussian", alpha = 1, lambda = 0.1)
  return(fit$beta)
}

# Train multiple base models using bootstrapping and store each set of coefficients
coefficients <- NULL
for (i in 1:10) {
  bootstrapped_indices <- sample(1:nrow(train), replace = TRUE)
  coefficients[[i]] <- base_model(train, bootstrapped_indices)
}

# Convert list into a matrix where each row represents the coefficients of a base model
coefficients_matrix <- t(do.call(cbind, coefficients))

# Average the coefficients to get the final model
final_coefficients <- as.matrix(colMeans(coefficients_matrix),
                                length(coefficients[[1]]))
round(final_coefficients, 2)
```

```
##      [,1]
## crim  -0.05
## zn     0.03
## indus -0.03
## chas   3.61
## nox    -9.60
## rm     3.98
## age    0.00
## dis    -1.03
## rad     0.12
## tax    0.00
## ptratio -0.89
## black  0.01
## lstat  -0.57
```

Let's compare that with the coefficients of a regular lasso regression model:

```
round(base_model(train, 1:nrow(train)),2)

## 13 x 1 sparse Matrix of class "dgCMatrix"
##      s0
## crim  -0.07
## zn     0.03
## indus -0.01
## chas   3.03
## nox    -13.40
## rm     3.69
## age    .
## dis    -1.20
## rad     0.15
## tax    0.00
## ptratio -0.92
## black  0.01
## lstat  -0.57
```

Which model performs better for the test data set?

```
# Make predictions on the test set using the final model
predictions <- as.matrix(as.matrix(test[, -14]) %*% final_coefficients)

# Calculate the mean squared error
mean((predictions - test$medv)^2)
```

```
## [1] 875.6266
```

```
# Make predictions on the test set using the base model
base_predictions <- as.matrix(as.matrix(test[, -14]) %*% base_model(train, 1:nrow(train)))

# Calculate the mean squared error
mean((base_predictions - test$medv)^2)
```

```
## [1] 1157.368
```

5.3.2 Random forests

Random forests are an ensemble learning method that builds multiple decision trees during training and outputs the mode of the classes (classification) or the average prediction (regression) of the individual trees. The method is very much related to bagged decision trees. However, random forests introduce extra randomness into the tree-building process to make the model more robust. Here's how random forests work:

1. Create bootstrap samples: Random forests start by generating multiple random samples (with replacement) from the original training data. These samples, called bootstrap replicates, are used to train individual decision trees.
2. Random feature selection: When building each decision tree, a random subset of features is selected as candidates for splitting at each node. This introduces diversity into the trees and helps to prevent overfitting.
3. Grow multiple trees: Multiple decision trees are grown using the bootstrap samples and random feature selection. Each tree is trained independently of the others.
4. Aggregate predictions: For regression tasks, the predictions from all the trees are averaged to get the final prediction. For classification tasks, the mode of the classes predicted by the individual trees is taken as the final prediction.
5. Out-of-bag evaluation: Random forests use out-of-bag (OOB) samples, which are data points not included in the bootstrap sample used to train each tree. These OOB samples can be used to estimate the model's performance without the need for a separate validation set.

Random forests are known for their robustness, scalability and ability to handle high-dimensional data. By combining the predictions of multiple trees trained on different subsets of the data, random forests can produce accurate and stable models for a wide range of machine learning tasks.

Out-of-bag samples are data points that are not included in the bootstrap sample used to train a particular tree. These samples can be used to estimate the performance of the model without the need for a separate validation set. The out-of-bag error is calculated by aggregating the predictions of the trees that did not use a particular data point in their training. This provides an unbiased estimate of the model's performance on unseen data and can be useful for small datasets where cross-validation may be impractical.

Figures 5.3 and 5.4 illustrate the two phases of employing random forests: the training phase and the prediction phase.

Example

In R, we can use the `randomForest` package to build a random forest model. Here's an example using the `iris` dataset:

```
# Load the required library
library(randomForest)

# Load the iris dataset
data(iris)
```

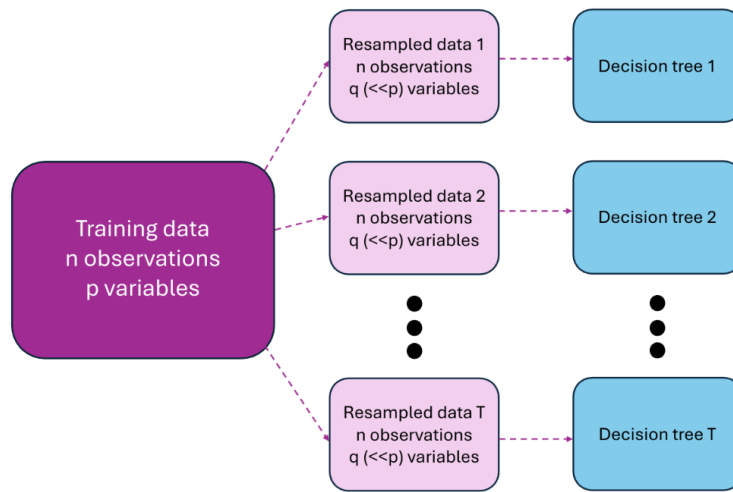


Figure 5.3: Training of random forests

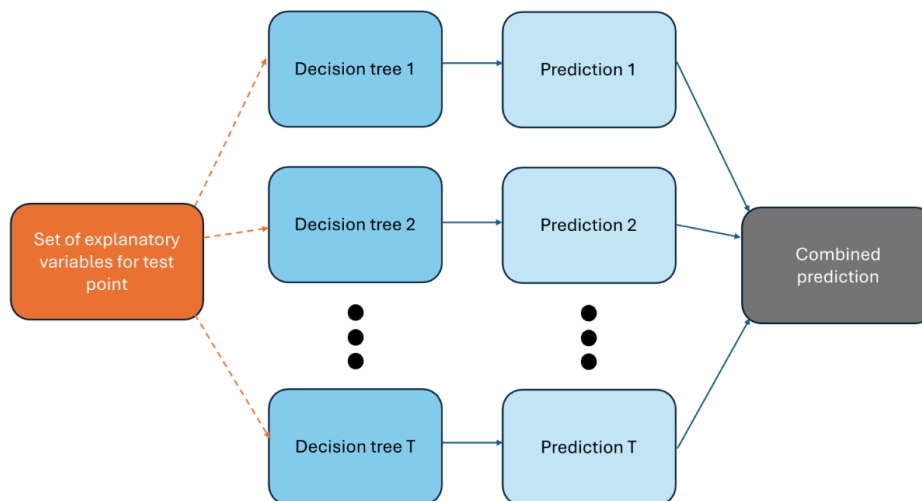


Figure 5.4: Predicting with random forests

```

# Create a training and test set using the sample function
set.seed(123)

train_index <- sample(1:nrow(iris), 0.8 * nrow(iris))
train <- iris[train_index, ]
test <- iris[-train_index, ]

# Train a random forest model
rf_model <- randomForest(Species ~ .,
                        data = train,
                        ntree = 500)

# Make predictions on the test set
rf_predictions <- predict(rf_model, newdata = test)

# Evaluate the performance of the model
confusionMatrix(test$Species, rf_predictions)$table

```

```

##              Reference
## Prediction  setosa versicolor virginica
##   setosa      10         0         0
##   versicolor   0        14         1
##   virginica    0         0         5

```

Let's compare that with bagged decision trees:

```

# Train a bagged decision tree model
bagged_model <- randomForest(Species ~ ., data = train,
                            ntree = 500,
                            mtry = ncol(train) - 1)

# Make predictions on the test set
bagged_predictions <- predict(bagged_model,
                             newdata = test)

# Evaluate the performance of the model
confusionMatrix(test$Species, bagged_predictions)$table

```

```

##              Reference
## Prediction  setosa versicolor virginica
##   setosa      10         0         0
##   versicolor   0        14         1
##   virginica    0         0         5

```

Notice that the R function `randomForest` can be used to build both random forests and bagged decision trees. The difference between the two lies in the `mtry` parameter, which controls the number of features randomly selected at each split. In random forests, `mtry` is set to the square root of the total number of features by default, while in bagged decision trees, it is set to the total number of features.

Example

It is instructive to code a random forest (almost) from scratch. Here is an example of a random forest for the iris dataset using the `rpart` package:

```
# Load the required library
library(rpart)

# Load the iris dataset
data(iris)

# Create a training and test set using the sample function
set.seed(123)

train_index <- sample(1:nrow(iris), 0.8 * nrow(iris))
train <- iris[train_index, ]
test <- iris[-train_index, ]

# Train multiple decision trees
ntrees <- 500
trees <- list()

for (i in 1:ntrees) {
  # Create a bootstrap sample
  bootstrap <- sample(1:nrow(train), replace = TRUE)
  train_bootstrap <- train[bootstrap, ]

  # Randomly select a subset of features for each tree
  train_bootstrap <- train_bootstrap[,
    c(sample(1:(ncol(train) - 1),
      size = floor(sqrt(ncol(train) - 1))),
      ncol(train))]

  # Train a decision tree on the bootstrap sample
  tree <- rpart(Species ~ ., data = train_bootstrap, method = "class")
  trees[[i]] <- tree
}

# Make predictions on the test set
predictions <- matrix(0, nrow = nrow(test), ncol = ntrees)

for (i in 1:ntrees) {
  predictions[, i] <- predict(trees[[i]],
    newdata = test,
    type = "class")
}

# Aggregate the predictions
```

```

final_predictions <- apply(predictions, 1, function(x) {
  names(sort(table(x), decreasing = TRUE)[1])
})

# Convert the predictions of "1", "2", "3" to
# "setosa", "versicolor", "virginica"
final_predictions_ <- factor(final_predictions,
                             levels = c("1", "2", "3"),
                             labels = c("setosa",
                                         "versicolor",
                                         "virginica"))

# Evaluate the performance of the model
confusionMatrix(test$Species, final_predictions_)$table

##           Reference
## Prediction  setosa versicolor virginica
##   setosa      10          0          0
##   versicolor   0          14          1
##   virginica    0           0          5

```

In the “Aggregate the predictions” step, we use a majority vote to determine the final prediction for each data point. The class with the most votes from the individual trees is chosen as the final prediction. This is a common approach in random forests and helps to reduce the variance of the model. You might find it helpful to play with the function in the `apply` function to see how the predictions are determined.

5.3.2.1 Optimisation for random forest fitting

The `randomForest` package in R provides a number of parameters that can be tuned to optimise the performance of the model. Some of the key parameters include:

- **ntree**: The number of trees to grow in the forest. Increasing the number of trees can improve the model’s performance but also increases the computational cost.
- **mtry**: The number of features randomly selected at each split. A smaller **mtry** value can increase the diversity of the trees and reduce overfitting.
- **nodesize**: The minimum size of terminal nodes. A larger **nodesize** value can prevent the trees from growing too deep and overfitting the data.
- **sampsiz**: The number of samples randomly selected for each tree. A smaller **sampsiz** value can increase the diversity of the trees and reduce overfitting.

By tuning these parameters, you can optimise the performance of the random forest model for your specific dataset and task. Random forests are fitted using individually optimised decision trees as discussed in the section on decision trees in Chapter 3.

5.4 Boosting

Boosting is yet another powerful ensemble method in machine learning that focuses on improving the accuracy of predictions by sequentially training weak learners into a single, strong learner. Unlike

bagging, which trains base learners in parallel, boosting trains them sequentially, with each new learner learning from the mistakes of its predecessors. Here's how boosting works:

1. Train weak learners iteratively: Boosting starts by training a weak learner on the entire dataset. A weak learner is a simple model that performs only slightly better than random guessing.
2. Focus on errors: After the first weak learner is trained, the algorithm identifies the data points that the model incorrectly predicted. These points are then given higher importance (weight) in the next iteration.
3. Train subsequent learners: A new weak learner is trained using the adjusted weights, focusing on the previously misclassified examples. This process continues iteratively, with each new learner trying to improve upon the performance of the previous ones.
4. Combine predictions: Finally, the predictions from all the weak learners are combined, typically using a weighted voting scheme where better-performing learners have more influence. The combined prediction from the ensemble is the final output of the boosting model.

By strategically focusing on the hard-to-learn examples and leveraging a team of weak learners, boosting helps create a more robust and accurate final model compared to any single weak learner.

5.4.1 AdaBoost

AdaBoost (Adaptive Boosting) is one of the most popular boosting algorithms that has been widely used in practice. It was introduced by Yoav Freund and Robert Schapire in 1996. AdaBoost works by combining multiple weak learners to create a strong learner. The key idea behind AdaBoost is to assign weights to the training examples and adjust these weights at each iteration to focus on the misclassified examples.

In R, we can use the `adabag` package to implement AdaBoost. Here's an example using the `iris` dataset:

