BI Notes

James Solomon-Rounce

2018-04-25

Contents

Preface

The following notes were taken by me for educational, non-commercial, purposes. If you find the information useful, buy the material/take the course.

Chapter 1

Querying Data with Transact-SQL

Notes taken during/inspired by the edX course 'Querying Data with Transact-SQL - Microsoft: DAT201x' by Graeme Malcolm and Geoff Allix.

Course Handouts

- Course Syllabus
- Getting Started Guide Including Install
- Adventure Works Entity Relationship Diagram
- Adeventure Works db install script
  NOTE: Remember to ensure read only access for everyone to the folder containing the .SQL and other files
- GitHib Repo for course including course materials, slids, labs etc
- A copy of the above materials should they be changed or removed

Other useful links * Transact-SQL Refrence

```
library(DBI)
```

```
## Loading required package: methods
```

```
# creates a connection to the SQL database
# note that "con" will be used later in each connection to the database
con <- DBI::dbConnect(odbc::odbc(),
                      Driver = "SQL Server",
                      Server = "localhost\\SQLEXPRESS",
                      Database = "AdventureWorksLT",
                      Trusted_Connection = "True")

# Sets knitr to use this connection as the default so we don't need to specify it for every chunk
knitr::opts_chunk$set(connection = "con")
```

1.1 Introduction to Transact-SQL

SQL or Structured Query Language was first developed in the 1970s by IBM as a way of interacting with databases. Other vendors have specfic versions of SQL for instance Oracle is PL/SQL, Microsoft's implentation is TSQL or Transact SQL. Both SQL Server (on prem) and Azure SQL Databases (cloud) use the same query language, however Azure is a subset of full TSQL since it some commands relate to local files and

data functions within .NET that relate only to SQL Server. However, as new features are added to Azure, some new commands are being added to Azure.

SQL is a declarative language - you express what it is that you want, the results - rather than specifying the steps taken to acheive that - it is not prodecural like other programming languages, it is set theory based. It is possible to write proceedural elements or steps within TSQL, however if this is ocurring a lot, it is perhaps better done in another language, which may also perform or run better.

In databases, we typically talk about entities - one type of thing - which is contained in each table.
* Entities are represented as relations (tables) * And entity attributes as domains (columns)

Most relationships are normalized, with relationships between primary and foreign keys. This helps to reduce duplication, however there are instances where de-normalised data is desired.

Schemas are namespaces for database objects - is shows a logical layout for all or part of a relational database, *"As part of a data dictionary, a database schema indicates how the entities that make up the database relate to one another, including tables, views, stored procedures, and more."*(see Lucid Chart on Database Schemas). The process of creating a database schema is called data modelling.

When referring to objects in a database, we could use a fully qualified name, such as:

- [server_name.][database_name.][schema_name.]object_name

This is only really relevant for SQL Server, since Azure will only work with one database at a time. Most of the time we typically just use

- schema_name.object_name

The schema name somwtimes be discarded, but it is considered best practice to include this, since there is sometimes some ambiguity about tables e.g. if we have two tables - Product.Order and Customer.Order - which order table is being referred to, that in the customer or product schema?

SQL has a number of SQL Statement Types:

- DML or Data Manipulation Language - SELECT, INSERT, UPDATE, DELETE
- DDL or Data Definition Language - CREATE, ALTER, DROP
- DCL or Data Control Language - GRANT, REVOKE, DENY

The course focuses on DML which is typically for working with data.

SELECT statement has a number of possible sub-components:

- FROM [table]
- WHERE [condition for filtering rows]
- GROUP BY [arranges rows by groups]
- HAVING [condition for filtering groups]
- ORDER BY [sorts the output]

Whilst a SQL statement can look like English, it doesn't neccessarily run from top to bottom in terms of the sequence of elements that are run in a query. For instance, the FROM is the first thing that will be run, then the WHERE filter will be run, then we GROUP BY, then SELECT the columns we are interested in and finally ORDER the results. This can be important when running some queries, which will be explored later in the course. When we run a query, it is not an actual table in a database that is return but a set of rows or record set or subset.

1.1.1 Data Types

There are a number of different data types in T-SQL as shown below, which are grouped in to a number of different types.

Exact Numeric	Approximate Numeric	Character
tinyint	float	char
smallint	real	varchar
int		text
bigint		nchar
bit		nvarchar
decimal/numeric		ntext
numeric		
money		
smallmoney		

Figure 1.1: Transact-SQL Data Types

(#fig:Data Types)

This is more relevant when designing a database, however it is useful to know when querying what data type you have in a broad sense - numeric, data, string and so on - as the types will determine what type of combinations can be combined together in expressions e.g. you can concatenate strings or add numbers together, but you can't concatenate a string and a number together.

Sometimes it is neccessary to convert data from one type to another, there are two ways this could happen

- Implicit conversion - compatible data types are automatically converted
- Explicit conversion - requires an explicit function e.g. CAST / TRY_CAST, STR, PARSE ? TRY_PARSE, CONVERT / TRY_CONVERT

The TRY options will attempt a conversion and if it does not work, a NULL will be returned rather than an error in the non-TRY version.

1.1.2 Working with NULLs

There are recognised standards for treating NULL values - ANSI - which says that anythign involving a NULL should return a NULL. There are functions that help us handle NULL values:

- ISNULL(column/variable, value) - Returns *value* (which you can specify) if the column or variable is NULL
- NULLIF(column/variable, value) - Returns NULL if the column or variable is a value - we are almost recoding a non-null to a null
- COALESCE (column/variable1, column/variable2, …) - Returns the value of the first non-NULL column or variable in the list - for instance if contact details, someone might not have an email, so we might want a telephone number, if they don't have that, return an address etc

NULL is used to indicate an unknown or missing value. NULL is **not** equivalent to zero or an empty string.

ISNULL can be used like an IF function in excel, for instance:

```
SELECT name, ISNULL(TRY_CAST(size AS Integer), 0) AS NumericSize
FROM SalesLT.Product;
```

In this instance, if there is a value that will be returned, if not, the NULL value will be returned as a 0.

We can also use a CASE statement to return a value whilst integrating NULL in to our query, e.g.

```
SELECT name,
    CASE size
      WHEN 'S' THEN 'SMALL'
      WHEN 'M' THEN 'MEDIUM'
      WHEN 'L' THEN 'LARGE'
      WHEN 'XL' THEN 'EXTRA LARGE'
      ELSE ISNULL(Size, 'N/A')
    END AS PRODUCT
  FROM SalesLT.Product;
```

1.1.3 Lab Exercises

AdventureWorks Cycles is a company that sells directly to retailers, who then sell products to consumers. Each retailer that is an AdventureWorks customer has provided a named contact for all communication from AdventureWorks.

The sales manager at AdventureWorks has asked you to generate some reports containing details of the company's customers to support a direct sales campaign. Let's start with some basic exploration.

First we display the sales person, the customer's title, surname and telephone number

Table 1.1: Displaying records 1 - 10

name	PRODUCT
HL Road Frame - Black, 58	58
HL Road Frame - Red, 58	58
Sport-100 Helmet, Red	N/A
Sport-100 Helmet, Black	N/A
Mountain Bike Socks, M	MEDIUM
Mountain Bike Socks, L	LARGE
Sport-100 Helmet, Blue	N/A
AWC Logo Cap	N/A
Long-Sleeve Logo Jersey, S	SMALL
Long-Sleeve Logo Jersey, M	MEDIUM

Table 1.2: Displaying records 1 - 10

SalesPerson	CustomerName	Phone
adventure-works\pamela0	Mr. Gee	245-555-0173
adventure-works\david8	Mr. Harris	170-555-0127
adventure-works\jillian0	Ms. Carreras	279-555-0130
adventure-works\jillian0	Ms. Gates	710-555-0173
adventure-works\shu0	Mr. Harrington	828-555-0186
adventure-works\linda3	Ms. Carroll	244-555-0112
adventure-works\shu0	Mr. Gash	192-555-0173
adventure-works\josé1	Ms. Garza	150-555-0127
adventure-works\josé1	Ms. Harding	926-555-0159
adventure-works\garrett1	Mr. Caprio	112-555-0191

```
SELECT SalesPerson, Title + ' ' + LastName AS CustomerName, Phone
FROM SalesLT.Customer;
```

Next we cast the CustomerID column to a VARCHAR and concatenate with the CompanyName column

```
SELECT CAST(CustomerID AS VARCHAR) + ': ' + CompanyName AS CustomerCompany
FROM SalesLT.Customer;
```

The SalesLT.SalesOrderHeader table contains records of sales orders. You have been asked to retrieve data for a report that shows:

The sales order number and revision number in the format () (e.g. SO71774 (2)). The order date converted to ANSI standard format yyyy.mm.dd (e.g. 2015.01.31).

```
SELECT SalesOrderNumber + ' (' + STR(RevisionNumber, 1) + ')' AS OrderRevision,
       CONVERT(NVARCHAR(30), OrderDate, 102) AS OrderDate
FROM SalesLT.SalesOrderHeader;
```

Next we write a query that returns a list of customer names. We use ISNULL to check for middle names and concatenate with FirstName and LastName.

```
SELECT FirstName + ' ' + ISNULL(MiddleName + ' ', '') + LastName
AS CustomerName
FROM SalesLT.Customer;
```

Next, we will imagine that some data has been deleted - customer email addresses - then we try to find

Table 1.3: Displaying records 1 - 10

CustomerCompany
1: A Bike Store
2: Progressive Sports
3: Advanced Bike Components
4: Modular Cycle Systems
5: Metropolitan Sports Supply
6: Aerobic Exercise Company
7: Associated Bikes
10: Rural Cycle Emporium
11: Sharp Bikes
12: Bikes and Motorbikes

Table 1.4: Displaying records 1 - 10

OrderRevision	OrderDate
SO71774 (2)	2008.06.01
SO71776 (2)	2008.06.01
SO71780 (2)	2008.06.01
SO71782 (2)	2008.06.01
SO71783 (2)	2008.06.01
SO71784 (2)	2008.06.01
SO71796 (2)	2008.06.01
SO71797 (2)	2008.06.01
SO71815 (2)	2008.06.01
SO71816 (2)	2008.06.01

Table 1.5: Displaying records 1 - 10

CustomerName
Orlando N. Gee
Keith Harris
Donna F. Carreras
Janet M. Gates
Lucy Harrington
Rosmarie J. Carroll
Dominic P. Gash
Kathleen M. Garza
Katherine Harding
Johnny A. Caprio

Table 1.6: Displaying records 1 - 10

CustomerID	PrimaryContact
1	245-555-0173
2	keith0@adventure-works.com
3	donna0@adventure-works.com
4	janet1@adventure-works.com
5	lucy0@adventure-works.com
6	rosmarie0@adventure-works.com
7	dominic0@adventure-works.com
10	kathleen0@adventure-works.com
11	katherine0@adventure-works.com
12	johnny0@adventure-works.com

contact details in sequence.

```
UPDATE SalesLT.Customer
SET EmailAddress = NULL
WHERE CustomerID % 7 = 1;
```

Next we write a query that returns a list of customer IDs in one column, and a second column named PrimaryContact that contains the email address if known, and otherwise the phone number.

```
SELECT CustomerID, COALESCE(EmailAddress, Phone) AS PrimaryContact
FROM SalesLT.Customer;
```

You have been asked to create a query that returns a list of sales order IDs and order dates with a column named ShippingStatus that contains the text "Shipped" for orders with a known ship date, and "Awaiting Shipment" for orders with no ship date.

Again, we imagine that some data is missing by deleting some first.

```
UPDATE SalesLT.SalesOrderHeader
SET ShipDate = NULL
WHERE SalesOrderID > 71899;
```

```
SELECT SalesOrderID, OrderDate,
  CASE
    WHEN ShipDate IS NULL THEN 'Awaiting Shipment'
    ELSE 'Shipped'
  END AS ShippingStatus
FROM SalesLT.SalesOrderHeader;
```

1.2 Querying Tables with SELECT

1.2.1 Removing Duplicates

If we wanted to know what colours our products are, we would run something like the following.

```
SELECT Color
FROM SalesLT.Product;
```

Here each product has a row and corresponding colour. However, if we just want colour, we are typically interested in removing duplicates to just show what colours we are actually producing. This is achieve using

Table 1.7: Displaying records 1 - 10

SalesOrderID	OrderDate	ShippingStatus
71774	2008-06-01	Shipped
71776	2008-06-01	Shipped
71780	2008-06-01	Shipped
71782	2008-06-01	Shipped
71783	2008-06-01	Shipped
71784	2008-06-01	Shipped
71796	2008-06-01	Shipped
71797	2008-06-01	Shipped
71815	2008-06-01	Shipped
71816	2008-06-01	Shipped

Table 1.8: Displaying records 1 - 10

Color
Black
Red
Red
Black
White
White
Blue
Multi
Multi
Multi

Table 1.9: Displaying records 1 - 10

Color
NA
Black
Blue
Grey
Multi
Red
Silver
Silver/Black
White
Yellow

Table 1.10: Displaying records 1 - 10

Color	Size
NA	NA
Black	NA
Black	38
Black	40
Black	42
Black	44
Black	46
Black	48
Black	52
Black	58

the DISTINCT keyword

```
SELECT DISTINCT Color
FROM SalesLT.Product;
```

These results are DISTINCT at the row level, so if we have two combinitions of columns - say size and colour - then it would be DISTINCT colour and size combinations that appear in the database.

```
SELECT DISTINCT Color, Size
FROM SalesLT.Product;
```

Here is just the size - here IS NULL will return a 'None' if the Size is missing (NA).

```
SELECT DISTINCT ISNULL(Size, 'None') AS Size
FROM SalesLT.Product;
```

1.2.2 Sorting Results

ORDER By is how we sort the results. Any aliased fields used in the SELECT element are visible by ORDER BY. You can oder the results using columns that are not selected in the SELECT clause. You can also ORDER BY multiple columns, either ascending or descending.

We can also just show the top 10 products, e.g. the top 10 most expensive. This is done using the keywork TOP

Table 1.11: Displaying records 1 - 10

Size
38
40
42
44
46
48
50
52
54
56

Table 1.12: Displaying records 1 - 10

Category	Name	ListPrice
6	Road-150 Red, 62	3578.27
6	Road-150 Red, 44	3578.27
6	Road-150 Red, 48	3578.27
6	Road-150 Red, 52	3578.27
6	Road-150 Red, 56	3578.27
5	Mountain-100 Silver, 38	3399.99
5	Mountain-100 Silver, 42	3399.99
5	Mountain-100 Silver, 44	3399.99
5	Mountain-100 Silver, 48	3399.99
5	Mountain-100 Black, 38	3374.99

```
SELECT TOP (10) ProductCategoryID AS Category, Name, ListPrice
FROM SalesLT.Product
ORDER BY ListPrice DESC, Category;
```

We can also use TOP (N) Percent or TOP (N) WITH TIES. If we say wanted the bottom 10 items - say those with the lowest price - there is no 'BOTTOM' Keyword, instead we would sort our data so that they are now in the the order we want - with those we are interested at the top - the use TOP again.

1.2.3 Paging through results

This is achieved through using the OFFSET-FETCH which is an extension of ORDER BY. This might be useful if you have a set of web page results and you want to see certain ones.

You first say how many rows you want to skip using e.g. OFFSET 10 ROWS, then use specify how many rows you are interested in retrieving from the database e.g. FETCH NEXT 10 ROWS ONLY.

1.2.4 Filtering and Using Predicates

We can use the WHERE clause with a number of coniditions or predicates. For instance = (equals) <> (not equals), IN, BETWEEN (is an inclusive statement e.g. BETWEEN 100 AND 200 includes 100 and 200), LIKE, AND, OR and NOT. IN can be more efficient in coding terms when testing multiple attributes, as you just say color IN (red, blue) rather than colour = 'red' OR colour = 'blue'. This becomes more useful

Table 1.13: Displaying records 1 - 10

Name	Color	Size	ProductNumber
HL Road Frame - Black, 58	Black	58	FR-R92B-58
HL Road Frame - Red, 58	Red	58	FR-R92R-58
HL Road Frame - Red, 62	Red	62	FR-R92R-62
HL Road Frame - Red, 44	Red	44	FR-R92R-44
HL Road Frame - Red, 48	Red	48	FR-R92R-48
HL Road Frame - Red, 52	Red	52	FR-R92R-52
HL Road Frame - Red, 56	Red	56	FR-R92R-56
LL Road Frame - Black, 58	Black	58	FR-R38B-58
LL Road Frame - Black, 60	Black	60	FR-R38B-60
LL Road Frame - Black, 62	Black	62	FR-R38B-62

Table 1.14: Displaying records 1 - 10

Name	Color	Size	ProductNumber
HL Road Frame - Black, 58	Black	58	FR-R92B-58
HL Road Frame - Red, 58	Red	58	FR-R92R-58
LL Road Frame - Black, 58	Black	58	FR-R38B-58
LL Road Frame - Red, 58	Red	58	FR-R38R-58
ML Road Frame - Red, 58	Red	58	FR-R72R-58
Road-450 Red, 58	Red	58	BK-R68R-58
Road-650 Red, 58	Red	58	BK-R50R-58
Road-650 Black, 58	Black	58	BK-R50B-58
Road-250 Red, 58	Red	58	BK-R89R-58
Road-250 Black, 58	Black	58	BK-R89B-58

when testing multiple conditions e.g. IN (red, blue) AND size = large - this would have more typing with explicit code for each combition.

Like mathematics, SQL works on PEMDAS sequencing - parenthesis, exponents, multiplication, division, addition, subtraction.

Some examples.

First look for products that start with an FR:

```
SELECT Name, Color, Size, ProductNumber
FROM SalesLT.Product
WHERE ProductNumber LIKE 'FR%';
```

Or we can look for products that end in a 58:

```
SELECT Name, Color, Size, ProductNumber
FROM SalesLT.Product
WHERE ProductNumber LIKE '%58';
```

Or we can use underscores to specify a number of charecters e.g. one _ is one missing charecter. A wildcard (%) would match any number of chars. The figures in brackets then are like regex, so if we want a numeric value between 0 and 9 we use [0-9]. Equally we could use a similar query to find things like email addresses in a string, or email addresses that end in a .co.uk.

```
SELECT Name, Color, Size, ProductNumber
FROM SalesLT.Product
```

Table 1.15: Displaying records 1 - 10

Name	Color	Size	ProductNumber
Road-150 Red, 62	Red	62	BK-R93R-62
Road-150 Red, 44	Red	44	BK-R93R-44
Road-150 Red, 48	Red	48	BK-R93R-48
Road-150 Red, 52	Red	52	BK-R93R-52
Road-150 Red, 56	Red	56	BK-R93R-56
Road-450 Red, 58	Red	58	BK-R68R-58
Road-450 Red, 60	Red	60	BK-R68R-60
Road-450 Red, 44	Red	44	BK-R68R-44
Road-450 Red, 48	Red	48	BK-R68R-48
Road-450 Red, 52	Red	52	BK-R68R-52

Table 1.16: Displaying records 1 - 10

Name
Mountain Bike Socks, M
Mountain Bike Socks, L
ML Road Frame - Red, 44
ML Road Frame - Red, 48
ML Road Frame - Red, 52
ML Road Frame - Red, 58
ML Road Frame - Red, 60
HL Mountain Frame - Silver, 44
HL Mountain Frame - Silver, 48
HL Mountain Frame - Black, 44

```
WHERE ProductNumber LIKE 'BK-_[0-9][0-9]_-[0-9][0-9]';
```

We can use the BETWEEN clause on things like dates to select all products that were removed from sale in 2016:

```
SELECT Name
FROM SalesLT.Product
WHERE SellEndDate BETWEEN '2006/1/1' AND '2006/12/31';
```

Note that it is often useful to order results in the order you want, even if it currently appears it the correct order. Sometimes these queries may change as data or the database does, so it is best to be explicit and use an ORDER BY.

1.2.5 Lab Exercises

You are being told that transportation costs are increasing and you need to identify the heaviest products.

```
-- select the top 10 percent from the Name column
SELECT TOP (10) Percent Name, Weight
FROM SalesLT.Product
-- order by the weight in descending order
ORDER BY Weight DESC;
```

Next, we want to ignore the first 10 records - to page through using offset

Table 1.17: Displaying records 1 - 10

Name	Weight
Touring-3000 Blue, 62	13607.70
Touring-3000 Yellow, 62	13607.70
Touring-3000 Blue, 58	13562.34
Touring-3000 Yellow, 58	13512.45
Touring-3000 Blue, 54	13462.55
Touring-3000 Yellow, 54	13344.62
Touring-3000 Yellow, 50	13213.08
Touring-3000 Blue, 50	13213.08
Touring-3000 Blue, 44	13049.78
Touring-3000 Yellow, 44	13049.78

Table 1.18: Displaying records 1 - 10

Name
Mountain-500 Silver, 52
Mountain-500 Black, 52
Mountain-500 Black, 48
Mountain-500 Silver, 48
Mountain-500 Silver, 44
Mountain-500 Black, 44
Touring-2000 Blue, 60
Mountain-500 Black, 42
Mountain-500 Silver, 42
Touring-2000 Blue, 54

```
SELECT Name
FROM SalesLT.Product
ORDER BY Weight DESC
-- offset 10 rows and get the next 100
OFFSET 10 ROWS FETCH NEXT 100 ROWS ONLY;
```

Next we create a query to find the names, colors, and sizes of the products with a product model ID of 1.

```
-- select the Name, Color, and Size columns
SELECT Name, Color, Size
FROM SalesLT.Product
-- check ProductModelID is 1
WHERE ProductModelID = 1;
```

Now we would like more information on products of certain colors and sizes. We retrieve the product number and name of the products that have a Color of ‘Black’, ‘Red’, or ‘White’ and a Size of ‘S’ or ‘M’.

Table 1.19: 3 records

Name	Color	Size
Classic Vest, S	Blue	S
Classic Vest, M	Blue	M
Classic Vest, L	Blue	L

Table 1.20: Displaying records 1 - 10

ProductNumber	Name
SO-B909-M	Mountain Bike Socks, M
SH-M897-S	Men's Sports Shorts, S
SH-M897-M	Men's Sports Shorts, M
TG-W091-S	Women's Tights, S
TG-W091-M	Women's Tights, M
GL-H102-S	Half-Finger Gloves, S
GL-H102-M	Half-Finger Gloves, M
GL-F110-S	Full-Finger Gloves, S
GL-F110-M	Full-Finger Gloves, M
SH-W890-S	Women's Mountain Shorts, S

Table 1.21: Displaying records 1 - 10

ProductNumber	Name	ListPrice
BK-R93R-62	Road-150 Red, 62	3578.27
BK-R93R-44	Road-150 Red, 44	3578.27
BK-R93R-48	Road-150 Red, 48	3578.27
BK-R93R-52	Road-150 Red, 52	3578.27
BK-R93R-56	Road-150 Red, 56	3578.27
BK-R68R-58	Road-450 Red, 58	1457.99
BK-R68R-60	Road-450 Red, 60	1457.99
BK-R68R-44	Road-450 Red, 44	1457.99
BK-R68R-48	Road-450 Red, 48	1457.99
BK-R68R-52	Road-450 Red, 52	1457.99

```
-- select the ProductNumber and Name columns
SELECT ProductNumber, Name
FROM SalesLT.Product
-- check that Color is one of 'Black', 'Red' or 'White'
-- check that Size is one of 'S' or 'M'
WHERE Color IN ('Black', 'Red', 'White') AND Size IN ('S', 'M');
```

Next you have been asked to retrieve the product number, name, and list price of products that have a product number beginning with 'BK-'.

```
-- select the ProductNumber, Name, and ListPrice columns
SELECT ProductNumber, Name, ListPrice
FROM SalesLT.Product
-- filter for product numbers beginning with BK- using LIKE
WHERE ProductNumber LIKE 'BK-%';
```

Finally, the product manager is interested in a slight variation of the last request regarding product numbers with a particular prefix.