```
# Load the required library
library(adabag)

# Load the iris dataset
data(iris)

# Create a training and test set using the sample function
set.seed(123)

train_index <- sample(1:nrow(iris), 0.8 * nrow(iris))
train <- iris[train_index, ]
test <- iris[-train_index, ]

# Train an AdaBoost model
ada_model <- boosting(Species ~ ., data = train,
                      boos = TRUE, mfinal = 500)

# Make predictions on the test set
ada_predictions <- predict(ada_model, newdata = test)

# Convert the predictions in ada_predictions$class to a factor
```

```

ada_predictions$class <- factor(ada_predictions$class,
                                levels = c("setosa",
                                             "versicolor",
                                             "virginica"))

# Evaluate the performance of the model
confusionMatrix(test$Species,
                 ada_predictions$class)$table

```

```

##           Reference
## Prediction  setosa versicolor virginica
##   setosa      10         0         0
##   versicolor   0        14         1
##   virginica    0         0         5

```

The function `boosting` from the `adabag` package is used to train an AdaBoost model. The `boos` parameter is set to `TRUE` to indicate that AdaBoost should be used. The `mfinal` parameter specifies the maximum number of iterations (weak learners) to train.

5.4.2 XGBoost

XGBoost (Extreme Gradient Boosting) is an advanced implementation of the gradient boosting algorithm that has gained popularity in recent years. Developed by Tianqi Chen and Carlos Guestrin in 2016, XGBoost is known for its speed, performance, and scalability. It has been widely used in machine learning competitions and real-world applications.

In R, we can use the `xgboost` package to implement XGBoost. Here's an example using the `iris` dataset:

```

# Load the required library
library(xgboost)

# Load the iris dataset
data(iris)

iris_out <- iris
iris_out$Species <- as.numeric(iris_out$Species) - 1

# Create a training and test set using the sample function
set.seed(123)

train_index <- sample(1:nrow(iris_out),
                      0.8 * nrow(iris_out))
train <- iris_out[train_index, ]
test <- iris_out[-train_index, ]

# Convert the data to a DMatrix object
dtrain <- xgb.DMatrix(data = as.matrix(train[, -5]),
                      label = train$Species)

```

```

dtest <- xgb.DMatrix(data = as.matrix(test[, -5]),
                    label = test$Species)

# Train an XGBoost model
params <- list(
  objective = "multi:softmax",
  num_class = 3
)
xgb_model <- xgb.train(
  params = params,
  data = dtrain,
  nrounds = 500
)

# Make predictions on the test set
xgb_predictions <- predict(xgb_model, dtest)

# Convert the predictions to original class labels
xgb_predictions <- ifelse(xgb_predictions == 0, "setosa",
                        ifelse(xgb_predictions == 1,
                              "versicolor", "virginica"))
xgb_predictions <- factor(xgb_predictions,
                        levels = c("setosa",
                                  "versicolor",
                                  "virginica"))

# Evaluate the performance of the model
# Confusion matrix
table(test$Species,
      xgb_predictions)

##      xgb_predictions
##      setosa versicolor virginica
##  0         10          0          0
##  1          0         14          1
##  2          0          0          5

```

The conversion of the data to a `DMatrix` object is required for training an XGBoost model because the algorithm is optimised for this data structure. The `nrounds` parameter specifies the number of boosting rounds (iterations) to train. The `objective` parameter specifies the loss function to be optimised, and `num_class` specifies the number of classes in the dataset. In this example, we use the “softmax” objective for multiclass classification. Mathematically, the softmax function is defined as:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^N e^{x_j}}$$

where x_i is the input to the function and N is the number of classes. The rationale behind using the softmax function is to convert the raw output of the model into probabilities that sum to one.

As discussed earlier, this translates a deterministic output into a probability that the data point belongs to a particular class.

5.5 Error-Correcting Output Codes

Error-Correcting Output Codes (ECOC) is a technique used in multiclass classification to improve the performance of an ensemble of binary classifiers. The idea behind ECOC is to represent each class as a unique binary code and train multiple binary classifiers to distinguish between the different classes based on these codes.

Here's how ECOC works in the training phase:

1. **Code Matrix Construction:** Construct a code matrix where each row represents a class and each column represents an independent binary classifier. The entries in the matrix are typically -1, 0, or 1, where -1 and 1 indicate the two classes for the binary classifier, and 0 indicates that the classifier does not consider that class.
2. **Binary Classifier Training:** We train the binary classifiers using the binary codes as the target variable.

And in the prediction phase:

1. **Prediction:** For a new instance, each binary classifier makes a prediction resulting in a sequence of -1 and 1 (and possibly 0).
2. **Codeword Matching:** The predicted binary vector is compared to the codewords in the code matrix. The class with the closest codeword is selected as the predicted class.

For the codeword matching, the Hamming distance is often used as a measure of similarity between the predicted binary vector and the codewords. The class with the codeword that has the smallest Hamming distance to the predicted vector is chosen as the final prediction.

Example

Let's consider an example where we have three classes. If M1 denotes the first binary classifier, for example, our code matrix could look like this:

Class	M1	M2	M3
A	-1	-1	0
B	1	0	-1
C	0	1	1

We could add further redundancy to the code matrix to improve the performance of the model.

Class	M1	M2	M3	M4	M5	M6
A	-1	-1	0	1	1	0
B	1	0	-1	0	1	1
C	0	1	1	-1	0	-1

Imagine that we have a new instance that is predicted by the binary classifiers as:

M1	M2	M3	M4	M5	M6
-1	1	1	1	0	-1

The Hamming distance between this vector and the codewords in the code matrix is:

Class	Hamming Distance
A	4
B	6
C	2

Therefore, the predicted class would be C.

Example

We can use base R to implement Error-Correcting Output Codes after we have fitted our individual models. Here's an example using the `iris` dataset and three separate decision tree classifiers:

```
library(rpart)
library(rpart.plot)
library(MASS)

# Load the iris dataset
data(iris)

# Encode classes as binary codes for the six models
iris$o1 <- as.factor(ifelse(iris$Species == "setosa", 1,
                           ifelse(iris$Species == "versicolor",
                                   -1, 0)))
iris$o2 <- as.factor(ifelse(iris$Species == "setosa", 1,
                           ifelse(iris$Species == "virginica",
                                   -1, 0)))
iris$o3 <- as.factor(ifelse(iris$Species == "versicolor", 1,
                           ifelse(iris$Species == "virginica",
                                   -1, 0)))

iris$o4 <- iris$o1
iris$o5 <- iris$o2
iris$o6 <- iris$o3

# Set-up the code matrix corresponding to these binary classifiers
code_matrix <- matrix(c(1, 1, 0, 1, 1, 0,
                       -1, 0, 1, 1, 0, 1,
                       0, -1, -1, 0, -1, -1),
                     nrow = 3, byrow = TRUE)

# Create a training and test set using the sample function
set.seed(123)
```

```

train_index <- sample(1:nrow(iris), 0.8 * nrow(iris))
train <- iris[train_index, ]
test <- iris[-train_index, ]

# Train multiple binary classifiers
M1 <- rpart(o1 ~ Sepal.Length+Sepal.Width+
            Petal.Length+Petal.Width,
            data = train[train$o1 != "0",], method = "class")
M2 <- rpart(o2 ~ Sepal.Length+Sepal.Width+
            Petal.Length+Petal.Width,
            data = train[train$o2 != "0",], method = "class")
M3 <- rpart(o3 ~ Sepal.Length+Sepal.Width+
            Petal.Length+Petal.Width,
            data = train[train$o3 != "0",], method = "class")
M4 <- lda(o4 ~ Sepal.Length+Sepal.Width+
            Petal.Length+Petal.Width,
            data = train[train$o4 != "0",])
M5 <- lda(o5 ~ Sepal.Length+Sepal.Width+
            Petal.Length+Petal.Width,
            data = train[train$o5 != "0",])
M6 <- lda(o6 ~ Sepal.Length+Sepal.Width+
            Petal.Length+Petal.Width,
            data = train[train$o6 != "0",])

# Make predictions on the test set
pred1 <- predict(M1, newdata = test, type = "class")
pred2 <- predict(M2, newdata = test, type = "class")
pred3 <- predict(M3, newdata = test, type = "class")
pred4 <- predict(M4, newdata = test)$class
pred5 <- predict(M5, newdata = test)$class
pred6 <- predict(M6, newdata = test)$class

# Combine the predictions into a data frame
stacked_data <- data.frame(pred1, pred2, pred3,
                           pred4, pred5, pred6)

# Compare each row of the stacked data to the code matrix
# and calculate the Hamming distance
hamming_distances <- apply(stacked_data, 1, function(x) {
  apply(code_matrix, 1, function(y) sum(x != y))
})

# Determine the predicted class based on the
# minimum Hamming distance
predicted_classes <- apply(hamming_distances, 2, function(x) {
  which.min(x)
})

```

```
# Convert the predicted classes to the original class labels
predicted_classes <- factor(ifelse(predicted_classes == 1,
                                   "setosa",
                                   ifelse(predicted_classes == 2,
                                           "versicolor", "virginica")),
                           levels = c("setosa",
                                       "versicolor", "virginica"))

# Evaluate the performance of the model
table(test$Species,
      predicted_classes)
```

```
##           predicted_classes
##           setosa versicolor virginica
##  setosa           10           0           0
##  versicolor        0          13           2
##  virginica         0           0           5
```

Let's compare the performance of the ECOC model with a regular random forest model:

```
# Train a random forest model
rf_model <- randomForest(Species ~ Sepal.Length+Sepal.Width+
                        Petal.Length+Petal.Width,
                        data = train, ntree = 500)

# Make predictions on the test set
rf_predictions <- predict(rf_model, newdata = test)

# Evaluate the performance of the model
table(test$Species,
      rf_predictions)
```

```
##           rf_predictions
##           setosa versicolor virginica
##  setosa           10           0           0
##  versicolor        0          14           1
##  virginica         0           0           5
```

With far fewer models, we have achieved a similar performance to the random forest model.

5.6 Other ensemble methods

There are many other ensemble methods in machine learning that leverage the power of combining multiple models to improve performance. Some other popular ensemble methods include:

- **Voting:** In voting, multiple models are trained independently, and their predictions are combined using a majority vote (classification) or averaging (regression).
- **Blending:** Blending is similar to stacking but uses a separate validation set to train the meta-model.

- **Bayesian model averaging:** Bayesian model averaging combines the predictions of multiple models using Bayesian inference to estimate the model weights.
- **Hierarchical ensembles:** Hierarchical ensembles combine multiple levels of ensembles to create a more complex model.

Each ensemble method has its strengths and weaknesses, and the choice of method depends on the specific problem at hand.

The question remains as to why we would use ensemble methods and why we don't just utilise the best model. The answer is that ensemble methods can often outperform individual models by leveraging the diversity of multiple models. It is often easier to get a richer model by combining simple models rather than trying to build a complex model from scratch. By combining the predictions of different models, ensemble methods can reduce the risk of overfitting, improve the robustness of the model, and achieve better generalisation on unseen data.

Chapter 6

Model Interpretation

A general criticism of machine learning methods is that they are often black boxes: it is difficult to disentangle the influence of one variable from another. This is particularly true for complex models such as random forests and gradient boosting machines where very many simple models have been averaged or aggregated. However, there are several methods that can be used to interpret the results of these models. Model interpretability is crucial in many domains, especially healthcare, finance, and legal applications, where understanding the reasoning behind predictions is often as important as the predictions themselves. Regulatory requirements and ethical considerations often mandate explainable models.

There are several mechanistic insights that we may want to have given a set of data and a model:

1. **Variable importance:** Which variables are most important in predicting the outcome?
2. **Variable interactions:** Are there interactions between variables that are important in predicting the outcome?
3. **Local interpretability:** Given a single observation, can we understand why the model made the prediction it did?
4. **Model structure:** What does the model look like? What are the decision rules that the model is using?

In this chapter, we will explore several methods for interpreting the results of a model. We will use the DALEX package to do this. This package provides a unified interface for interpreting the results of a wide variety of models (akin to `caret` for cross-validation).

6.1 Variable importance

In a simple linear regression model, the coefficients of the model give us a direct measure of the importance of each variable. For example, consider the following linear regression model:

```
lm_model <- lm(mpg ~ ., data = mtcars)

summary(lm_model)
```

```
##
## Call:
```

```
## lm(formula = mpg ~ ., data = mtcars)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.4506 -1.6044 -0.1196  1.2193  4.6271
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 12.30337    18.71788   0.657   0.5181
## cyl         -0.11144     1.04502  -0.107   0.9161
## disp         0.01334     0.01786   0.747   0.4635
## hp          -0.02148     0.02177  -0.987   0.3350
## drat         0.78711     1.63537   0.481   0.6353
## wt          -3.71530     1.89441  -1.961   0.0633 .
## qsec         0.82104     0.73084   1.123   0.2739
## vs           0.31776     2.10451   0.151   0.8814
## am           2.52023     2.05665   1.225   0.2340
## gear         0.65541     1.49326   0.439   0.6652
## carb        -0.19942     0.82875  -0.241   0.8122
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.65 on 21 degrees of freedom
## Multiple R-squared:  0.869, Adjusted R-squared:  0.8066
## F-statistic: 13.93 on 10 and 21 DF,  p-value: 3.793e-07
```

The coefficients of the model give us a direct measure of the importance of each variable. For example, the coefficient of `wt` is -3.72, which means that for every one unit increase in `wt`, the predicted value of `mpg` decreases by 3.72.

It is also straightforward to determine the strength of interactions between variables.