We are interested in products with product number beginning with 'BK-' followed by any character other than 'R', and ending with a '-' followed by any two numerals. Not an R is [^R].

```
-- select the ProductNumber, Name, and ListPrice columns
SELECT ProductNumber, Name, ListPrice
FROM SalesLT.Product
```

Table 1.22: Displaying records 1 - 10

ProductNumber	Name	ListPrice
BK-M82S-38	Mountain-100 Silver, 38	3399.99
BK-M82S-42	Mountain-100 Silver, 42	3399.99
BK-M82S-44	Mountain-100 Silver, 44	3399.99
BK-M82S-48	Mountain-100 Silver, 48	3399.99
BK-M82B-38	Mountain-100 Black, 38	3374.99
BK-M82B-42	Mountain-100 Black, 42	3374.99
BK-M82B-44	Mountain-100 Black, 44	3374.99
BK-M82B-48	Mountain-100 Black, 48	3374.99
BK-M68S-38	Mountain-200 Silver, 38	2319.99
BK-M68S-42	Mountain-200 Silver, 42	2319.99

```
-- filter for ProductNumbers
WHERE ProductNumber LIKE 'BK-[^R]%-[0-9][0-9]';
```

1.3 Querying Tables with Joins

We usually join tables based on primary key - foreign key relationships. We don't run two queries then join, but match at the time of the query. When we are talking about joins, we typically represent them as Venn diagrams.

The convention (ANSI SQL-92) is to specify the JOIN operator in the FROM clause:

SELECT … FROM Table1 JOIN Table 2 ON ;

There is an older standard (ANSI SQL-89) where the tables are joined using commas in the FROM clause and using a WHERE operator, but this can lead to accidental cartesian (aka cross) products.

1.3.1 INNER Joins

INNER Joins are typically the most common join type. It involves a join only where a match is found in both input tables. You can add multiple joins after each other.

Some examples - first a basic inner join where the schema, table and column are explicitly stated.

```
SELECT SalesLT.Product.Name AS ProductName, SalesLT.ProductCategory.Name AS Category
FROM SalesLT.Product
INNER JOIN SalesLT.ProductCategory
ON SalesLT.Product.ProductCategoryID = SalesLT.ProductCategory.ProductCategoryID;
```

Next, we can do the same query but make this less cumbersome by using table aliases.

```
SELECT p.Name AS ProductName, c.Name AS Category
FROM SalesLT.Product AS p
INNER JOIN SalesLT.ProductCategory AS c
on p.ProductCategoryID = c.ProductCategoryID;
```

Next, we look at joining multiple tables, where we want the sales, including order level details, the products in the order and the product details. If we don't specify the join type, the assumption is it is a INNER join, as shown below.

Table 1.23: Displaying records 1 - 10

ProductName	Category
Mountain-100 Silver, 38	Mountain Bikes
Mountain-100 Silver, 42	Mountain Bikes
Mountain-100 Silver, 44	Mountain Bikes
Mountain-100 Silver, 48	Mountain Bikes
Mountain-100 Black, 38	Mountain Bikes
Mountain-100 Black, 42	Mountain Bikes
Mountain-100 Black, 44	Mountain Bikes
Mountain-100 Black, 48	Mountain Bikes
Mountain-200 Silver, 38	Mountain Bikes
Mountain-200 Silver, 42	Mountain Bikes

Table 1.24: Displaying records 1 - 10

ProductName	Category
Mountain-100 Silver, 38	Mountain Bikes
Mountain-100 Silver, 42	Mountain Bikes
Mountain-100 Silver, 44	Mountain Bikes
Mountain-100 Silver, 48	Mountain Bikes
Mountain-100 Black, 38	Mountain Bikes
Mountain-100 Black, 42	Mountain Bikes
Mountain-100 Black, 44	Mountain Bikes
Mountain-100 Black, 48	Mountain Bikes
Mountain-200 Silver, 38	Mountain Bikes
Mountain-200 Silver, 42	Mountain Bikes

Table 1.25: Displaying records 1 - 10

OrderDate	SalesOrderNumber	ProductName	OrderQty	UnitPrice	LineTotal
2008-06-01	SO71774	ML Road Frame-W - Yellow, 48	1	356.898	356.8980
2008-06-01	SO71774	ML Road Frame-W - Yellow, 38	1	356.898	356.8980
2008-06-01	SO71776	Rear Brakes	1	63.900	63.9000
2008-06-01	SO71780	ML Mountain Frame-W - Silver, 42	4	218.454	873.8160
2008-06-01	SO71780	Mountain-400-W Silver, 46	2	461.694	923.3880
2008-06-01	SO71780	Mountain-500 Silver, 52	6	112.998	406.7928
2008-06-01	SO71780	HL Mountain Frame - Silver, 38	2	818.700	1637.4000
2008-06-01	SO71780	Mountain-500 Black, 42	1	323.994	323.9940
2008-06-01	SO71780	LL Mountain Frame - Black, 48	1	149.874	149.8740
2008-06-01	SO71780	HL Mountain Frame - Black, 42	1	809.760	809.7600

Table 1.26: Displaying records 1 - 10

OrderDate	SalesOrderNumber	ProductName	OrderQty	UnitPrice	LineTotal
2008-06-01	SO71774	ML Road Frame-W - Yellow, 48	1	356.898	356.8980
2008-06-01	SO71774	ML Road Frame-W - Yellow, 38	1	356.898	356.8980
2008-06-01	SO71776	Rear Brakes	1	63.900	63.9000
2008-06-01	SO71780	ML Mountain Frame-W - Silver, 42	4	218.454	873.8160
2008-06-01	SO71780	Mountain-400-W Silver, 46	2	461.694	923.3880
2008-06-01	SO71780	Mountain-500 Silver, 52	6	112.998	406.7928
2008-06-01	SO71780	HL Mountain Frame - Silver, 38	2	818.700	1637.4000
2008-06-01	SO71780	Mountain-500 Black, 42	1	323.994	323.9940
2008-06-01	SO71780	LL Mountain Frame - Black, 48	1	149.874	149.8740
2008-06-01	SO71780	HL Mountain Frame - Black, 42	1	809.760	809.7600

```
SELECT oh.OrderDate, oh.SalesOrderNumber, p.Name As ProductName, od.OrderQty, od.UnitPrice, od.LineTotal
FROM SalesLT.SalesOrderHeader AS oh
JOIN SalesLT.SalesOrderDetail AS od
ON od.SalesOrderID = oh.SalesOrderID
JOIN SalesLT.Product AS p
ON od.ProductID = p.ProductID
ORDER BY oh.OrderDate, oh.SalesOrderID, od.SalesOrderDetailID;
```

It is possible to do joins based on more than one criteria, e.g. we could join based on the productID and where the ListPrice is less than the unitprice i.e. there has been a discount.

```
-- Multiple join predicates
SELECT oh.OrderDate, oh.SalesOrderNumber, p.Name As ProductName, od.OrderQty, od.UnitPrice, od.LineTotal
FROM SalesLT.SalesOrderHeader AS oh
JOIN SalesLT.SalesOrderDetail AS od
ON od.SalesOrderID = oh.SalesOrderID
JOIN SalesLT.Product AS p
ON od.ProductID = p.ProductID AND od.UnitPrice < p.ListPrice --Note multiple predicates
ORDER BY oh.OrderDate, oh.SalesOrderID, od.SalesOrderDetailID;
```

Table 1.27: Displaying records 1 - 10

FirstName	LastName	SalesOrderNumber
Orlando	Gee	NA
Keith	Harris	NA
Donna	Carreras	NA
Janet	Gates	NA
Lucy	Harrington	NA
Rosmarie	Carroll	NA
Dominic	Gash	NA
Kathleen	Garza	NA
Katherine	Harding	NA
Johnny	Caprio	NA

1.3.2 OUTER Joins

In an outer join we return all the rows from one table, and any matching rows from the second table. The records in the 'outer' table are preserved, typically we use language such as LEFT, RIGHT and FULL keywords. We are pulling in records from that OUTER table. FULL keeps records from both tables, but are typically not seen in practice. OUTER is OPTIONAL e.g. LEFT JOIN is the same as LEFT OUTER JOIN.

Now some examples. First, we bring up a list of customers, with any matching sales records i.e. we have a list of customers who HAVE bought someting and those WHO HAVE NOT.

```
--Get all customers, with sales orders for those who've bought anything
SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
FROM SalesLT.Customer AS c
LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
ON c.CustomerID = oh.CustomerID
ORDER BY c.CustomerID;
```

Next we can look just for those customers who have not purchased anything using IS NULL.

```
--Return only customers who haven't purchased anything
SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
FROM SalesLT.Customer AS c
LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
ON c.CustomerID = oh.CustomerID
WHERE oh.SalesOrderNumber IS NULL
ORDER BY c.CustomerID;
```

Next we add records from multiple tables. If we add tables on to the chain of tables, having first decleared a left or right join, you have to keep using LEFT joins. You could use an INNER Join but you would loose some records e.g. if you used an INNER join on the second table below, you would loose those records (products) that had never been sold.

Sometimes it is neccessary to go through one table to get to another e.g. to Products -> Orders requires going through order details first. Even if, we are not brining back any tables from the intermediary table.

```
--More than 2 tables
SELECT p.Name As ProductName, oh.SalesOrderNumber
FROM SalesLT.Product AS p
LEFT JOIN SalesLT.SalesOrderDetail AS od
ON p.ProductID = od.ProductID
LEFT JOIN SalesLT.SalesOrderHeader AS oh --Additional tables added to the right must also use a left jo
```

Table 1.28: Displaying records 1 - 10

FirstName	LastName	SalesOrderNumber
Orlando	Gee	NA
Keith	Harris	NA
Donna	Carreras	NA
Janet	Gates	NA
Lucy	Harrington	NA
Rosmarie	Carroll	NA
Dominic	Gash	NA
Kathleen	Garza	NA
Katherine	Harding	NA
Johnny	Caprio	NA

Table 1.29: Displaying records 1 - 10

ProductName	SalesOrderNumber
HL Road Frame - Black, 58	NA
HL Road Frame - Red, 58	NA
Sport-100 Helmet, Red	SO71782
Sport-100 Helmet, Red	SO71783
Sport-100 Helmet, Red	SO71784
Sport-100 Helmet, Red	SO71797
Sport-100 Helmet, Red	SO71902
Sport-100 Helmet, Red	SO71936
Sport-100 Helmet, Red	SO71938
Sport-100 Helmet, Black	SO71782

```
ON od.SalesOrderID = oh.SalesOrderID
ORDER BY p.ProductID;
```

Next another example with multiple tables, but this time the order of the tables is different. We SELECT from the product table, then we LEFT Join the Order details so we can identify products that have never sold, then we LEFT JOIN to the order header table so we can get the order number where we use a left outer join as we are left joining those records to our original table. Then we INNER Join product category, because we are joining product category back to the first table - Products. The final table joins back to our original table, before the outer joins. So whether you need to use an OUTER or INNER join depends on where you wish to place the records based on the current list of tables, however you could reorder the tables to do it differently e.g. do the INNER JOIN first, then then LEFT JOINS.

```
SELECT p.Name As ProductName, c.Name AS Category, oh.SalesOrderNumber
FROM SalesLT.Product AS p
LEFT OUTER JOIN SalesLT.SalesOrderDetail AS od
ON p.ProductID = od.ProductID
LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
ON od.SalesOrderID = oh.SalesOrderID
INNER JOIN SalesLT.ProductCategory AS c --Added to the left, so can use inner join
ON p.ProductCategoryID = c.ProductCategoryID
ORDER BY p.ProductID;
```

** Key Points**

- Use a Left Outer Join to include all rows from the first table and values from matched rows in the

Table 1.30: Displaying records 1 - 10

ProductName	Category	SalesOrderNumber
HL Road Frame - Black, 58	Road Frames	NA
HL Road Frame - Red, 58	Road Frames	NA
Sport-100 Helmet, Red	Helmets	SO71782
Sport-100 Helmet, Red	Helmets	SO71783
Sport-100 Helmet, Red	Helmets	SO71784
Sport-100 Helmet, Red	Helmets	SO71797
Sport-100 Helmet, Red	Helmets	SO71902
Sport-100 Helmet, Red	Helmets	SO71936
Sport-100 Helmet, Red	Helmets	SO71938
Sport-100 Helmet, Black	Helmets	SO71782

second table. Columns in the second table for which no matching rows exist are populated with NULLs.