```
lm_model <- lm(mpg ~ . + cyl:disp, data = mtcars)
summary(lm_model)

##
## Call:
## lm(formula = mpg ~ . + cyl:disp, data = mtcars)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.1697 -1.6096 -0.1275  1.1873  3.8355
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 29.976395  18.535141   1.617   0.1215
## cyl         -1.789619   1.183617  -1.512   0.1462
## disp        -0.095947   0.049001  -1.958   0.0643 .
```

```
## hp          -0.033409    0.020359   -1.641    0.1164
## drat        -0.541227    1.584761   -0.342    0.7363
## wt          -3.552721    1.717760   -2.068    0.0518 .
## qsec        0.698111    0.664203    1.051    0.3058
## vs          0.828745    1.918957    0.432    0.6705
## am          0.819051    1.997640    0.410    0.6862
## gear        1.554511    1.405425    1.106    0.2818
## carb        0.144212    0.764824    0.189    0.8523
## cyl:disp     0.013762    0.005825    2.363    0.0284 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.401 on 20 degrees of freedom
## Multiple R-squared:  0.8976, Adjusted R-squared:  0.8413
## F-statistic: 15.94 on 11 and 20 DF,  p-value: 1.441e-07
```

For example, the coefficient of `cyl:disp` is 0.01, which means that for every simultaneous unit increase in `cyl` and `disp`, the predicted value of `mpg` increases by 0.01.

6.1.1 Permutation-based variable importance

One way to determine the importance of each variable in a random forest or gradient boosting machine is to use a permutation-based variable importance method. If we are estimating performance metric θ , this method works as follows:

1. Fit the model to the data and calculate the model's performance, $\hat{\theta}_0$.
2. For each variable in turn ($j = 1, \dots, p$) and for multiple repeats ($i = 1, \dots, M$)
 - randomly permute the values of that variable in the data;
 - calculate the change in the model's performance; i.e., $\hat{\theta}_0 - \hat{\theta}_j^{(i)}$;
3. The importance of each variable is the average change in the model's performance across all permutations:

$$\text{Importance}(j) = \frac{1}{M} \sum_{i=1}^M \left(\hat{\theta}_0 - \hat{\theta}_j^{(i)} \right).$$

If a variable is important in predicting the outcome, then permuting the values of that variable should result in a large change in the model's performance. Conversely, if a variable is not important in predicting the outcome, then permuting the values of that variable should result in a small change in the model's performance.

The algorithm briefly outlined here is a one-at-a-time method. It is possible that such a method may miss important interactions between variables. There are other methods that can be used to determine the importance of interactions between variables.

Example

Here, we will look at this strategy for a linear regression model and consider its utility in contrast to the more traditional hypothesis tests. Again, instead of relying on a package, we will implement this method ourselves.

```

# Fit the model
lm_model <- lm(mpg ~ ., data = mtcars)

# Hypothesis test results for each variable
summary(lm_model)

##
## Call:
## lm(formula = mpg ~ ., data = mtcars)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.4506 -1.6044 -0.1196  1.2193  4.6271
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 12.30337    18.71788   0.657   0.5181
## cyl         -0.11144     1.04502  -0.107   0.9161
## disp         0.01334     0.01786   0.747   0.4635
## hp          -0.02148     0.02177  -0.987   0.3350
## drat         0.78711     1.63537   0.481   0.6353
## wt          -3.71530     1.89441  -1.961   0.0633 .
## qsec         0.82104     0.73084   1.123   0.2739
## vs          0.31776     2.10451   0.151   0.8814
## am          2.52023     2.05665   1.225   0.2340
## gear         0.65541     1.49326   0.439   0.6652
## carb        -0.19942     0.82875  -0.241   0.8122
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.65 on 21 degrees of freedom
## Multiple R-squared:  0.869, Adjusted R-squared:  0.8066
## F-statistic: 13.93 on 10 and 21 DF,  p-value: 3.793e-07

# Calculate the performance of the model
performance <- summary(lm_model)$r.squared

# Permute the values of each variable in turn
permuted_performance <- sapply(names(mtcars)[-1]),
  function(var) {
    # Permute the values of the variable
    permuted_data <- mtcars
    permuted_data[[var]] <- sample(permuted_data[[var]])

    # Fit the model to the permuted data
    permuted_lm_model <- lm(mpg ~ ., data = permuted_data)

    # Calculate the performance of the model

```

```

    permuted_performance <- summary(permuted_lm_model)$r.squared

    # Return the change in performance
    return(performance - permuted_performance)
  })

# Plot the results
barplot(permuted_performance,
        names.arg = names(mtcars[,-1]),
        las = 2)

```

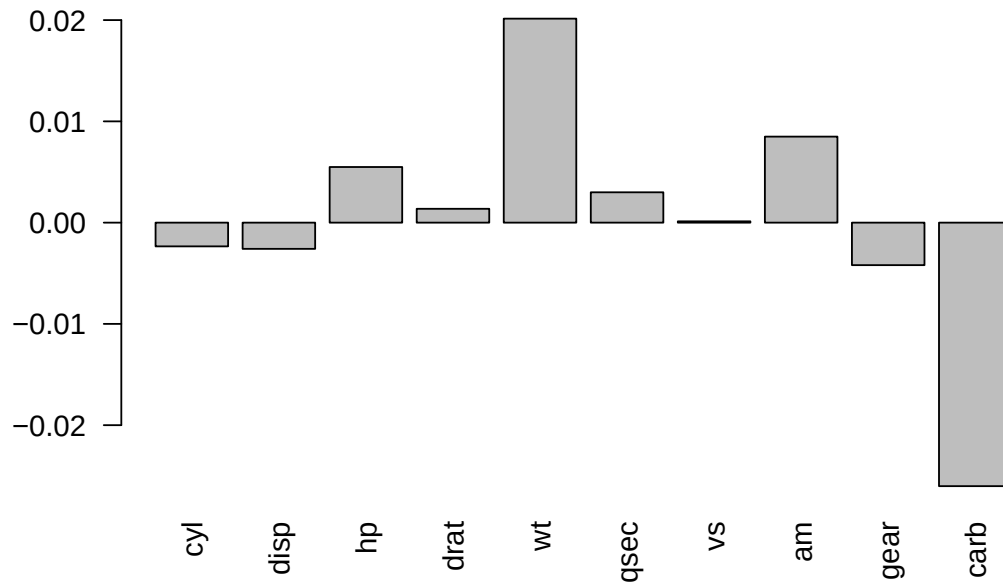


Figure 6.1: The change in performance of the model when the values of each variable are permuted.

There are, of course, packages available in R that will implement this method for you. One such package is DALEX.

Example

Now, we will utilise the DALEX package to determine the importance of each variable in a random forest model for the Boston housing data.

```
library(ISLR2)
```

```
##
## Attaching package: 'ISLR2'

## The following object is masked from 'package:MASS':
##
## Boston
```

```
library(DALEX)
```

```
## Registered S3 method overwritten by 'DALEX':
```

```
##      method          from
##      print.description questionr

## Welcome to DALEX (version: 2.5.3).
## Find examples and detailed introduction at: http://ema.drwhy.ai/
## Additional features will be available after installation of: ggpubr.
## Use 'install_dependencies()' to get all suggested dependencies

##
## Attaching package: 'DALEX'

## The following object is masked from 'package:dplyr':
##
##      explain
```

```
# Fit the random forest model
rf_model <- randomForest(medv ~ ., data = Boston)

# Create an explainer object
explainer <- explain(rf_model,
                     data = Boston[-14],
                     y = Boston$medv)
```

```
## Preparation of a new explainer is initiated
##   -> model label      : randomForest ( default )
##   -> data             : 506 rows 13 cols
##   -> target variable  : 506 values
##   -> predict function : yhat.randomForest will be used ( default )
##   -> predicted values : No value for predict function target column. ( default )
##   -> model_info       : package randomForest , ver. 4.7.1.2 , task regression ( default
##   -> predicted values : numerical, min = 6.762044 , mean = 22.55544 , max = 49.12006
##   -> residual function : difference between y and yhat ( default )
##   -> residuals        : numerical, min = -6.271153 , mean = -0.02263104 , max = 8.66107
##   A new explainer has been created!
```

```
# Calculate the importance of each variable
variable_importance <- variable_importance(explainer)
variable_importance
```

```
##      variable mean_dropout_loss      label
## 1  _full_model_      1.405939 randomForest
## 2      medv          1.405939 randomForest
## 3      zn            1.449780 randomForest
## 4      chas          1.477178 randomForest
## 5      rad           1.553602 randomForest
## 6      tax           1.861586 randomForest
## 7      age           1.889014 randomForest
## 8      indus         2.002525 randomForest
## 9      ptratio       2.335230 randomForest
## 10     crim         2.458435 randomForest
## 11     dis          2.563601 randomForest
```

```
## 12          nox          2.591746 randomForest
## 13           rm          5.553737 randomForest
## 14        lstat          6.446654 randomForest
## 15   _baseline_         12.565596 randomForest
```

Here, `mean_dropout_loss` is the average change in the model's performance across all permutations. The larger the value, the more important the variable is in predicting the outcome.

6.1.2 Decision tree structure

Another method for interpreting the results of a model is to examine the decision tree structure of the model. Of course, decision trees are far more accessible than other ML methods. As we have seen, there are several packages in R that can be used to visualise the decision tree structure of a model.

In a random forest setting, the decision tree structure of the model can be used to determine the importance of each variable. The importance of each variable is determined by the number of times the variable is used in the decision tree structure.

Example

Here we will use the `rpart` package to visualise the decision tree structure of a random forest model.

```
library(randomForest)

# Fit the random forest model
rf_model_cars <- randomForest(mpg ~ .,
                              data = mtcars)

# Extract the decision tree structure of the model
tree <- getTree(rf_model_cars, k = 1, labelVar = TRUE)

# Examine the decision tree structure
tree
```

##	left daughter	right daughter	split var	split point	status	prediction
## 1	2	3	cyl	5.00	-3	21.69063
## 2	4	5	carb	1.50	-3	27.15333
## 3	6	7	qsec	16.96	-3	16.87059
## 4	0	0	<NA>	0.00	-1	30.82000
## 5	8	9	wt	2.46	-3	25.32000
## 6	10	11	disp	339.00	-3	14.84286
## 7	12	13	carb	1.50	-3	18.29000
## 8	14	15	disp	107.70	-3	27.46667
## 9	0	0	<NA>	0.00	-1	22.10000
## 10	0	0	<NA>	0.00	-1	15.25000
## 11	0	0	<NA>	0.00	-1	14.30000
## 12	0	0	<NA>	0.00	-1	19.75000
## 13	16	17	disp	352.00	-3	17.31667
## 14	0	0	<NA>	0.00	-1	30.40000

```
## 15      0      0      <NA>      0.00      -1      26.00000
## 16      0      0      <NA>      0.00      -1      16.37500
## 17      0      0      <NA>      0.00      -1      19.20000
```

A function called `importance` within the `randomForest` package can be used to identify “important” variables. The measure of importance is the total decrease in node impurities from splitting on the variable, averaged over all trees. For classification, the node impurity is measured by the Gini index. For regression, it is measured by the residual sum of squares.

```
# Extract the importance of each variable
importance <- importance(rf_model_cars)
importance
```

```
##      IncNodePurity
## cyl      186.23363
## disp     254.52584
## hp       196.60489
## drat      55.19186
## wt       248.18756
## qsec      27.49346
## vs        24.11108
## am        16.54080
## gear      18.08646
## carb      33.35527
```

Looking at this measure alone, `cyl`, `disp`, `hp` and `wt` stand out as the most important as they are allowing the tree algorithm to split into relatively purer branches.

For a random forest model, we can also count how many times variables are used in the decision tree structure.

```
# Count how many times variables are used in the
# decision tree structure
variable.names(mtcars)[-1]
```

```
## [1] "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear" "carb"
varUsed(rf_model_cars)
```

```
## [1] 304 856 812 508 811 541 138 109 162 315
```

This might suggest that `disp` is the most important variable in the model, but it could be that `am` is the most important but its influence is captured early in the decision tree structure.

Instead, we might count the number of times each variable is used for the first split in a decision tree within the random forest:

```
# Count how many times each variable is used for the
# first split in a decision tree
first.split <- sapply(1:rf_model_cars$ntree, function(i) {
  rfTree <- getTree(rf_model_cars, k = i, labelVar = TRUE)[1,3]
  variable.names(mtcars)[-1][rfTree]
})
```

```
table(first.split)
```

```
## first.split
##   am  cyl disp drat   hp qsec   vs   wt
##    4   81  137   68   82   45   17   66
```

Now, as the first division tends to be the most important in terms of predicting the outcome, we can see that `disp` and `cyl` seem to be the most important variables in the model.

6.2 Main effect visualisations

6.2.1 Main effects

A main effect is the effect of a variable on the outcome, while holding all other variables constant. To help us to understand the utility of these let's consider a situation where we know the true model generating the data.

Example

Here, we have one outcome variable and two explanatory variables. The true model is:

$$Y = X_1 + X_2^2.$$

We will have strongly correlated explanatory variables:

$$\begin{aligned} X_1 &\sim \text{Uni}(0, 1), \\ X_2 &= X_1 + \epsilon, \quad \epsilon \sim N(0, 0.02). \end{aligned}$$

Here's some data for us to work with:

It is trivial to derive the main effect of the model for fixed values of X_1 :

$$\begin{aligned} E[Y|X_1 = x_1] &= x_1 + E[X_2^2|X_1 = x_1] \\ &= x_1 + E[(x_1 + \epsilon)^2] \\ &= x_1 + x_1^2 + E[\epsilon^2]. \end{aligned}$$

Similarly for fixed values of X_2 :

$$\begin{aligned} E[Y|X_2 = x_2] &= E[X_1|X_2 = x_2] + x_2^2 \\ &= E[x_2 - \epsilon] + x_2^2 \\ &= x_2 + x_2^2. \end{aligned}$$

You will notice in these derivations that we have had to account for the joint distribution of the variables.

Figures 6.3 and 6.4 show the main effects of X_1 and X_2 on Y respectively (plotted against all the observations).

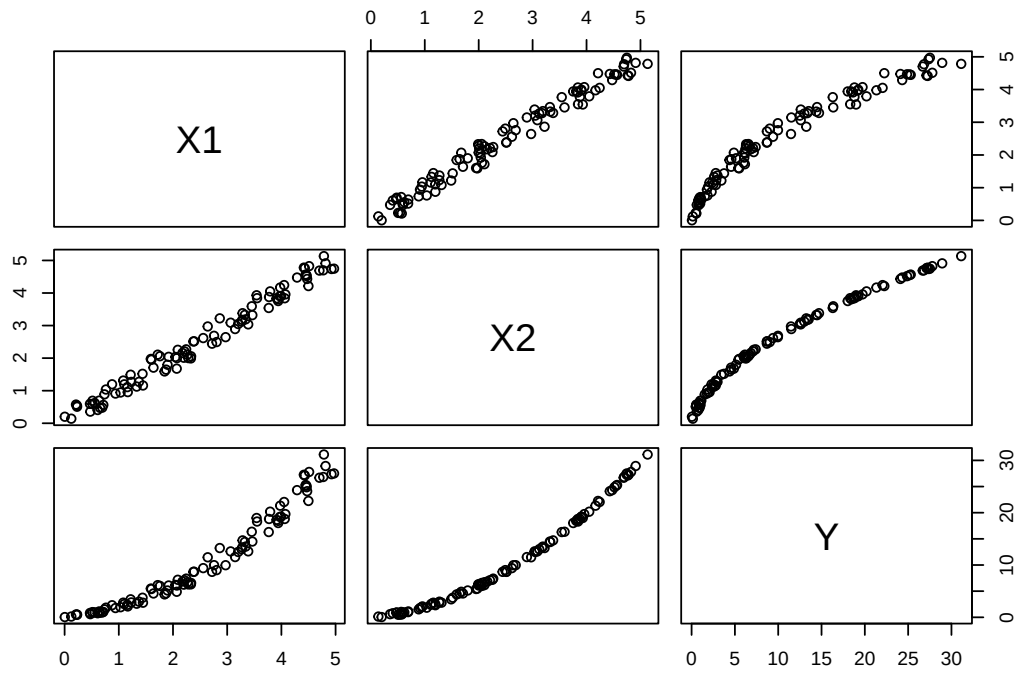
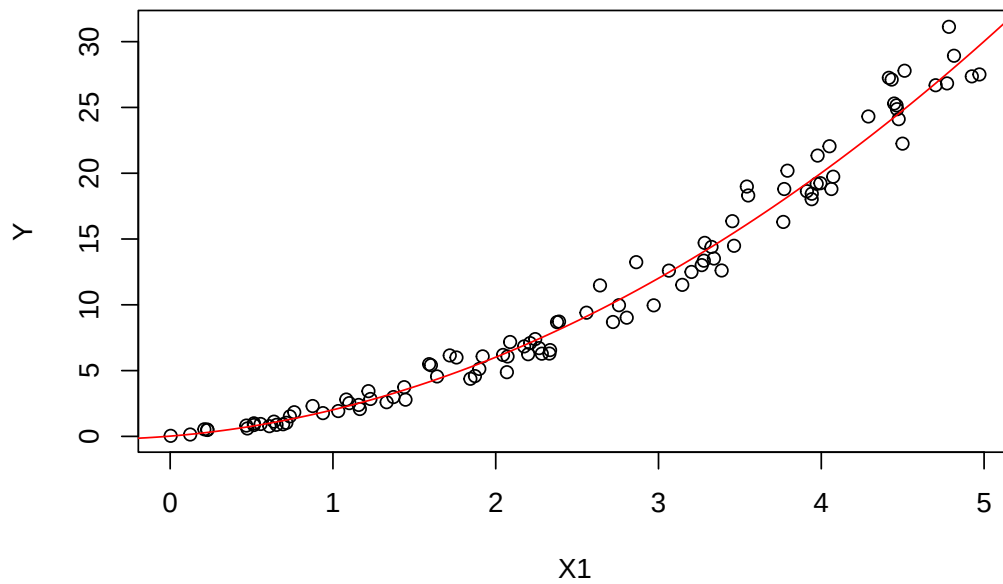


Figure 6.2: The relationship between the variables in the data.

Figure 6.3: The main effect of X_1 on Y .

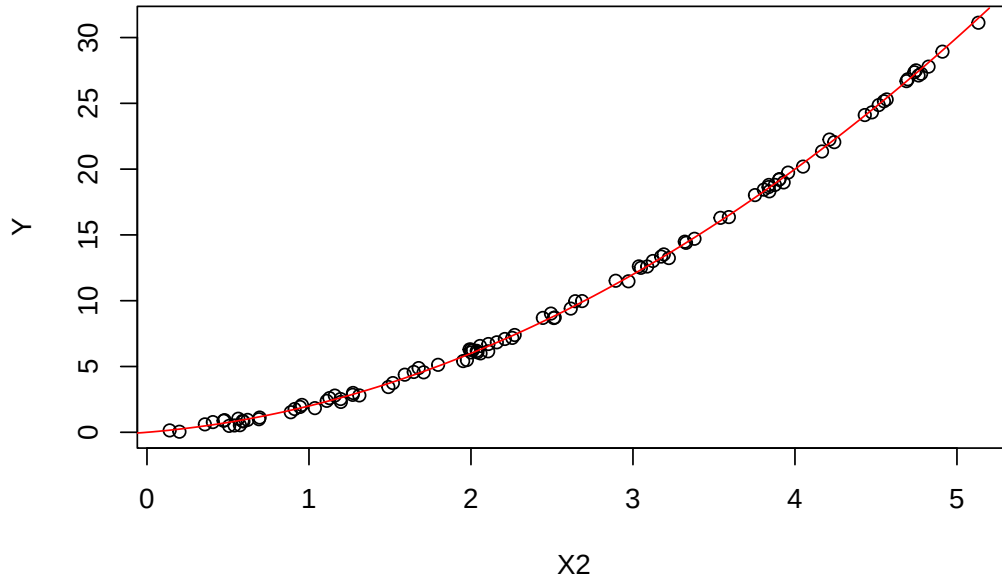


Figure 6.4: The main effect of X2 on Y.

6.2.2 Partial dependence plots

One method for interpreting the results of a model is to use partial dependence plots. A partial dependence plot shows the relationship between a variable and the outcome, while holding all other variables constant. This allows us to see the effect of a variable on the outcome, while controlling for the effects of other variables.

More formally, consider a prediction model $\hat{f}(\mathbf{x})$ where $\mathbf{x} = (x_1, x_2, \dots, x_p)$ is a vector of p covariates. Let $S \subseteq \{1, 2, \dots, p\}$ be a subset of covariate indices that we are interested in, and let $C = \{1, 2, \dots, p\} \setminus S$ be the complement set. We can partition the covariate vector as $\mathbf{x} = (\mathbf{x}_S, \mathbf{x}_C)$, where $\mathbf{x}_S$ contains the covariates of interest and $\mathbf{x}_C$ contains the remaining covariates.

The **partial dependence function** is defined as the expected value of the model predictions over the marginal distribution of $\mathbf{x}_C$:

$$\hat{f}_S(\mathbf{x}_S) = E_{\mathbf{x}_C} [\hat{f}(\mathbf{x}_S, \mathbf{x}_C)] = \int \hat{f}(\mathbf{x}_S, \mathbf{x}_C) p(\mathbf{x}_C) d\mathbf{x}_C,$$

where $p(\mathbf{x}_C)$ is the marginal distribution of $\mathbf{x}_C$.

In practice, the expectation is approximated using a Monte Carlo estimate over the training data. Given a training dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$, the partial dependence function is estimated as:

$$\hat{f}_S(\mathbf{x}_S) \approx \frac{1}{n} \sum_{i=1}^n \hat{f}(\mathbf{x}_S, \mathbf{x}_C^{(i)}).$$

For the common case where S contains a single covariate x_j , the partial dependence function simplifies to:

$$\hat{f}_j(x_j) = \frac{1}{n} \sum_{i=1}^n \hat{f}(x_1^{(i)}, \dots, x_{j-1}^{(i)}, x_j, x_{j+1}^{(i)}, \dots, x_p^{(i)}).$$

This formula shows that we evaluate the model at a fixed value of x_j while averaging over all observed values of the other covariates. It is important to note that partial dependence plots assume that

the covariate of interest is independent of the other covariates. When covariates are correlated, the partial dependence function may include predictions at covariate combinations that are unlikely or impossible in the real data, which can lead to misleading interpretations.

Example

Here, we will use the `iml` package to create a partial dependence plot for the random forest model for the Boston data.