- Use a Right Outer Join to include all rows from the second table and values from matched rows in the first table. Columns in the first table for which no matching rows exist are populated with NULLs.

- Use a Full Outer Join to include all rows from the first and second tables. Columns in the either table for which no matching rows exist are populated with NULLs.

1.3.3 Cross Joins

Cross Joins create caretesian products - they combine each row from the first table with each row from the second table - to give all possible combinations of products.

One example of it being used would be if we have a list of staff with their attributes, we might want to compare them to the attributes required for all jobs to see which they match, perhaps as an internal job search. Another example is if we have two list of addresses and we calculate the edit distance between the two addresses from the different sources, to try and match those addresses. We would rank the addresses based on their edit distance, with those scoring highest the closest and most likely to be the same address. It can be used to generate test data also.

An example

```
--Call each customer once per product - perhaps not the most realistic example!
SELECT p.Name, c.FirstName, c.LastName, c.Phone
FROM SalesLT.Product as p
CROSS JOIN SalesLT.Customer as c;
```

1.3.4 Self Joins

You might want to join data on to itself but in a different sequence. For instance, we might want to join a person's manager on to their employee(s), but managers are also employees, so this would be a self join. We would use aliases for the table names as we have the same table twice. So when defining a self-join, you must specify an alias for at least one instance of the table being joined.

So our original table looks like this:

```
SELECT *
FROM SalesLT.Employee;
```

Table 1.31: Displaying records 1 - 10

Name	FirstName	LastName	Phone
All-Purpose Bike Stand	Orlando	Gee	245-555-0173
All-Purpose Bike Stand	Keith	Harris	170-555-0127
All-Purpose Bike Stand	Donna	Carreras	279-555-0130
All-Purpose Bike Stand	Janet	Gates	710-555-0173
All-Purpose Bike Stand	Lucy	Harrington	828-555-0186
All-Purpose Bike Stand	Rosmarie	Carroll	244-555-0112
All-Purpose Bike Stand	Dominic	Gash	192-555-0173
All-Purpose Bike Stand	Kathleen	Garza	150-555-0127
All-Purpose Bike Stand	Katherine	Harding	926-555-0159
All-Purpose Bike Stand	Johnny	Caprio	112-555-0191

Table 1.32: 9 records

EmployeeID	EmployeeName	ManagerID
1	adventure-works\david8	8
2	adventure-works\garrett1	1
3	adventure-works\jae0	NA
4	adventure-works\jillian0	3
5	adventure-works\josé1	1
6	adventure-works\linda3	3
7	adventure-works\michael9	9
8	adventure-works\pamela0	3
9	adventure-works\shu0	3

Table 1.33: 9 records

EmployeeName	ManagerName
adventure-works\jae0	NA
adventure-works\garrett1	adventure-works\david8
adventure-works\josé1	adventure-works\david8
adventure-works\linda3	adventure-works\jae0
adventure-works\pamela0	adventure-works\jae0
adventure-works\shu0	adventure-works\jae0
adventure-works\jillian0	adventure-works\jae0
adventure-works\david8	adventure-works\pamela0
adventure-works\michael9	adventure-works\shu0

Table 1.34: Displaying records 1 - 10

CompanyName	SalesOrderId	TotalDue
Professional Sales and Service	71782	43962.7901
Remarkable Bike Store	71935	7330.8972
Bulk Discount Store	71938	98138.2131
Coalition Bike Company	71899	2669.3183
Futuristic Bikes	71895	272.6468
Channel Outlet	71885	608.1766
Aerobic Exercise Company	71915	2361.6403
Vigorous Sports Store	71867	1170.5376
Thrilling Bike Tours	71858	15275.1977
Extreme Riding Supplies	71796	63686.2708

Then the actual self join query - we use a left join as some people i.e. the CEO, will not have a manager.

```
SELECT e.EmployeeName, m.EmployeeName AS ManagerName
FROM SalesLT.Employee AS e
LEFT JOIN SalesLT.Employee AS m
ON e.ManagerID = m.EmployeeID
ORDER BY e.ManagerID;
```

1.3.5 Lab Exercises

Write a query that returns the company name from the Sale.Customer table, the sales order ID and total due from the SalesLT.SalesOrderHeader table.

```
-- select the CompanyName, SalesOrderId, and TotalDue columns from the appropriate tables
SELECT c.CompanyName, oh.SalesOrderId, oh.TotalDue
FROM SalesLT.Customer AS c
JOIN SalesLT.SalesOrderHeader AS oh
-- join tables based on CustomerID
ON c.CustomerID = oh.CustomerID;
```

In order to send out invoices to the customers, we need their addresses. Extend your customer orders query to include the main office address for each customer, including the full street address, city, state or province, postal code, and country or region.

Table 1.35: Displaying records 1 - 10

CompanyName	AddressLine1	AddressLine2	City	StateProvince	PostalCod
Good Toys	99700 Bell Road		Auburn	California	95603
West Side Mart	251 The Metro Center		Wokingham	England	RG41 1QV
Nearby Cycle Shop	Burgess Hill	Edward Way	West Sussex	England	RH15 9UD
Professional Sales and Service	57251 Serene Blvd		Van Nuys	California	91411
Eastside Department Store	9992 Whipple Rd		Union City	California	94587
Action Bicycle Specialists	Warrington Ldc Unit 25/2		Woolston	England	WA1 4SY
Extreme Riding Supplies	Riverside		Sherman Oaks	California	91403
Riding Cycles	Galashiels		Liverpool	England	L4 4HB
Thrifty Parts and Sales	Oxnard Outlet		Oxnard	California	93030
Engineered Bike Systems	123 Camelia Avenue		Oxnard	California	93030

```
SELECT c.CompanyName, a.AddressLine1, ISNULL(a.AddressLine2, '') AS AddressLine2, a.City, a.StateProvin
FROM SalesLT.Customer AS c
-- join the SalesOrderHeader table
JOIN SalesLT.SalesOrderHeader AS oh
ON oh.CustomerID = c.CustomerID
-- join the CustomerAddress table
JOIN SalesLT.CustomerAddress AS ca
-- filter for where the AddressType is 'Main Office'
ON c.CustomerID = ca.CustomerID AND AddressType = 'Main Office'
JOIN SalesLT.Address AS a
ON ca.AddressID = a.AddressID;
```

The sales manager wants a list of all customer companies and their contacts (first name and last name), showing the sales order ID and total due for each order they have placed. Customers who have not placed any orders should be included at the bottom of the list with NULL values for the order ID and total due.

```
-- select the CompanyName, FirstName, LastName, SalesOrderID and TotalDue columns
-- from the appropriate tables
SELECT c.CompanyName, c.FirstName, c.LastName, oh.SalesOrderID, oh.TotalDue
FROM SalesLT.Customer AS c
LEFT JOIN SalesLT.SalesOrderHeader AS oh
-- join based on CustomerID
ON oh.CustomerID  = c.CustomerID
-- order the SalesOrderIDs from highest to lowest
ORDER BY oh.SalesOrderID DESC;
```

A sales employee has noticed that AdventureWorks does not have address information for all customers. Write a query that returns a list of customer IDs, company names, contact names (first name and last name), and phone numbers for customers with no address stored in the database.

```
SELECT c.CompanyName, c.FirstName, c.LastName, c.Phone
FROM SalesLT.Customer AS c
LEFT JOIN SalesLT.CustomerAddress AS ca
-- join based on CustomerID
ON c.CustomerID = ca.CustomerID
-- filter for when the AddressID doesn't exist
WHERE ca.AddressID IS NULL;
```

Some customers have never placed orders, and some products have never been ordered.

Table 1.36: Displaying records 1 - 10

CompanyName	FirstName	LastName	SalesOrderID	TotalDue
Central Bicycle Specialists	Janeth	Esteves	71946	43.0437
Bulk Discount Store	Christopher	Beck	71938	98138.2131
Metropolitan Bicycle Supply	Krishna	Sunkammurali	71936	108597.9536
Remarkable Bike Store	Cory	Booth	71935	7330.8972
The Bicycle Accessories Company	Guy	Gilbert	71923	117.7276
Discount Tours	Melissa	Marple	71920	3293.7761
Essential Bike Works	Linda	Mitchell	71917	45.1995
Aerobic Exercise Company	Rosmarie	Carroll	71915	2361.6403
Many Bikes Store	Jeffrey	Kurtz	71902	81834.9826
Coalition Bike Company	Donald	Blanton	71899	2669.3183

Table 1.37: Displaying records 1 - 10

CompanyName	FirstName	LastName	Phone
A Bike Store	Orlando	Gee	245-555-0173
Progressive Sports	Keith	Harris	170-555-0127
Advanced Bike Components	Donna	Carreras	279-555-0130
Modular Cycle Systems	Janet	Gates	710-555-0173
Metropolitan Sports Supply	Lucy	Harrington	828-555-0186
Aerobic Exercise Company	Rosmarie	Carroll	244-555-0112
Associated Bikes	Dominic	Gash	192-555-0173
Rural Cycle Emporium	Kathleen	Garza	150-555-0127
Sharp Bikes	Katherine	Harding	926-555-0159
Bikes and Motorbikes	Johnny	Caprio	112-555-0191

Table 1.38: Displaying records 1 - 10

CustomerID	ProductID
1	NA
2	NA
3	NA
4	NA
5	NA
6	NA
7	NA
10	NA
11	NA
12	NA

Write a query that returns a column of customer IDs for customers who have never placed an order, and a column of product IDs for products that have never been ordered.

Each row with a customer ID should have a NULL product ID (because the customer has never ordered a product) and each row with a product ID should have a NULL customer ID (because the product has never been ordered by a customer).

```
SELECT c.CustomerID, p.ProductID
FROM SalesLT.Customer AS c
FULL JOIN SalesLT.SalesOrderHeader AS oh
ON c.CustomerID = oh.CustomerID
FULL JOIN SalesLT.SalesOrderDetail AS od
-- join based on the SalesOrderID
ON od.SalesOrderID = oh.SalesOrderID
FULL JOIN SalesLT.Product AS p
-- join based on the ProductID
ON p.ProductID = od.ProductID
-- filter for nonexistent SalesOrderIDs
WHERE oh.SalesOrderID IS NULL
ORDER BY ProductID, CustomerID;
```

1.4 Using Set Operators

A union query is unlike a join, where as a join adds more columns, a union typically adds more rows. You put all records from one query on to the records at the end of another query. NOTE that it is a list of distinct (non duplicate) records - this will be checked every single row in one table and then checks if that record exists across every single row in the other table. Obviously as more tables are added, checking this can become more time consuming and affects performance.