```
library(iml)

# Create an explainer object
explainer <- Predictor$new(rf_model,
                           data = Boston)

# Create a partial dependence plot for the variable `lstat`
lstat_pd <- FeatureEffect$new(explainer,
                              feature = "lstat",
                              method = 'pdp')

# Plot the partial dependence plot
plot(lstat_pd)
```

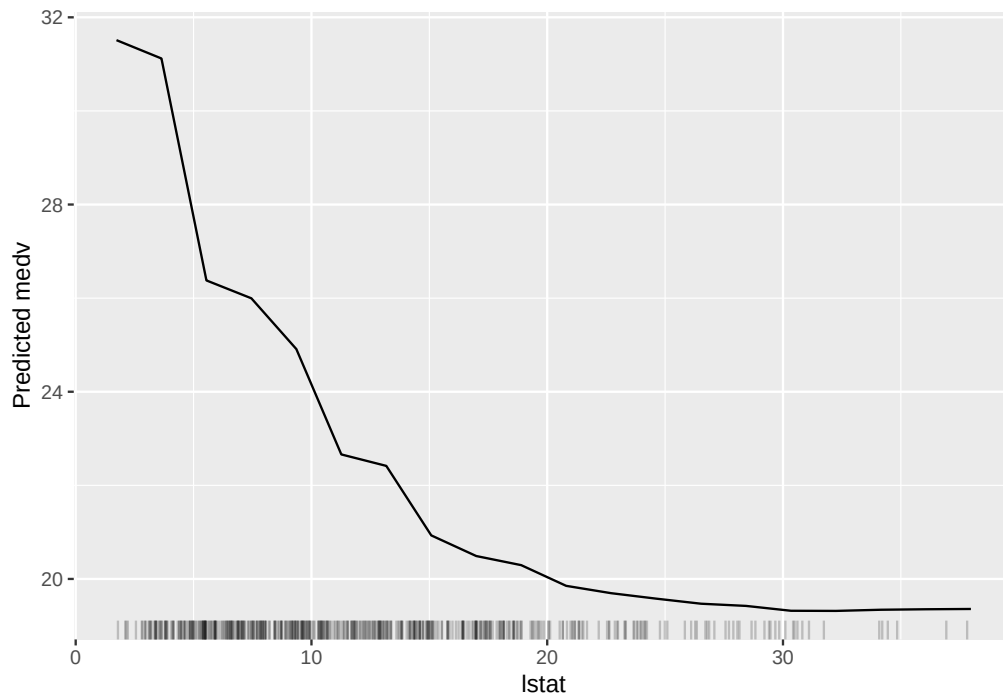


Figure 6.5: The partial dependence plot for the variable ‘lstat’.

In Figure 6.5, we can see a clear negative relationship between `lstat` and `medv`. As `lstat` increases, `medv` decreases. This is consistent with our understanding of the data because `lstat` is the percentage of lower status of the population, which is likely to be negatively correlated with the median value of owner-occupied homes.

6.2.3 Accumulated local effects

Another method for interpreting the results of a model is to use accumulated local effects (ALE). ALE plots address a key limitation of partial dependence plots: when covariates are correlated, PDPs can extrapolate to unrealistic regions of the covariate space. ALE plots avoid this by considering only local changes in predictions within intervals where data actually exist.

Mathematically, the ALE function for a single covariate x_S is defined as:

$$\text{ALE}_S(x_S) = \int_{z_{0,S}}^{x_S} \mathbb{E}_{\mathbf{x}_C | x_S=z} \left[\frac{\partial \hat{f}(\mathbf{x})}{\partial x_S} \right] dz - c,$$

where $z_{0,S}$ is a reference point (typically the minimum value of x_S), $\mathbf{x}_C$ denotes the covariates other than x_S , and c is a centering constant chosen so that the ALE function averages to zero over the distribution of x_S .

In practice, we approximate this integral using a discrete grid. Let $\{z_{0,S}, z_{1,S}, \dots, z_{K,S}\}$ be a partition of the range of x_S into K intervals. For each interval $[z_{k-1,S}, z_{k,S}]$, let $n_S(k)$ denote the number of training observations with $x_S \in [z_{k-1,S}, z_{k,S}]$. The ALE function is then estimated as:

$$\widehat{\text{ALE}}_S(x_S) = \sum_{k=1}^{k_S(x_S)} \frac{1}{n_S(k)} \sum_{i: x_S^{(i)} \in [z_{k-1,S}, z_{k,S}]} \left[\hat{f}(z_{k,S}, \mathbf{x}_C^{(i)}) - \hat{f}(z_{k-1,S}, \mathbf{x}_C^{(i)}) \right] - c,$$

where $k_S(x_S)$ is the index of the interval containing x_S , and c is chosen to ensure that the sample mean of $\widehat{\text{ALE}}_S$ is zero.

The key insight is that, for each observation in an interval, we compute the difference in predictions when x_S is moved from the lower to the upper boundary of the interval, while keeping all other covariates at their observed values. This local effect is then averaged over all observations in the interval and accumulated across intervals.

The algorithm can be summarised as follows:

1. Partitioning the variable's range:
 - Divide the range of the variable into a reasonable number of intervals;
 - The number of intervals can influence the smoothness of the ALE plot;
 - For each interval, calculate the average value of the variable within that interval.
2. Calculating Local Effects:
 - For each interval, calculate the difference in the average model prediction between:
 - The original data within that interval;
 - The data where the variable values within that interval are shifted by a small amount (e.g., half the interval width);
 - This difference represents the local effect of the variable within that interval.
3. Accumulating Local Effects:
 - Start with the lowest interval and calculate the local effect for that interval;
 - For each subsequent interval, add the local effect of that interval to the accumulated effect from the previous intervals;
 - This gives you the cumulative effect of the variable on the model's predictions up to that point.

The ALE approach has several advantages over PDPs:

- **Unbiased estimation:** ALE plots are unbiased even when covariates are correlated, because they only use observed covariate combinations.
- **Computational efficiency:** ALE plots require fewer model evaluations than PDPs.
- **Interpretation:** The ALE value at a point x_S represents the effect of x_S on the prediction relative to the average effect.

Example

Here, we will again use the `iml` package to create an ALE plot.

```
lstat_pd <- FeatureEffect$new(explainer,  
                             feature = "lstat",  
                             method = 'ale')  
  
# Plot the partial dependence plot  
plot(lstat_pd)
```

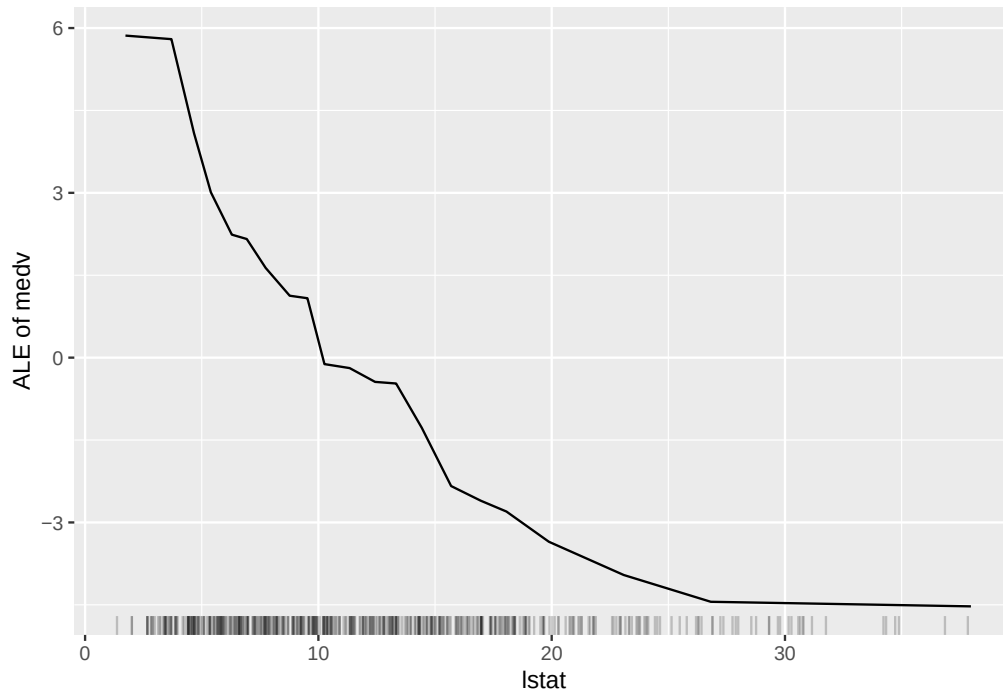


Figure 6.6: The ALE plot for the variable ‘lstat’.

Figure 6.6 is very similar to Figure 6.5. This is perhaps because there is not a great amount of interaction between `lstat` and the other variables in the model. However, the ALE plot is more interpretable than the partial dependence plot because it shows the difference in predictions when `lstat` is changed, rather than the average prediction itself.

Chapter 7

Unsupervised learning

Unsupervised learning is a type of machine learning that deals with finding hidden patterns or intrinsic structures in data without explicit supervision. Unlike supervised learning, where the algorithm is trained on labelled data, unsupervised learning algorithms work on unlabelled data with an aim to discover the underlying patterns or relationships within the data.

7.1 Traditional statistical approaches

7.1.1 Clustering

Clustering is a common unsupervised learning technique that groups similar data points together based on some similarity measure. The goal of clustering is to partition the data into groups such that data points within the same group are more similar to each other than to those in other groups. Clustering is useful for exploratory data analysis, pattern recognition, and anomaly detection.

7.1.1.1 Hierarchical clustering

Hierarchical clustering is a method of cluster analysis that builds a hierarchy of clusters. It starts with each data point as a separate cluster and then merges the closest clusters iteratively until all data points belong to a single cluster. There are two main types of hierarchical clustering: agglomerative and divisive.

In agglomerative hierarchical clustering, each data point starts as a single cluster, and at each step, the two closest clusters are merged until all data points belong to a single cluster. The distance between clusters can be measured using different methods, such as Euclidean distance, Manhattan distance, or correlation distance.

In divisive hierarchical clustering, all data points start in a single cluster, and at each step, the cluster with the largest dissimilarity is split into two clusters. This process continues until each data point is in its own cluster.

In R, hierarchical clustering is easily implemented using the `hclust()` function from the base `stats` package. The `hclust()` function takes a distance matrix as input, which can be computed using the `dist()` function. Agglomerative hierarchical clustering is the default method in R.

```
# Load the iris dataset
data(iris)

# Perform hierarchical clustering
hc <- hclust(dist(iris[, -5]))

# Plot the dendrogram
plot(hc)
```

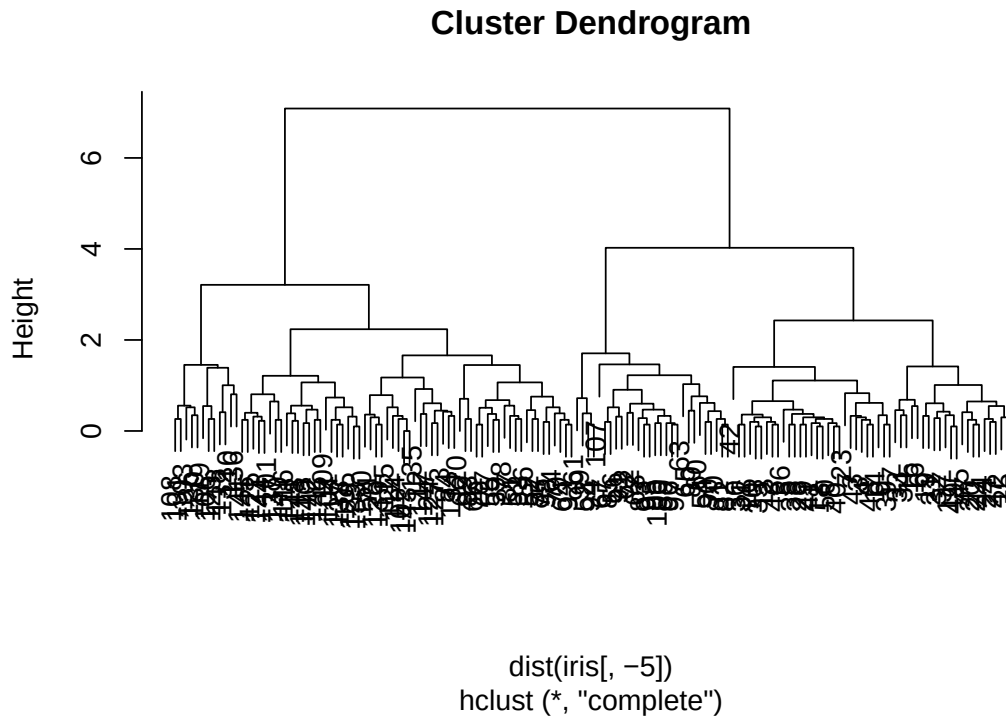


Figure 7.1: Dendrogram of the iris dataset using hierarchical clustering.

The algorithm produces a dendrogram, which is a tree-like diagram that shows the hierarchical relationship between clusters. The height of the dendrogram represents the dissimilarity between clusters. The `cutree()` function can be used to cut the dendrogram at a specific height to obtain the cluster assignments.