UNION ALL will not undertake this checking, leading to more performance, but some duplicates. John Smith may appear in the employees table but he may also appear in the customers table. Sometimes this is the desired result. In such an instance it makes sense to add a new field using an alias at the time of ther query, such as record type, so we know where this record occurs.

When using Union:

- It is a good idea to use column aliases, so we know which table the column occurs in. However, only aliases in the first query are recognised, so any aliases should be set against the first table

Table 1.39: Displaying records 1 - 10

FirstName	LastName
Catherine	Abel
Kim	Abercrombie
Frances	Adams
Jay	Adams
Samuel	Agcaoili
Robert	Ahlering
Stanley	Alan
Amy	Alberts
Paul	Alcorn
Gregory	Alderson

Table 1.40: Displaying records 1 - 10

FirstName	LastName
Catherine	Abel
Kim	Abercrombie
Frances	Adams
Jay	Adams
Samuel	Agcaoili
Robert	Ahlering
Stanley	Alan
Amy	Alberts
Paul	Alcorn
Gregory	Alderson

- The number of columns must be the same across tables, you can add an additional column but this must be given a specific value or a NULL
- The data types must be approximatley similar i.e. we can do an implicit or explict conversion so they match.

The following example joins the Employees to the customer using Union i.e. removing duplicates.

NOTE: In the following example, views were created (using code provided in the course), before running the queries on the database. Therefore to reproduce these results, it is neccessary to run that same code (Module 4, Union sql file, in the CourseFiles zip) before the following queries will run on the AdventureWorksLT db.

```
SELECT FirstName, LastName
FROM SalesLT.Employees
UNION
SELECT FirstName, LastName
FROM SalesLT.Customers
ORDER BY LastName;
```

This gives 440 rows, if however we use UNION ALL

```
SELECT FirstName, LastName
FROM SalesLT.Employees
UNION ALL
SELECT FirstName, LastName
FROM SalesLT.Customers
ORDER BY LastName;
```

Table 1.41: 5 records

FirstName	LastName	Type
Andy	Carothers	Employee
Donna	Carreras	Customer
Donna	Carreras	Employee
Rosmarie	Carroll	Employee
Joseph	Castellucio	Employee

Table 1.42: 1 records

FirstName	LastName
Donna	Carreras

This results in 441 rows, so there is 1 row which occurs in both - a Donna Carreras appears both in the customer and emplyee table. This could be the same person or indeed just someone with the same name.

It might be useful to know which table the record occurs in, which is where we can introduce a Type column. Not that after the first alias AS Type, subsequent fields e.g. 'Customer' as shown below do not need to be explicitly aliased, however it can make it easier to understand if we do include the AS Type.

```
SELECT FirstName, LastName, 'Employee' AS Type
FROM SalesLT.Employees
UNION
SELECT FirstName, LastName, 'Customer' AS Type
FROM SalesLT.Customers
ORDER BY LastName
OFFSET 100 ROWS FETCH NEXT 5 ROWS ONLY;
```

Note that even though we have used UNION rather than UNION ALL, this would result in the same number of rows as UNION ALL (441), because the presence of the type column means this is no longer the same person - it is not a duplicate record but a unique one. However, it is worth using UNION ALL to limit the number of records that have to be checked for duplicates, so improving peformance.

By default, UNION eliminates duplicate rows. Specify the ALL option to include duplicates (or to avoid the overhead of checking for duplicates when you know in advance that there are none).

1.4.1 INTERSECT and EXCEPT Queries

In INTERSECT we look at only rows that appear in each set - we are looking for the duplicates. Whereas using a JOIN would append columns to the source table, in set theory, we are looking for rows, so in an INTERSECT we are looking for those rows that appear in both datasets.

IN EXCEPT we are looking for records that appear in one source but not the other. In such a scenario, the order of the tables matters.

If we now try and identify the person who appears in both our datasets, we can identify the person directly.

```
SELECT FirstName, LastName
FROM SalesLT.Customers
INTERSECT
SELECT FirstName, LastName
FROM SalesLT.Employees;
```

Table 1.43: Displaying records 1 - 10

FirstName	LastName
Abraham	Swearengin
Ajay	Manchepalli
Alan	Steiner
Alice	Steiner
Andrea	Thomsen
Ben	Miller
Billy	Trent
Brad	Sutton
Caroline	Vicknair
Cecelia	Marshall

Next we want to find the EXCEPT - so let's find customers who are not employees, so we don't try and sel products to employees. There are 104 customers, so we should get 103 rows, as we remove Donna Carreras.

```
SELECT FirstName, LastName
FROM SalesLT.Customers
EXCEPT
SELECT FirstName, LastName
FROM SalesLT.Employees;
```

1.4.2 Lab Exercises

Customers can have two kinds of address: a main office address and a shipping address. The accounts department wants to ensure that the main office address is always used for billing, and have asked you to write a query that clearly identifies the different types of address for each customer.

```
-- select the CompanyName, AddressLine1 columns
-- alias as per the instructions
SELECT CompanyName, AddressLine1, City, 'Billing' AS AddressType
FROM SalesLT.Customer AS c
JOIN SalesLT.CustomerAddress AS ca
-- join based on CustomerID
ON c.CustomerID = ca.CustomerID
-- join another table
JOIN SalesLT.Address AS a
-- join based on AddressID
ON ca.AddressID = a.AddressID
-- filter for where the correct AddressType
WHERE ca.AddressType = 'Main Office';
```

Adapt the query to retrieve the company name, first line of the street address, city, and a column named AddressType with the value 'Shipping' for customers where the address type in the SalesLT.CustomerAddress table is 'Shipping'.

```
SELECT c.CompanyName, a.AddressLine1, a.City, 'Shipping' AS AddressType
FROM SalesLT.Customer AS c
JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
JOIN SalesLT.Address AS a
ON ca.AddressID = a.AddressID
```

Table 1.44: Displaying records 1 - 10

CompanyName	AddressLine1	City	AddressType
Professional Sales and Service	57251 Serene Blvd	Van Nuys	Billing
Riders Company	Tanger Factory	Branch	Billing
Area Bike Accessories	6900 Sisk Road	Modesto	Billing
Bicycle Accessories and Kits	Lewiston Mall	Lewiston	Billing
Valley Bicycle Specialists	Blue Ridge Mall	Kansas City	Billing
Vinyl and Plastic Goods Corporation	No. 25800-130 King Street West	Toronto	Billing
Fun Toys and Bikes	6500 East Grant Road	Tucson	Billing
Great Bikes	Eastridge Mall	Casper	Billing
Valley Toy Store	252851 Rowan Place	Richmond	Billing
Major Sport Suppliers	White Mountain Mall	Rock Springs	Billing

Table 1.45: Displaying records 1 - 10

CompanyName	AddressLine1	City	AddressType
Family's Favorite Bike Shop	26910 Indela Road	Montreal	Shipping
Center Cycle Shop	1318 Lasalle Street	Bothell	Shipping
Safe Cycles Shop	2681 Eagle Peak	Bellevue	Shipping
Modular Cycle Systems	165 North Main	Austin	Shipping
Progressive Sports	7943 Walnut Ave	Renton	Shipping
Hardware Components	99 Front Street	Minneapolis	Shipping
Sample Bike Store	2000 300th Street	Denver	Shipping
Racing Toys	9228 Via Del Sol	Phoenix	Shipping
Elite Bikes	9178 Jumping St.	Dallas	Shipping
All Cycle Shop	8713 Yosemite Ct.	Bothell	Shipping

```
WHERE ca.AddressType = 'Shipping';
```

Next we can union all these records together to create a list of shipping and billing addresses, using UNION ALL.

```
SELECT c.CompanyName, a.AddressLine1, a.City, 'Billing' AS AddressType
FROM SalesLT.Customer AS c
JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
JOIN SalesLT.Address AS a
ON ca.AddressID = a.AddressID
WHERE ca.AddressType = 'Main Office'
-- UNION
UNION ALL
SELECT c.CompanyName, a.AddressLine1, a.City, 'Shipping' AS AddressType
FROM SalesLT.Customer AS c
JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
JOIN SalesLT.Address AS a
ON ca.AddressID = a.AddressID
WHERE ca.AddressType = 'Shipping'
ORDER BY c.CompanyName
OFFSET 10 ROWS;
```

Table 1.46: Displaying records 1 - 10

CompanyName	AddressLine1	City	AddressType
All Cycle Shop	8713 Yosemite Ct.	Bothell	Shipping
All Cycle Shop	25111 228th St Sw	Bothell	Billing
All Seasons Sports Supply	Ohms Road	Houston	Billing
Alpine Ski House	7505 Laguna Boulevard	Elk Grove	Billing
Alternative Vehicles	3307 Evergreen Blvd	Washougal	Billing
Another Bicycle Company	567 Sw Mcloughlin Blvd	Milwaukie	Billing
Area Bike Accessories	6900 Sisk Road	Modesto	Billing
Area Sheet Metal Supply	399 Clearing Green	London	Billing
Associated Bikes	5420 West 22500 South	Salt Lake City	Billing
Authentic Sales and Service	99 Dean Street, Soho	London	Billing

Table 1.47: Displaying records 1 - 10

CompanyName
A Bike Store
A Great Bicycle Company
A Typical Bike Shop
Acceptable Sales & Service
Action Bicycle Specialists
Active Life Toys
Active Systems
Advanced Bike Components
Aerobic Exercise Company
Affordable Sports Equipment

You have created a master list of all customer addresses, but now you have been asked to create filtered lists that show which customers have only a main office address, and which customers have both a main office and a shipping address.

```
SELECT c.CompanyName
FROM SalesLT.Customer AS c
INNER JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
INNER JOIN SalesLT.Address AS a
ON a.AddressID = ca.AddressID
WHERE ca.AddressType = 'Main Office' --Filters out shipping addresses.  This is a table of all Billing
EXCEPT
SELECT c.CompanyName
FROM SalesLT.Customer AS c
INNER JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
INNER JOIN SalesLT.Address AS a
ON a.AddressID = ca.AddressID
WHERE ca.AddressType = 'Shipping'
ORDER BY c.CompanyName;
```

Or the INTERSECT version to identify the company name of each company that appears in a table of customers with a 'Main Office' address, and also in a table of customers with a 'Shipping' address.

Table 1.48: Displaying records 1 - 10

CompanyName
All Cycle Shop
Center Cycle Shop
Elite Bikes
Family's Favorite Bike Shop
Hardware Components
Modular Cycle Systems
Progressive Sports
Racing Toys
Safe Cycles Shop
Sample Bike Store

```
SELECT c.CompanyName
FROM SalesLT.Customer AS c
INNER JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
INNER JOIN SalesLT.Address AS a
ON a.AddressID = ca.AddressID
WHERE ca.AddressType = 'Main Office' --Filters out shipping addresses.  This is a table of all Billing
INTERSECT
SELECT c.CompanyName
FROM SalesLT.Customer AS c
INNER JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
INNER JOIN SalesLT.Address AS a
ON a.AddressID = ca.AddressID
WHERE ca.AddressType = 'Shipping'
ORDER BY c.CompanyName;
```

1.5 Using Functions and Aggregating Data

Now we are looking to not just bring back individual rows but perform calculations, such as aggregations, on those results.

The functions we will look at are shown below.

1.5.1 Scalar Functions

A scalar function returns a single value, not a row or column or table. In database design things can be deterministic or non-deterministic

- Deterministic - we know what the result will be, assuming the data hasn't changed. The data going in and coming out will be the same
- Non-deterministic - we cannot gaurantee what the result will be. For instance if we tested if todays date was less than a value in a table, it would depend on the time and day that the query was run. We can't say what the result will be, based on the data going in/from the db.