```
# Cut the dendrogram to obtain cluster assignments
clusters <- cutree(hc, k = 3)

# Display the dendrogram with cluster assignments
plot(hc)
rect.hclust(hc, k = 3, border = 2:4)
```

We may ask how well these “natural” clusters accord with the species of iris in the dataset. We can use the `table()` function to compare the cluster assignments with the actual species.

```
# Compare cluster assignments with species
table(iris$Species, clusters)
```

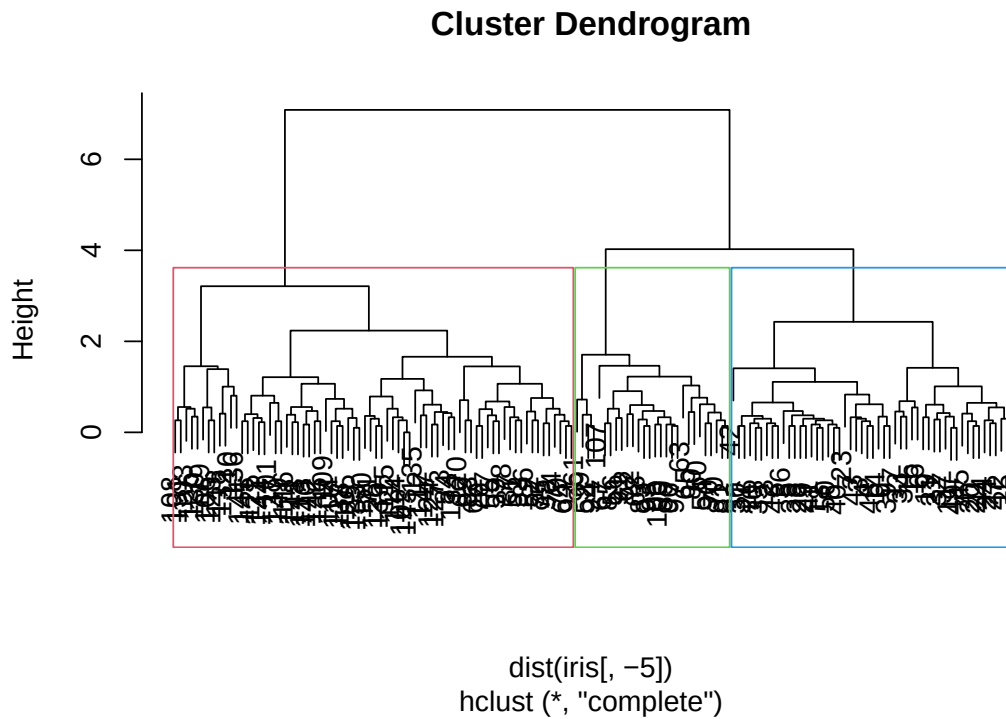


Figure 7.2: Dendrogram of the iris dataset with cluster assignments.

```
##           clusters
##           1  2  3
##  setosa    50  0  0
##  versicolor 0 23 27
##  virginica  0 49  1
```

7.1.1.2 *k*-means clustering

k-means is a popular clustering algorithm that partitions data into k clusters based on the similarity of data points. The algorithm works as follows:

1. Randomly initialise k cluster centroids.
2. Assign each data point to the nearest cluster centroid.
3. Update the cluster centroids by computing the mean of all data points assigned to each cluster.
4. Repeat steps 2 and 3 until the cluster centroids no longer change.

Elaborating on step 3, the new cluster centroid is calculated as the mean of all data points assigned to that cluster. This is why *k*-means is also known as the “centroid-based” clustering algorithm. The algorithm is sensitive to the initialisation of cluster centroids, and the choice of k is crucial for the quality of the clustering. Cluster membership is closely related to the tiles formed by a Voronoi tessellation with tiles defined by the cluster centroids.

Like we saw earlier with linear discriminant analysis, it can be useful to partition the total sum of squares into within-cluster and between-cluster sums of squares. The within-cluster sum of squares measures the compactness of the clusters, while the between-cluster sum of squares measures the

separation between clusters. You may think of the goal of k -means clustering to be to minimise the within-cluster sum of squares.

If we have k clusters, n data points overall and n_j data points for the j th cluster, the total sum of squares (SST) can be partitioned as follows:

$$\text{SST} = \sum_{i=1}^n \|\mathbf{x}_i - \bar{\mathbf{x}}\|_2^2$$

where $\bar{\mathbf{x}}$ is the mean of all data points. The within-cluster sum of squares ($\text{SS}_{\text{within}}$) is given by:

$$\text{SS}_{\text{within}} = \sum_{j=1}^k \sum_{i=1}^{n_j} \|\mathbf{x}_{ij} - \bar{\mathbf{x}}_j\|_2^2$$

where $\bar{\mathbf{x}}_j$ is the mean of the j th cluster. The between-cluster sum of squares ($\text{SS}_{\text{between}}$) is then given by:

$$\text{SS}_{\text{between}} = \sum_{j=1}^k n_j \|\bar{\mathbf{x}}_j - \bar{\mathbf{x}}\|_2^2$$

and we have that $\text{SST} = \text{SS}_{\text{within}} + \text{SS}_{\text{between}}$.

In R, k -means clustering is easily implemented using the `kmeans()` function from the base `stats` package.

```
# Load the iris dataset
data(iris)

# Perform k-means clustering
kmeans_result <- kmeans(iris[, -5], centers = 3)
```

The `kmeans()` function returns a list with the cluster assignments, cluster centroids, and the within-cluster sum of squares. It is also useful to visualise the results of clustering using a scatter plot.

```
# Scatter plot of the first two variables coloured by cluster
plot(iris[, 1:2], col = kmeans_result$cluster)
points(kmeans_result$centers[, 1:2], col = 1:3, pch = 8, cex = 2)
```

Again, we can check to see how well these align with the true species.

```
# Compare cluster assignments with species
table(iris$Species, kmeans_result$cluster)
```

```
##
##           1  2  3
##  setosa    0 50  0
##  versicolor 2  0 48
##  virginica 36  0 14
```

The question still remains of how to choose the number of clusters. The “elbow method” is a common technique for determining the optimal number of clusters in k -means clustering. The idea is to plot the between-cluster sum of squares against the number of clusters and look for an “elbow” point where the rate of decrease slows down. This point is considered the optimal number of clusters.

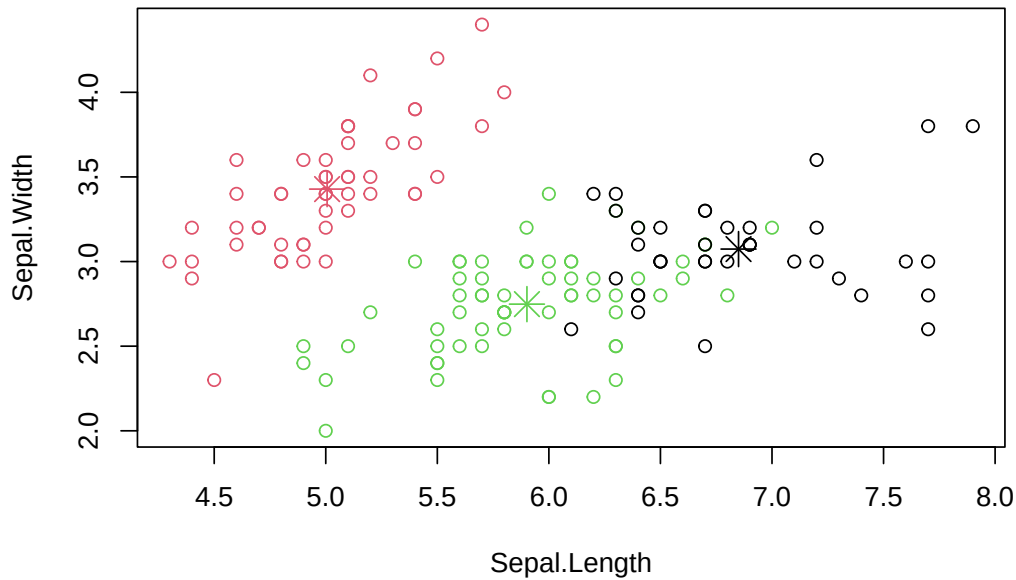


Figure 7.3: Scatter plot of the iris dataset with cluster assignments.

For the iris dataset, the elbow point could be interpreted as being at $k = 2$, which corresponds to the two clusters that can be easily identified in the pairwise scatter plots. However, it is easy to see that $k = 4$ is another valid choice as this is at the start of the final plateau of within-cluster sum of squares. Unfortunately, and like so much in statistics, the choice of k is subjective and depends on the context of the data and the goals of the analysis.

```
# Elbow plot
wss <- numeric(10)
for (i in 1:10) {
  kmeans_result <- kmeans(iris[, -5], centers = i)
  wss[i] <- sum(kmeans_result$withinss)
}
plot(1:10, wss, type = "b",
     xlab = "Number of clusters",
     ylab = "Within sum of squares")
```

7.1.2 Dimension reduction

Dimension reduction techniques are used to reduce the number of variables in a dataset while preserving the most important information. This is useful for visualisation, data compression, and speeding up machine learning algorithms. Two popular techniques are Principal Component Analysis (PCA) and Multidimensional Scaling.

As ever, we have been imprecise here by saying “most important information”. It is a subjective decision based on the context of the data and the goals of the analysis. It might be important to maximise the diversity of the data, or to maximise the variance, or to minimise the correlation between variables, or to focus on specific features that are most relevant to the downstream task. The choice depends on whether the goal is data compression, visualisation, or improving the performance of a specific machine learning algorithm.

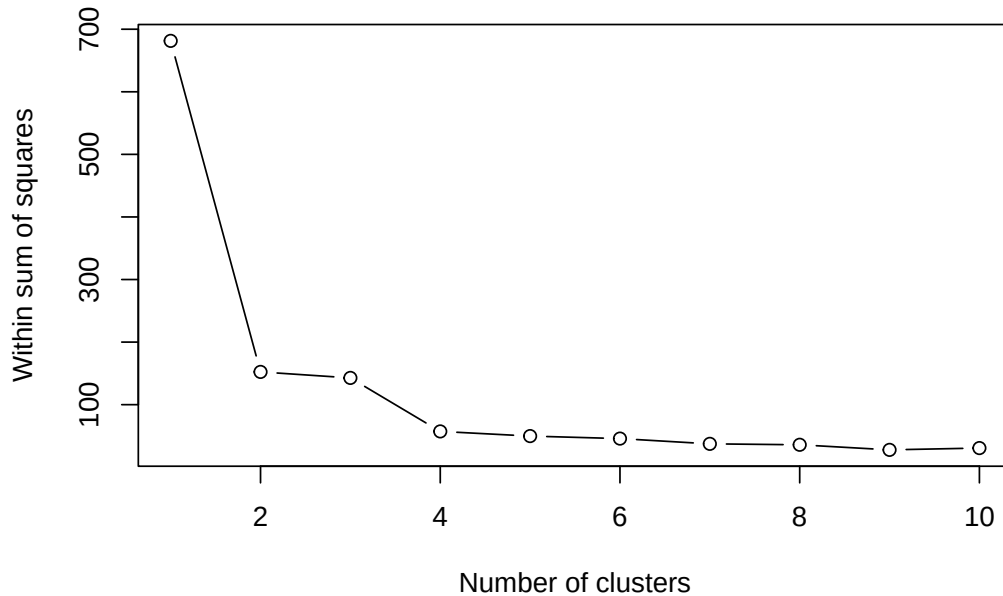


Figure 7.4: Elbow plot for the iris dataset.

(#fig:unsup-chunk-7, 7.kmean.elbow)

7.1.2.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a popular (and old) technique for dimensionality reduction. It transforms high-dimensional data into a lower-dimensional space whilst preserving the variance in the data. It is desirable to maintain the variance as it represents the information content of the data. PCA works by finding the directions of maximum variance in the data and projecting the data onto these directions.

The algorithm is as follows:

1. Standardise the data by subtracting the mean and dividing by the standard deviation.
2. Compute the covariance matrix of the data.
3. Compute the eigenvectors and eigenvalues of the covariance matrix.
4. Sort the eigenvectors by decreasing eigenvalues and choose the top k eigenvectors.
5. Project the data onto the top k eigenvectors to obtain the lower-dimensional representation.

PCA is a linear technique, which means it can only capture linear relationships in the data. For non-linear relationships, other techniques or non-linear transformations are more appropriate.

Mathematically, we can represent the transformation as follows:

$$Y = XW$$

where X is the original data matrix, Y is the transformed data matrix, and W is the matrix of eigenvectors. The columns of W are the principal components, and the rows of Y are the data points in the lower-dimensional space.

In R, PCA is easily implemented using the `prcomp()` function from the base `stats` package.