A scalar function can be either deterministic or non-deterministic. Scalar is a set of functions, and can do multiple things - date and times, text and image, mathematical, system and system statistical, metadata

Function Category	Description
Scalar	Operate on a single row,
Logical	Scalar functions that com single output
Aggregate	Take one or more input v value
Window	Operate on a window (se
Rowset	Return a virtual table tha Transact-SQL statement

Figure 1.2: Transact-SQL Functions

Table 1.49: Displaying records 1 - 10

SellStartYear	ProductID	Name
2002	680	HL Road Frame - Black, 58
2002	706	HL Road Frame - Red, 58
2005	707	Sport-100 Helmet, Red
2005	708	Sport-100 Helmet, Black
2005	709	Mountain Bike Socks, M
2005	710	Mountain Bike Socks, L
2005	711	Sport-100 Helmet, Blue
2005	712	AWC Logo Cap
2005	713	Long-Sleeve Logo Jersey, S
2005	714	Long-Sleeve Logo Jersey, M

Table 1.50: Displaying records 1 - 10

SellStartYear	SellStartMonth	SellStartDay	SellStartWeekday	ProductID	Name
2002	June	1	Saturday	680	HL Road Frame - Black, 58
2002	June	1	Saturday	706	HL Road Frame - Red, 58
2005	July	1	Friday	707	Sport-100 Helmet, Red
2005	July	1	Friday	708	Sport-100 Helmet, Black
2005	July	1	Friday	709	Mountain Bike Socks, M
2005	July	1	Friday	710	Mountain Bike Socks, L
2005	July	1	Friday	711	Sport-100 Helmet, Blue
2005	July	1	Friday	712	AWC Logo Cap
2005	July	1	Friday	713	Long-Sleeve Logo Jersey, S
2005	July	1	Friday	714	Long-Sleeve Logo Jersey, M

and so on.

We might want to extract the year from a date, for instance to determine when a product was first sold.

```
SELECT YEAR(SellStartDate) SellStartYear, ProductID, Name
FROM SalesLT.Product
ORDER BY SellStartYear;
```

Other examples might include being able to extract certain parts of the date, like the month or day of the week, this can be achived with the DATENAME function

```
SELECT YEAR(SellStartDate) SellStartYear, DATENAME(mm,SellStartDate) SellStartMonth,
       DAY(SellStartDate) SellStartDay, DATENAME(dw, SellStartDate) SellStartWeekday,
       ProductID, Name
FROM SalesLT.Product
ORDER BY SellStartYear;
```

Or we might want to calculate how long a product has been sold, we can use the DATEDIFF function to work this out.

```
SELECT DATEDIFF(yy,SellStartDate, GETDATE()) YearsSold,
       DATEDIFF(mm,SellStartDate, GETDATE()) MonthsSold,
       ProductID, Name
FROM SalesLT.Product
ORDER BY ProductID;
```

Another common example might be to convert text to upper case.

Table 1.51: Displaying records 1 - 10

YearsSold	MonthsSold	ProductID	Name
16	190	680	HL Road Frame - Black, 58
16	190	706	HL Road Frame - Red, 58
13	153	707	Sport-100 Helmet, Red
13	153	708	Sport-100 Helmet, Black
13	153	709	Mountain Bike Socks, M
13	153	710	Mountain Bike Socks, L
13	153	711	Sport-100 Helmet, Blue
13	153	712	AWC Logo Cap
13	153	713	Long-Sleeve Logo Jersey, S
13	153	714	Long-Sleeve Logo Jersey, M

Table 1.52: Displaying records 1 - 10

ProductName
ALL-PURPOSE BIKE STAND
AWC LOGO CAP
BIKE WASH - DISSOLVER
CABLE LOCK
CHAIN
CLASSIC VEST, L
CLASSIC VEST, M
CLASSIC VEST, S
FENDER SET - MOUNTAIN
FRONT BRAKES

```
SELECT UPPER(Name) AS ProductName
FROM SalesLT.Product;
```

We might want to add two strings together but with a space or other text, so we could use the CONCAT function to achieve this.

```
SELECT CONCAT(FirstName + ' ', LastName) AS FullName,
       FirstName, LastName
FROM SalesLT.Customer;
```

Or we might want to return just a specific number of charecters from a string, perhaps there is a structure in the sequence - like the first two chars relate to a product type - so we want to just extract these first two.

```
SELECT Name, ProductNumber, LEFT(ProductNumber, 2) AS ProductType
FROM SalesLT.Product;
```

Or we might have something more complex, where we want to find certain elements of text, we can use CHARINDEX to identify where a char occurs, then combine it with SUBSTRING to extract just a portion of the full string which is n chars before or after a particular charecter.

```
SELECT Name, ProductNumber, LEFT(ProductNumber, 2) AS ProductType,
                            SUBSTRING(ProductNumber,CHARINDEX('-', ProductNumber) + 1, 4) AS ModelCode,
                                        SUBSTRING(ProductNumber, LEN(ProductNumber) - CHARINDEX('-', 
FROM SalesLT.Product;
```

Table 1.53: Displaying records 1 - 10

FullName	FirstName	LastName
Orlando Gee	Orlando	Gee
Keith Harris	Keith	Harris
Donna Carreras	Donna	Carreras
Janet Gates	Janet	Gates
Lucy Harrington	Lucy	Harrington
Rosmarie Carroll	Rosmarie	Carroll
Dominic Gash	Dominic	Gash
Kathleen Garza	Kathleen	Garza
Katherine Harding	Katherine	Harding
Johnny Caprio	Johnny	Caprio

Table 1.54: Displaying records 1 - 10

Name	ProductNumber	ProductType
HL Road Frame - Black, 58	FR-R92B-58	FR
HL Road Frame - Red, 58	FR-R92R-58	FR
Sport-100 Helmet, Red	HL-U509-R	HL
Sport-100 Helmet, Black	HL-U509	HL
Mountain Bike Socks, M	SO-B909-M	SO
Mountain Bike Socks, L	SO-B909-L	SO
Sport-100 Helmet, Blue	HL-U509-B	HL
AWC Logo Cap	CA-1098	CA
Long-Sleeve Logo Jersey, S	LJ-0192-S	LJ
Long-Sleeve Logo Jersey, M	LJ-0192-M	LJ

Table 1.55: Displaying records 1 - 10

Name	ProductNumber	ProductType	ModelCode	SizeCode
HL Road Frame - Black, 58	FR-R92B-58	FR	R92B	58
HL Road Frame - Red, 58	FR-R92R-58	FR	R92R	58
Sport-100 Helmet, Red	HL-U509-R	HL	U509	R
Sport-100 Helmet, Black	HL-U509	HL	U509	
Mountain Bike Socks, M	SO-B909-M	SO	B909	M
Mountain Bike Socks, L	SO-B909-L	SO	B909	L
Sport-100 Helmet, Blue	HL-U509-B	HL	U509	B
AWC Logo Cap	CA-1098	CA	1098	
Long-Sleeve Logo Jersey, S	LJ-0192-S	LJ	0192	S
Long-Sleeve Logo Jersey, M	LJ-0192-M	LJ	0192	M

Table 1.56: Displaying records 1 - 10

Name	Size
HL Road Frame - Black, 58	58
HL Road Frame - Red, 58	58
Sport-100 Helmet, Red	NA
Sport-100 Helmet, Black	NA
Mountain Bike Socks, M	M
Mountain Bike Socks, L	L
Sport-100 Helmet, Blue	NA
AWC Logo Cap	NA
Long-Sleeve Logo Jersey, S	S
Long-Sleeve Logo Jersey, M	M

Table 1.57: Displaying records 1 - 10

Name	NumericSize
HL Road Frame - Black, 58	58
HL Road Frame - Red, 58	58
HL Road Frame - Red, 62	62
HL Road Frame - Red, 44	44
HL Road Frame - Red, 48	48
HL Road Frame - Red, 52	52
HL Road Frame - Red, 56	56
LL Road Frame - Black, 58	58
LL Road Frame - Black, 60	60
LL Road Frame - Black, 62	62

1.5.2 Logical Functions

Logical functions test if someting is true or not i.e. traditional boolean. But we can use logical functions as filters by using CHOOSE for example, you could use IIF which is also a logical, or we could use CASE to achieve the same result.

An example might be if we want to return some numeric sizes, for instance if we have a table of data with different size formats:

```
SELECT Name, Size
FROM SalesLT.Product;
```

We might only be interested in those that have a numeric type, even though the data is in a char data string, 1 in this instance means true:

```
SELECT Name, Size AS NumericSize
FROM SalesLT.Product
WHERE ISNUMERIC(Size) = 1;
```

Or we might want to assign a value to something, like a product type of bike to certain values and other to everything else.

```
SELECT Name, IIF(ProductCategoryID IN (5,6,7), 'Bike', 'Other') AS ProductType
FROM SalesLT.Product;
```

Or a more complicated query where we assign mutiple product types.

Table 1.58: Displaying records 1 - 10

Name	ProductType
HL Road Frame - Black, 58	Other
HL Road Frame - Red, 58	Other
Sport-100 Helmet, Red	Other
Sport-100 Helmet, Black	Other
Mountain Bike Socks, M	Other
Mountain Bike Socks, L	Other
Sport-100 Helmet, Blue	Other
AWC Logo Cap	Other
Long-Sleeve Logo Jersey, S	Other
Long-Sleeve Logo Jersey, M	Other

Table 1.59: Displaying records 1 - 10

ProductName	Category	ProductType
Mountain-100 Silver, 38	Mountain Bikes	Bikes
Mountain-100 Silver, 42	Mountain Bikes	Bikes
Mountain-100 Silver, 44	Mountain Bikes	Bikes
Mountain-100 Silver, 48	Mountain Bikes	Bikes
Mountain-100 Black, 38	Mountain Bikes	Bikes
Mountain-100 Black, 42	Mountain Bikes	Bikes
Mountain-100 Black, 44	Mountain Bikes	Bikes
Mountain-100 Black, 48	Mountain Bikes	Bikes
Mountain-200 Silver, 38	Mountain Bikes	Bikes
Mountain-200 Silver, 42	Mountain Bikes	Bikes

```
SELECT prd.Name AS ProductName, cat.Name AS Category,
      CHOOSE (cat.ParentProductCategoryID, 'Bikes','Components','Clothing','Accessories') AS ProductTyp
FROM SalesLT.Product AS prd
JOIN SalesLT.ProductCategory AS cat
ON prd.ProductCategoryID = cat.ProductCategoryID;
```

1.5.3 Window Functions

A window in this context is a set of rows - a window in to the database or table. For instance, we might want to RANK a set (or window) of data. The query below will pull out the top 100 more expensive products (ordered by list price) then creates a ranking based on this list price, then orders the results by this ranking.

```
SELECT TOP(100) ProductID, Name, ListPrice,
    RANK() OVER(ORDER BY ListPrice DESC) AS RankByPrice
FROM SalesLT.Product AS p
ORDER BY RankByPrice;
```

Or we might want to partition the results by product category, similar to grouping by product category. This will result in a ranking for each product category. The results might look a little odd at first glance.

```
SELECT c.Name AS Category, p.Name AS Product, ListPrice,
    RANK() OVER(PARTITION BY c.Name ORDER BY ListPrice DESC) AS RankByPrice
FROM SalesLT.Product AS p
```

Table 1.60: Displaying records 1 - 10

ProductID	Name	ListPrice	RankByPrice
749	Road-150 Red, 62	3578.27	1
750	Road-150 Red, 44	3578.27	1
751	Road-150 Red, 48	3578.27	1
752	Road-150 Red, 52	3578.27	1
753	Road-150 Red, 56	3578.27	1
771	Mountain-100 Silver, 38	3399.99	6
772	Mountain-100 Silver, 42	3399.99	6
773	Mountain-100 Silver, 44	3399.99	6
774	Mountain-100 Silver, 48	3399.99	6
775	Mountain-100 Black, 38	3374.99	10

Table 1.61: Displaying records 1 - 10

Category	Product	ListPrice	RankByPrice
Bib-Shorts	Men's Bib-Shorts, S	89.99	1
Bib-Shorts	Men's Bib-Shorts, M	89.99	1
Bib-Shorts	Men's Bib-Shorts, L	89.99	1
Bike Racks	Hitch Rack - 4-Bike	120.00	1
Bike Stands	All-Purpose Bike Stand	159.00	1
Bottles and Cages	Mountain Bottle Cage	9.99	1
Bottles and Cages	Road Bottle Cage	8.99	2
Bottles and Cages	Water Bottle - 30 oz.	4.99	3
Bottom Brackets	HL Bottom Bracket	121.49	1
Bottom Brackets	ML Bottom Bracket	101.24	2

```
JOIN SalesLT.ProductCategory AS c
ON p.ProductCategoryID = c.ProductcategoryID
ORDER BY Category, RankByPrice;
```

1.5.4 Aggregate Functions

There are a lot of aggregate functions including some statistical functions like s.d. as well as some more standard things like MIN, MAX, SUM etc over a set of data. We can use GROUP BY.