```
# Load the iris dataset
data(iris)

# Perform PCA
pca <- prcomp(iris[, -5], scale = TRUE)

# Print the summary
summary(pca)
```

```
## Importance of components:
##              PC1      PC2      PC3      PC4
## Standard deviation  1.7084 0.9560 0.38309 0.14393
## Proportion of Variance 0.7296 0.2285 0.03669 0.00518
## Cumulative Proportion 0.7296 0.9581 0.99482 1.00000
```

The `summary()` function provides the proportion of variance explained by each principal component, the cumulative proportion of variance explained, and the rotation matrix (eigenvectors). It is also useful to visualise the results of PCA using a scree plot or a biplot.

```
# Scree plot
plot(pca)
```

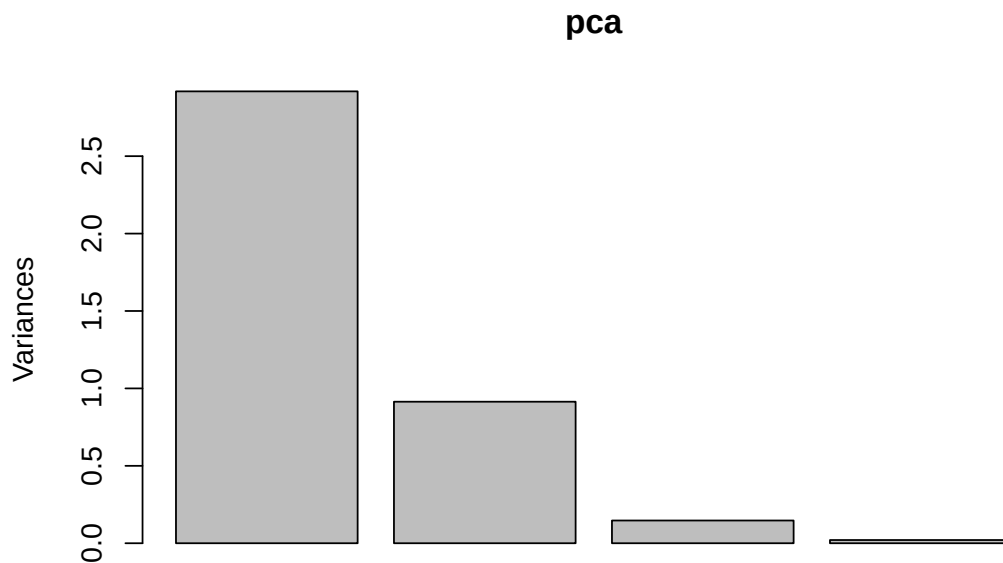


Figure 7.5: Scree plot of the iris dataset using PCA.

```
# Biplot
biplot(pca)
```

We should note that PCA is affected by the scale of the data, so it is important to standardise the data before applying PCA. The `scale = TRUE` argument in the `prcomp()` function standardises the data by subtracting the mean and dividing by the standard deviation. This leads to manipulations of the correlation matrix rather than the covariance matrix.

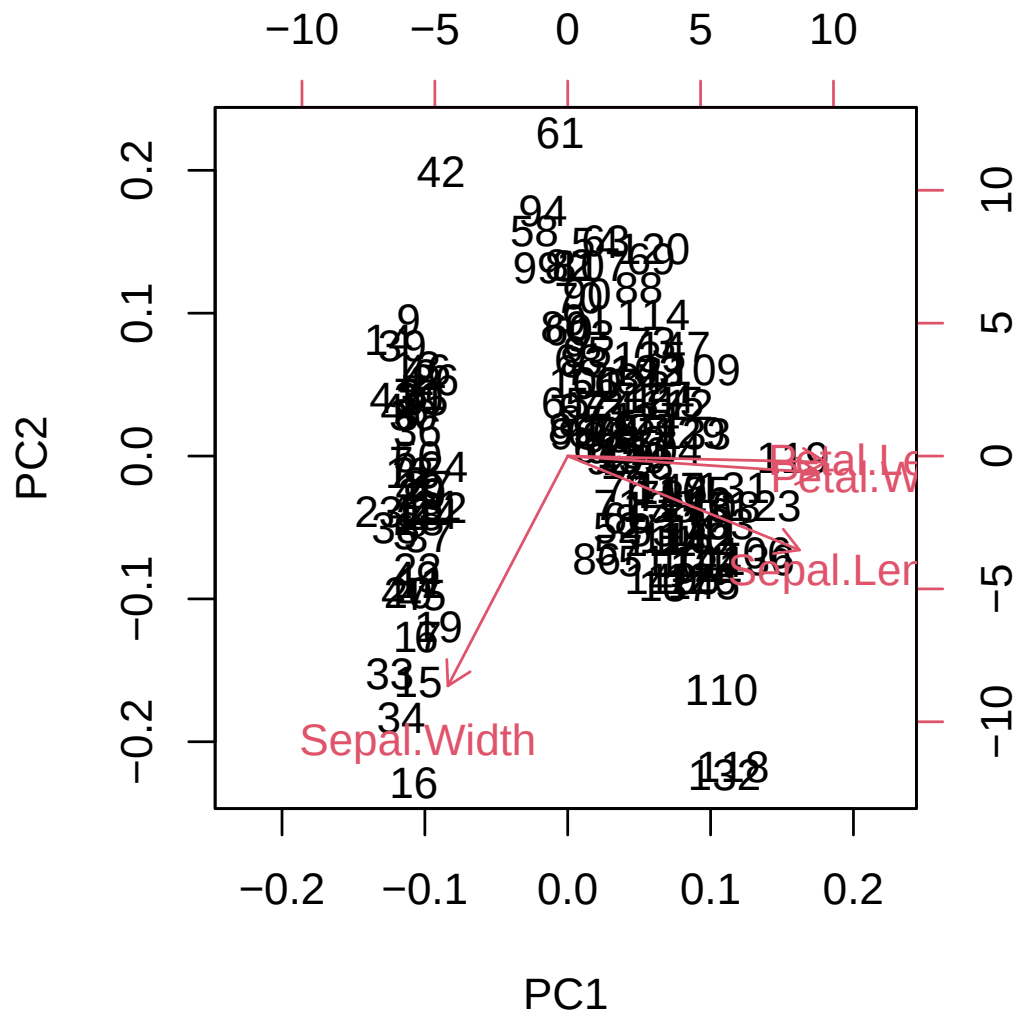


Figure 7.6: Biplot of the iris dataset using PCA.

```
# Perform PCA with standardised data
pca <- prcomp(iris[, -5], scale = TRUE)

# Print the summary
summary(pca)

## Importance of components:
##              PC1      PC2      PC3      PC4
## Standard deviation  1.7084 0.9560 0.38309 0.14393
## Proportion of Variance 0.7296 0.2285 0.03669 0.00518
## Cumulative Proportion 0.7296 0.9581 0.99482 1.00000
```

For those who are interested, you may also want to investigate the other traditional statistical techniques of Multidimensional Scaling (MDS) and Factor Analysis. These are also used for dimensionality reduction and have been used extensively across the sciences.

7.2 Modern unsupervised learning techniques

7.2.1 Clustering with Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

This is a powerful algorithm for clustering data that does not require pre-defining the number of clusters (unlike k -means). It can handle clusters of arbitrary shapes and identify outliers, making it a versatile tool for real-world datasets. “Density-based” means that the algorithm groups together points that are closely packed rather than probability density. The “applications with noise” part of the name refers to the ability of the algorithm to identify outliers as points that do not belong to any cluster. Again, there is a subjective decision to be made about what constitutes a cluster and what constitutes noise. Specifically, we need to think about what distance metric to use and what the minimum number of points in a cluster should be.

The algorithm works by defining two parameters: ϵ , which specifies the maximum distance between two points for them to be considered in the same neighbourhood, and m , which specifies the minimum number of points required to form a dense region. Points that do not meet these two criteria are considered outliers.

The algorithm works as follows:

1. For each data point, find all points within a distance of ϵ .
2. If a point has at least m points within ϵ , it is considered a core point.
3. Expand the cluster by adding all reachable points from the core point.
4. Repeat the process for all unvisited points until all points are assigned to a cluster or marked as outliers.

In R, DBSCAN is implemented using the `dbscan` package.

```
library(dbscan)

# Load the iris dataset
data(iris)
```

```
# Perform DBSCAN clustering
dbscan_result <- dbscan(iris[, -5],
                        eps = 0.5,
                        minPts = 5)
```

The `dbscan()` function returns a list with the cluster assignments and the number of clusters found. It is also useful to visualise the results of clustering using a scatter plot.

```
# Scatter plot of the first two variables coloured
# by cluster
plot(iris[, 1:2], col = dbscan_result$cluster + 1)
```

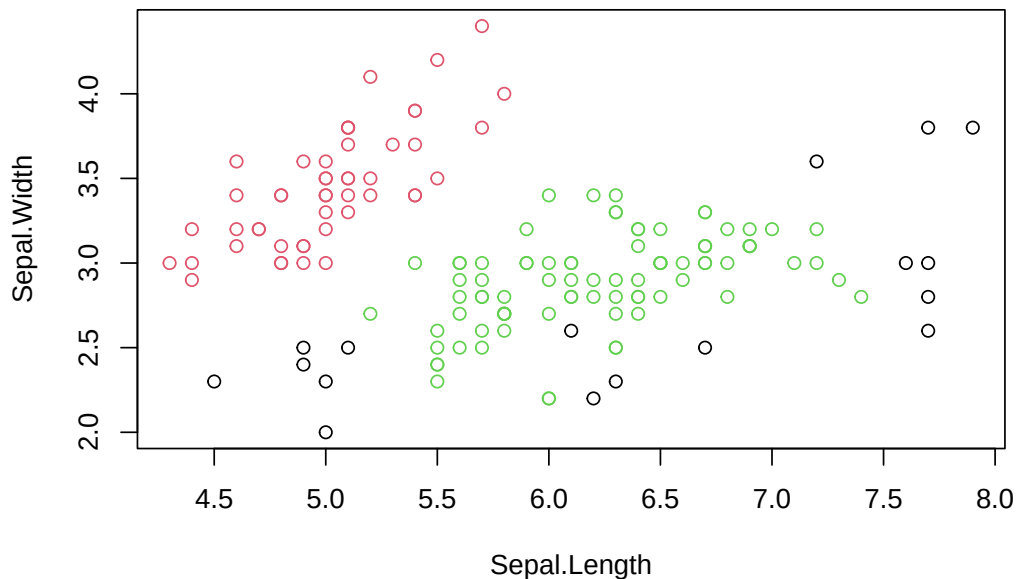


Figure 7.7: Scatter plot of the iris dataset with cluster assignments using DBSCAN.

We can look at a contingency table to evaluate whether the clusters align with the true species.

```
# Compare cluster assignments with species
table(iris$Species, dbscan_result$cluster)
```

```
##
##           0  1  2
##  setosa    1 49  0
##  versicolor 6  0 44
##  virginica 10  0 40
```

It is worth comparing the results of DBSCAN with *k*-means clustering to see how they differ in handling complex data structures. For the iris dataset, DBSCAN is likely to identify the three species of iris more accurately than *k*-means because of the non-linear relationships between the variables.