First we can get some headline stats of what is in the table.

```
SELECT COUNT(*) AS Products, COUNT(DISTINCT ProductCategoryID) AS Categories, AVG(ListPrice) AS Average
FROM SalesLT.Product;
```

Or we might be interested in summary figures for just one product type, like bikes. Note that we use COUNT then the specific item, since using COUNT(*) would return the number of rows, which may include duplicates.

Table 1.62: 1 records

Products	Categories	AveragePrice
296	37	742.1235

Table 1.63: 1 records

BikeModels	AveragePrice
97	1586.737

Table 1.64: 3 records

Salesperson	SalesRevenue
adventure-works\jae0	518096.4
adventure-works\linda3	209219.8
adventure-works\shu0	138116.9

```
SELECT COUNT(p.ProductID) BikeModels, AVG(p.ListPrice) AveragePrice
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory AS c
ON p.ProductCategoryID = c.ProductCategoryID
WHERE c.Name LIKE '%Bikes';
```

However, our aggregate functions are currently returnign totals. In practice, we may want to return figures by groups, so we need to use GROUP BY in our queries. When we use a GROUP BY which groups up the results, we have to group up everything in the select row that isn't being aggregated. You can't have something in the SELECT which is neither being aggregated nor being grouped. As we are grouping the results, we don't see individual rows of data anymore.

The following example calulates the sales revenue for each sales person.

```
SELECT c.Salesperson, SUM(oh.SubTotal) SalesRevenue
FROM SalesLT.Customer c
JOIN SalesLT.SalesOrderHeader oh
ON c.CustomerID = oh.CustomerID
GROUP BY c.Salesperson
ORDER BY SalesRevenue DESC;
```

Could it be that some sales people have no sales? If so, we need to return values where the total is NULL, by assigning NULL a value of zero. WE also use the LEFT JOIN rather than the INNER JOIN previously, we get sales people included who haven't sold anything.

```
SELECT c.Salesperson, ISNULL(SUM(oh.SubTotal), 0.00) SalesRevenue
FROM SalesLT.Customer c
LEFT JOIN SalesLT.SalesOrderHeader oh
ON c.CustomerID = oh.CustomerID
GROUP BY c.Salesperson
ORDER BY SalesRevenue DESC;
```

Or we might want to know the number of customers each sales person has.

```
SELECT Salesperson, COUNT(CustomerID) Customers
FROM SalesLT.Customer
GROUP BY Salesperson
ORDER BY Salesperson;
```

We might also want to know the sales Revenue by sales person and customer. Note that we now includer the customer details, we cannot use the CUSTOMER alias from the SELECT query in the GROUP BY, as the SELECT query is run after the GROUP BY.

Table 1.65: 9 records

Salesperson	SalesRevenue
adventure-works\jae0	518096.4
adventure-works\linda3	209219.8
adventure-works\shu0	138116.9
adventure-works\michael9	0.0
adventure-works\pamela0	0.0
adventure-works\jillian0	0.0
adventure-works\josé1	0.0
adventure-works\david8	0.0
adventure-works\garrett1	0.0

Table 1.66: 9 records

Salesperson	Customers
adventure-works\david8	73
adventure-works\garrett1	78
adventure-works\jae0	78
adventure-works\jillian0	148
adventure-works\josé1	142
adventure-works\linda3	71
adventure-works\michael9	32
adventure-works\pamela0	74
adventure-works\shu0	151

```
SELECT c.Salesperson, CONCAT(c.FirstName +' ', c.LastName) AS Customer, ISNULL(SUM(oh.SubTotal), 0.00) S
FROM SalesLT.Customer c
LEFT JOIN SalesLT.SalesOrderHeader oh
ON c.CustomerID = oh.CustomerID
GROUP BY c.Salesperson, CONCAT(c.FirstName +' ', c.LastName)
ORDER BY SalesRevenue DESC, Customer;
```

Or the number of products in each category.

```
SELECT c.Name AS Category, COUNT(p.ProductID) AS Products
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory AS c
ON p.ProductCategoryID = c.ProductCategoryID
GROUP BY c.Name
ORDER BY Category;
```

1.5.5 Filtering Groups

If we want to filter, we can use the WHERE clause as we have seen in some previous examples. However, most of the time we filter with the HAVING caluse. **HAVING will filter the resutls RATHER than the input**.

Having clause provides a search condition that each group must satisfy, for instance where the number of orders by a customer is greater than 10.

In the example below, we are interested in finding out which sales people have more than 100 customers.

Table 1.67: Displaying records 1 - 10

Salesperson	Customer	SalesRevenue
adventure-works\jae0	Terry Eminhizer	108561.83
adventure-works\jae0	Krishna Sunkammurali	98278.69
adventure-works\jae0	Christopher Beck	88812.86
adventure-works\linda3	Kevin Liu	83858.43
adventure-works\jae0	Jon Grande	78029.69
adventure-works\shu0	Jeffrey Kurtz	74058.81
adventure-works\jae0	Rebecca Laszlo	63980.99
adventure-works\linda3	Anthony Chor	57634.63
adventure-works\shu0	Frank Campbell	41622.05
adventure-works\linda3	Catherine Abel	39785.33

Table 1.68: Displaying records 1 - 10

Category	Products
Bib-Shorts	3
Bike Racks	1
Bike Stands	1
Bottles and Cages	3
Bottom Brackets	3
Brakes	2
Caps	1
Chains	1
Cleaners	1
Cranksets	3

Table 1.69: 3 records

Salesperson	Customers
adventure-works\jillian0	148
adventure-works\josé1	142
adventure-works\shu0	151

Table 1.70: Displaying records 1 - 10

ProductID	ProductName	ApproxWeight
680	HL ROAD FRAME - BLACK, 58	1016
706	HL ROAD FRAME - RED, 58	1016
707	SPORT-100 HELMET, RED	NA
708	SPORT-100 HELMET, BLACK	NA
709	MOUNTAIN BIKE SOCKS, M	NA
710	MOUNTAIN BIKE SOCKS, L	NA
711	SPORT-100 HELMET, BLUE	NA
712	AWC LOGO CAP	NA
713	LONG-SLEEVE LOGO JERSEY, S	NA
714	LONG-SLEEVE LOGO JERSEY, M	NA

```
SELECT Salesperson, COUNT(CustomerID) Customers
FROM SalesLT.Customer
GROUP BY Salesperson
HAVING COUNT(CustomerID) > 100
ORDER BY Salesperson;
```

** Key Points **

- You can use GROUP BY with aggregate functions to return aggregations grouped by one or more columns or expressions.
- All columns in the SELECT clause that are not aggregate function expressions must be included in a GROUP BY clause.
- The order in which columns or expressions are listed in the GROUP BY clause determines the grouping hierarchy.
- You can filter the groups that are included in the query results by specifying a HAVING clause.

1.5.6 Lab Exercises

Write a query to return the product ID of each product, together with the product name formatted as upper case and a column named ApproxWeight with the weight of each product rounded to the nearest whole unit.

```
SELECT ProductID, UPPER(Name) AS ProductName, ROUND(WEIGHT, 0) AS ApproxWeight
FROM SalesLT.Product;
```

Extend your query to include columns named SellStartYear and SellStartMonth containing the year and month in which AdventureWorks started selling each product. The month should be displayed as the month name (e.g. ‘January’).

Table 1.71: Displaying records 1 - 10

ProductID	ProductName	ApproxWeight	SellStartYear	SellStartMonth
680	HL ROAD FRAME - BLACK, 58	1016	2002	June
706	HL ROAD FRAME - RED, 58	1016	2002	June
707	SPORT-100 HELMET, RED	NA	2005	July
708	SPORT-100 HELMET, BLACK	NA	2005	July
709	MOUNTAIN BIKE SOCKS, M	NA	2005	July
710	MOUNTAIN BIKE SOCKS, L	NA	2005	July
711	SPORT-100 HELMET, BLUE	NA	2005	July
712	AWC LOGO CAP	NA	2005	July
713	LONG-SLEEVE LOGO JERSEY, S	NA	2005	July
714	LONG-SLEEVE LOGO JERSEY, M	NA	2005	July

Table 1.72: Displaying records 1 - 10

ProductID	ProductName	ApproxWeight	SellStartYear	SellStartMonth	ProductType
680	HL ROAD FRAME - BLACK, 58	1016	2002	June	FR
706	HL ROAD FRAME - RED, 58	1016	2002	June	FR
707	SPORT-100 HELMET, RED	NA	2005	July	HL
708	SPORT-100 HELMET, BLACK	NA	2005	July	HL
709	MOUNTAIN BIKE SOCKS, M	NA	2005	July	SO
710	MOUNTAIN BIKE SOCKS, L	NA	2005	July	SO
711	SPORT-100 HELMET, BLUE	NA	2005	July	HL
712	AWC LOGO CAP	NA	2005	July	CA
713	LONG-SLEEVE LOGO JERSEY, S	NA	2005	July	LJ
714	LONG-SLEEVE LOGO JERSEY, M	NA	2005	July	LJ

```
SELECT ProductID, UPPER(Name) AS ProductName, ROUND(Weight, 0) AS ApproxWeight,
       YEAR(SellStartDate) as SellStartYear,
       DATENAME(m, SellStartDate) as SellStartMonth
FROM SalesLT.Product;
```

Extend your query to include a column named ProductType that contains the leftmost two characters from the product number.

```
SELECT ProductID, UPPER(Name) AS ProductName, ROUND(Weight, 0) AS ApproxWeight,
       YEAR(SellStartDate) as SellStartYear,
       DATENAME(m, SellStartDate) as SellStartMonth,
       LEFT(ProductNumber, 2) AS ProductType
FROM SalesLT.Product;
```

Extend your query to filter the product returned so that only products with a numeric size are included.

```
SELECT ProductID, UPPER(Name) AS ProductName, ROUND(Weight, 0) AS ApproxWeight,
       YEAR(SellStartDate) as SellStartYear,
       DATENAME(m, SellStartDate) as SellStartMonth,
       LEFT(ProductNumber, 2) AS ProductType
FROM SalesLT.Product
WHERE ISNUMERIC(SIZE) = 1;
```

Write a query that returns a list of company names with a ranking of their place in a list of highest TotalDue values from the SalesOrderHeader table.