```
# Compare DBSCAN with k-means clustering
par(mfrow = c(1, 2))
plot(iris[, 1:2], col = dbscan_result$cluster + 1,
```

```

main = "DBSCAN")
plot(iris[, 1:2], col = kmeans_result$cluster,
     main = "k-means")

```

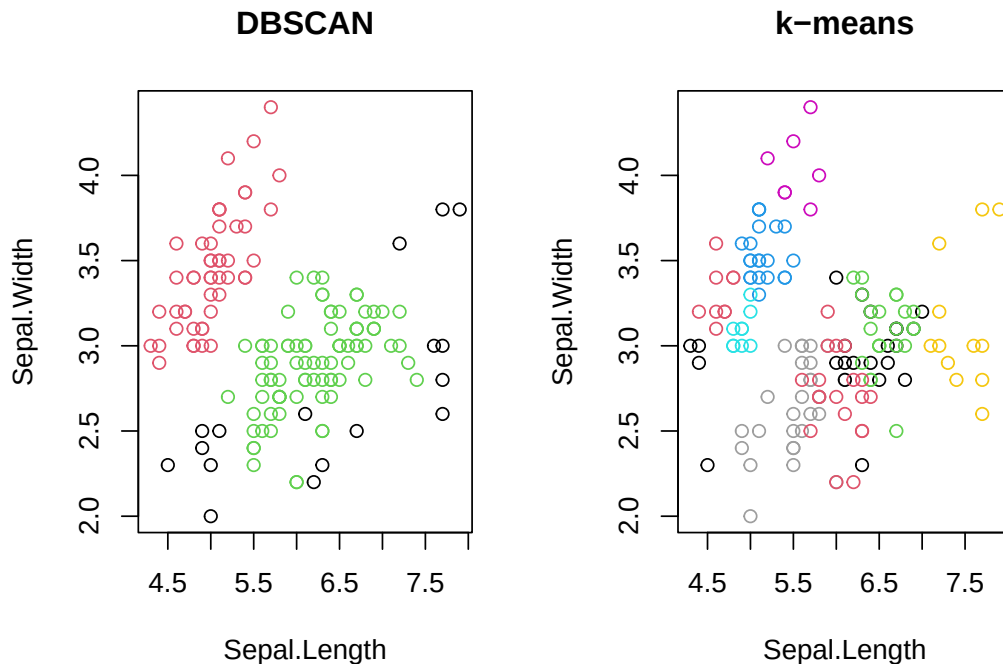


Figure 7.8: Comparison of DBSCAN and k-means clustering on the iris dataset.

The iris dataset is an archetypal dataset that is used alongside Euclidean distances. However, DBSCAN can be used with other distance metrics, such as the Manhattan distance, depending on the nature of the data.

```

# DBSCAN with Manhattan distance
dbscan_result_manhattan <- dbscan(dist(iris[, -5],
                                     method = "manhattan"),
                                  eps = 0.5, minPts = 5)

# Scatter plot of the first two variables
# coloured by cluster
plot(iris[, 1:2],
     col = dbscan_result_manhattan$cluster + 1)

```

Again, we are reminded that decisions will need to be made when telling a machine how to learn.

7.2.2 Dimensionality Reduction with t-SNE

While PCA is based on fundamentals from linear algebra and is easily implemented, t-SNE (t-distributed Stochastic Neighbour Embedding) is a more recent technique that does better at preserving the non-linear structure of high-dimensional data. t-SNE is a non-linear dimensionality reduction technique that is particularly useful for visualising high-dimensional data in two or three dimensions. It works by converting the high-dimensional data into a probability distribution and then

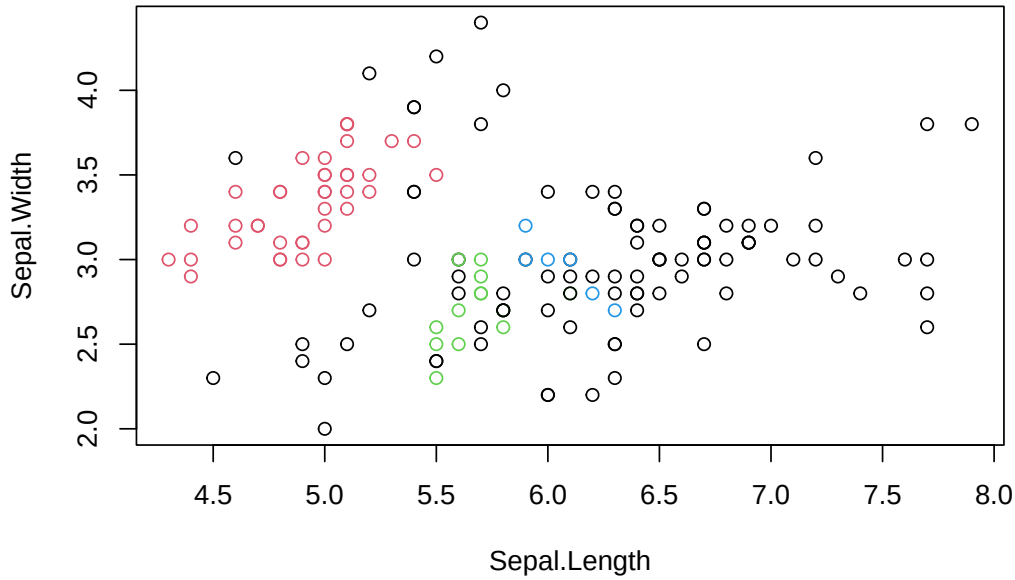


Figure 7.9: Scatter plot of the iris dataset with cluster assignments using DBSCAN with Manhattan distance.

minimising the Kullback-Leibler divergence between the joint probabilities in the high-dimensional space and the joint probabilities in the low-dimensional space.

Here are the steps of the algorithm:

1. Compute pairwise similarities between data points in high-dimensional space.
2. Convert the similarities to probabilities using a kernel.
3. Minimise the Kullback-Leibler divergence between the joint probabilities in high-dimensional space and the joint probabilities in the low-dimensional space.

Similarities are opposite to dissimilarities (and distances). It is always possible to convert from one to the other through a monotonic transformation:

$$\text{similarity} = \frac{1}{1 + \text{distance}}$$

Of course, this is just one way to do it. The choice of similarity measure is crucial for the success of t-SNE, and we have an infinite choice.

The t distribution appears in the name of the algorithm because the similarities are converted to probabilities using a t distribution with one degree of freedom. The t distribution has heavier tails than the Gaussian distribution, which helps to preserve the local structure of the data.

Kullback-Leibler divergence is a measure of how one probability distribution diverges from a second probability distribution. The general formula is

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

where P and Q are the two probability distributions. In the context of t-SNE, the goal is to minimise the Kullback-Leibler divergence between the joint probabilities in the high-dimensional space

(modelled using a mixture of normal distributions) and the joint probabilities in the low-dimensional space (modelled using a mixture of Cauchy distributions). The formula is as follows:

$$C = \sum_i KL(P_i || Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

The intuition is that having close probability distributions in the high-dimensional space should result in close probability distributions in the low-dimensional space. The Gaussian and Cauchy distributions are used to model the similarities in the high-dimensional and low-dimensional spaces, respectively. The Cauchy distribution is used because it has heavier tails than the Gaussian distribution, which gives higher probability to points that are far apart in the high-dimensional space. This helps to preserve the local structure of the data in the low-dimensional space. Specifically, we have

$$p_{j|i} = \frac{\exp(-||\mathbf{x}_i - \mathbf{x}_j||_2^2 / 2\sigma^2)}{\sum_{j' \neq i} \exp(-||\mathbf{x}_i - \mathbf{x}_{j'}||_2^2 / 2\sigma^2)}$$

and

$$q_{j|i} = \frac{(1 + ||\mathbf{y}_i - \mathbf{y}_j||_2^2)^{-1}}{\sum_{j' \neq i} (1 + ||\mathbf{y}_i - \mathbf{y}_{j'}||_2^2)^{-1}}$$

where σ is a hyperparameter that controls the variance of the Gaussian kernel in the high-dimensional space, and $\mathbf{x}_i$ and $\mathbf{y}_i$ are the high-dimensional and low-dimensional representations of the i th data point, respectively. There is no equivalent hyperparameter in the low-dimensional space as scaling is not required for visualisation purposes.

In practice, the minimisation is done using gradient descent (which will be discussed in the second half of the course). Before applying t-SNE, we must decide on the number of dimensions for the low-dimensional space (usually two or three for visualisation). The variances of the Gaussian kernel are also important hyperparameters that need to be tuned.

In R, t-SNE is implemented using the `Rtsne` package.

```
library(Rtsne)

# Load the iris dataset
data(iris)

# Remove duplicates
iris <- unique(iris)

# Perform t-SNE
tsne_result <- Rtsne(iris[, -5], dims = 2,
                     perplexity = 30)

# Plot the t-SNE results
plot(tsne_result$Y, col = iris$Species, pch = 19)
```

The `Rtsne` function has an argument `perplexity` that controls the number of nearest neighbours used in the Gaussian kernel. It is essentially a proxy for the variance in the Gaussian kernel with higher values of perplexity corresponding to larger variances.

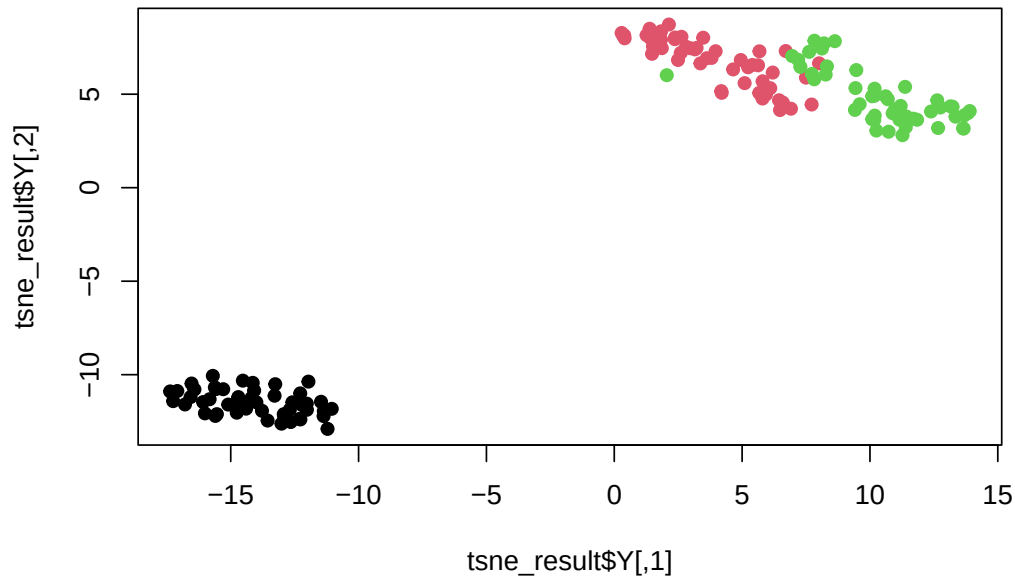


Figure 7.10: t-SNE plot of the iris dataset.

7.3 Other ideas in unsupervised learning

There are many more techniques in unsupervised learning that are beyond the scope of this course. Some of the popular ones include Generative Adversarial Networks (GANs), Autoencoders, and Self-Organising Maps (SOMs). These techniques are used for various tasks such as generating new data, learning representations, and visualising high-dimensional data.

1. **Generative Adversarial Networks (GANs):** GANs are a type of neural network architecture that consists of two networks: a generator and a discriminator. The generator generates new data samples, while the discriminator tries to distinguish between real and generated samples. The two networks are trained in an adversarial manner, where the generator tries to fool the discriminator, and the discriminator tries to distinguish between real and generated samples. GANs are used for tasks such as image generation, data augmentation, and style transfer.
2. **Autoencoders:** Autoencoders are neural networks that learn to encode data into a lower-dimensional representation and then decode it back to the original data. The goal of autoencoders is to learn a compressed representation of the data that captures the most important features. Autoencoders are used for tasks such as dimensionality reduction, denoising, and anomaly detection.
3. **Self-Organising Maps (SOMs):** SOMs are a type of neural network that learns to map high-dimensional data onto a low-dimensional grid. SOMs are used for tasks such as clustering, visualisation, and feature extraction. They are particularly useful for visualising high-dimensional data in two or three dimensions.