Table 1.73: Displaying records 1 - 10

ProductID	ProductName	ApproxWeight	SellStartYear	SellStartMonth	ProductType
680	HL ROAD FRAME - BLACK, 58	1016	2002	June	FR
706	HL ROAD FRAME - RED, 58	1016	2002	June	FR
717	HL ROAD FRAME - RED, 62	1043	2005	July	FR
718	HL ROAD FRAME - RED, 44	962	2005	July	FR
719	HL ROAD FRAME - RED, 48	980	2005	July	FR
720	HL ROAD FRAME - RED, 52	998	2005	July	FR
721	HL ROAD FRAME - RED, 56	1016	2005	July	FR
722	LL ROAD FRAME - BLACK, 58	1116	2005	July	FR
723	LL ROAD FRAME - BLACK, 60	1125	2005	July	FR
724	LL ROAD FRAME - BLACK, 62	1134	2005	July	FR

Table 1.74: Displaying records 1 - 10

CompanyName	Revenue	RankByRevenue
Action Bicycle Specialists	119960.82	1
Metropolitan Bicycle Supply	108597.95	2
Bulk Discount Store	98138.21	3
Eastside Department Store	92663.56	4
Riding Cycles	86222.81	5
Many Bikes Store	81834.98	6
Instruments and Parts Company	70698.99	7
Extreme Riding Supplies	63686.27	8
Trailblazing Sports	45992.37	9
Professional Sales and Service	43962.79	10

```
SELECT CompanyName, TotalDue AS Revenue,
       RANK() OVER (ORDER BY TotalDue DESC) AS RankByRevenue
FROM SalesLT.SalesOrderHeader AS SOH
LEFT JOIN SalesLT.Customer AS C
ON SOH.CustomerID = C.CustomerID;
```

Write a query to retrieve a list of the product names and the total revenue calculated as the sum of the LineTotal from the SalesLT.SalesOrderDetail table, with the results sorted in descending order of total revenue.

```
SELECT Name, SUM(LineTotal) AS TotalRevenue
FROM SalesLT.SalesOrderDetail AS SOD
LEFT JOIN SalesLT.Product AS P
ON SOD.ProductID = p.ProductID
GROUP BY P.Name
ORDER BY TotalRevenue DESC;
```

Modify the previous query to include sales totals for products that have a list price of more than 1000.

```
SELECT Name, SUM(LineTotal) AS TotalRevenue
FROM SalesLT.SalesOrderDetail AS SOD
JOIN SalesLT.Product AS P
ON SOD.ProductID = P.ProductID
WHERE ListPrice > 1000
GROUP BY P.Name
```

Table 1.75: Displaying records 1 - 10

Name	TotalRevenue
Touring-1000 Blue, 60	37191.49
Mountain-200 Black, 42	37178.84
Road-350-W Yellow, 48	36486.24
Mountain-200 Black, 38	35801.84
Touring-1000 Yellow, 60	23413.47
Touring-1000 Blue, 50	22887.07
Mountain-200 Silver, 42	20879.91
Road-350-W Yellow, 40	20411.88
Mountain-200 Black, 46	19277.92
Road-350-W Yellow, 42	18692.52

Table 1.76: Displaying records 1 - 10

Name	TotalRevenue
Touring-1000 Blue, 60	37191.49
Mountain-200 Black, 42	37178.84
Road-350-W Yellow, 48	36486.24
Mountain-200 Black, 38	35801.84
Touring-1000 Yellow, 60	23413.47
Touring-1000 Blue, 50	22887.07
Mountain-200 Silver, 42	20879.91
Road-350-W Yellow, 40	20411.88
Mountain-200 Black, 46	19277.92
Road-350-W Yellow, 42	18692.52

```
ORDER BY TotalRevenue DESC;
```

Modify the previous query to only include products with total sales greater than 20000.

```
SELECT Name, SUM(LineTotal) AS TotalRevenue
FROM SalesLT.SalesOrderDetail AS SOD
JOIN SalesLT.Product AS P
ON SOD.ProductID = P.ProductID
WHERE P.ListPrice > 1000
GROUP BY P.Name
-- add having clause as per instructions
HAVING SUM(LineTotal) > 20000
ORDER BY TotalRevenue DESC;
```

1.6 Sub-queries and Apply

Subqueries are queries within queries (nested). The results of the inner or nested query are pasted to the outer query and behave much like an expression.

Our subquery might be scalr, for instance we might have a single value which is passed to the outer query. So we might be interested in the last order (the maxium id) and the details from that order, where the information is held in two different tables.

Table 1.77: 8 records

Name	TotalRevenue
Touring-1000 Blue, 60	37191.49
Mountain-200 Black, 42	37178.84
Road-350-W Yellow, 48	36486.24
Mountain-200 Black, 38	35801.84
Touring-1000 Yellow, 60	23413.47
Touring-1000 Blue, 50	22887.07
Mountain-200 Silver, 42	20879.91
Road-350-W Yellow, 40	20411.88

Table 1.78: 1 records

salesorderid	productid	unitprice	orderqty
71946	916	31.584	1

```
SELECT salesorderid, productid, unitprice, orderqty
From SalesLT.SalesOrderDetail
WHERE salesorderid =
    (SELECT MAX(salesorderid) AS lastorder
    FROM SalesLT.SalesOrderHeader);
```

The subquery is in brackets, and the system will treat whatever is in brackets as an individual unit.

In a multi-valued query, mutiple values are returned but as a single column set to the outer query. So we might be interested in every customer in Mexico. One will get returned from the subquery is a one dimensional array - a vector.

1.6.1 Self Contained or Correlated Query

Queries we have looked at so far are self contained queries. The sub-query parts works, then it supplies its value to the outer query. Correlated sub-queries refer to elements of tables used in the outer query. Because the sub-query is searching for values in the outer query, this can be resource intensive. We might, for instance, make a reference to an employee number in the subquery, but that number is dependent on the result of the outer query, so the subqueriy works with the outer query.

As with sub-queries generally, it is best to build the query up iteratively, so you can a) check the query is working and b) check your final sub-query results with those of the individual components to make sure it is doing what it should be. However, this is more difficult when using a correlated query, since there is a dependency.

So our goal is for each customer list all sales on the last day that they made a sale.

So first, lets get a list of customers with their orders and dates.

```
SELECT CustomerID, SalesOrderID, OrderDate
FROM SalesLT.SalesOrderHeader AS SO1
ORDER BY CustomerID,OrderDate
```

Then let's say we want just the latest (max) order date in the db, for each customer ID we have, so we have their individual last order details.

Table 1.79: Displaying records 1 - 10

CustomerID	SalesOrderID	OrderDate
29485	71782	2008-06-01
29531	71935	2008-06-01
29546	71938	2008-06-01
29568	71899	2008-06-01
29584	71895	2008-06-01
29612	71885	2008-06-01
29638	71915	2008-06-01
29644	71867	2008-06-01
29653	71858	2008-06-01
29660	71796	2008-06-01

Table 1.80: Displaying records 1 - 10

CustomerID	SalesOrderID	OrderDate
29485	71782	2008-06-01
29531	71935	2008-06-01
29546	71938	2008-06-01
29568	71899	2008-06-01
29584	71895	2008-06-01
29612	71885	2008-06-01
29638	71915	2008-06-01
29644	71867	2008-06-01
29653	71858	2008-06-01
29660	71796	2008-06-01

```
SELECT CustomerID, SalesOrderID, OrderDate
FROM SalesLT.SalesOrderHeader AS SO1
WHERE orderdate =
    (SELECT MAX(orderdate)
FROM SalesLT.SalesOrderHeader AS SO2
WHERE SO2.CustomerID = SO1.CustomerID)
ORDER BY CustomerID;
```

1.6.2 The Apply Operator

Using an apply operator and a table function can achieve something similar to a correlated query, but in less complicated code. A CROSS APPLY applies the right table expression to each row in the left table, a bit like the correlated sub-query, with CROSS APPLY being similar to a CROSS JOIN.

Similarily OUTER APPLY adds rows for those with NULL in columns for the right table, similar to LEFT OUTER JOIN. In these instances, the right table is the table returned by the APPLY function.

The first thing we need to do is to create a function. Notice we are passing the function the SalesOrderID, and the most expensive item in the order gets passed back.

```
-- Setup
CREATE FUNCTION SalesLT.udfMaxUnitPrice (@SalesOrderID int)
RETURNS TABLE
AS
```

Table 1.81: 1 records

Text
– Setup CREATE FUNCTION SalesLT.udfMaxUnitPrice (@SalesOrderID int) RETURNS TABLE AS RETURN SELEC

Table 1.82: Displaying records 1 - 10

SalesOrderID	MaxUnitPrice
71774	356.898
71774	356.898
71776	63.900
71780	1391.994
71780	1391.994
71780	1391.994
71780	1391.994
71780	1391.994
71780	1391.994
71780	1391.994

```
RETURN
SELECT SalesOrderID,Max(UnitPrice) as MaxUnitPrice FROM
SalesLT.SalesOrderDetail
WHERE SalesOrderID=@SalesOrderID
GROUP BY SalesOrderID;
```

To see what code created a function, to help understand what it does at a later time or if created by someone else, we can use the sp_helptext command.

```
sp_helptext 'saleslt.udfmaxunitprice'
```

Then we can use that query to get the results. Note this could have been achieved with a correlated sub-query, but this is perahps a little easier to interpret.

```
SELECT SOH.SalesOrderID, MUP.MaxUnitPrice
FROM SalesLT.SalesOrderDetail AS SOH
CROSS APPLY SalesLT.udfMaxUnitPrice(SOH.SalesOrderID) AS MUP
ORDER BY SOH.SalesOrderID;
```

** Key Points **

- The APPLY operator enables you to execute a table-valued function for each row in a rowset returned by a SELECT statement. Conceptually, this approach is similar to a correlated subquery.
- CROSS APPLY returns matching rows, similar to an inner join. OUTER APPLY returns all rows in the original SELECT query results with NULL values for rows where no match was found.

1.6.3 Lab Exercises

AdventureWorks products each have a standard cost that indicates the cost of manufacturing the product, and a list price that indicates the recommended selling price for the product. This data is stored in the SalesLT.Product table.

Whenever a product is ordered, the actual unit price at which it was sold is also recorded in the SalesLT.SalesOrderDetail table.

Table 1.83: Displaying records 1 - 10

ProductID	Name	ListPrice
680	HL Road Frame - Black, 58	1431.50
706	HL Road Frame - Red, 58	1431.50
717	HL Road Frame - Red, 62	1431.50
718	HL Road Frame - Red, 44	1431.50
719	HL Road Frame - Red, 48	1431.50
720	HL Road Frame - Red, 52	1431.50
721	HL Road Frame - Red, 56	1431.50
731	ML Road Frame - Red, 44	594.83
732	ML Road Frame - Red, 48	594.83
733	ML Road Frame - Red, 52	594.83

Table 1.84: 7 records

ProductID	Name	ListPrice
810	HL Mountain Handlebars	120.27
813	HL Road Handlebars	120.27
876	Hitch Rack - 4-Bike	120.00
894	Rear Derailleur	121.46
907	Rear Brakes	106.50
948	Front Brakes	106.50
996	HL Bottom Bracket	121.49

Use subqueries to compare the cost and list prices for each product with the unit prices charged in each sale.

```
SELECT ProductID, Name, ListPrice
FROM SalesLT.Product
WHERE ListPrice >
    (SELECT AVG(UnitPRice) FROM SalesLT.SalesOrderDetail)
ORDER BY ProductID;
```

AdventureWorks is interested in finding out which products are being sold at a loss. Retrieve the product ID, name, and list price for each product where the list price is 100 or more, and the product has been sold for (strictly) less than 100.

```
SELECT ProductID, Name, ListPrice
FROM SalesLT.Product
WHERE ProductID IN
  (SELECT ProductID FROM SalesLT.SalesOrderDetail
   WHERE UnitPrice < 100)
AND ListPrice >= 100
ORDER BY ProductID;
```

In order to get an idea of how many products are selling above or below list price, you want to gather some aggregate product data. Retrieve the product ID, name, cost, and list price for each product along with the average unit price for which that product has been sold.

```
SELECT ProductID, Name, StandardCost, ListPrice,
    (SELECT AVG(UnitPrice)
    FROM SalesLT.SalesOrderDetail AS SOD
    WHERE P.ProductID = SOD.ProductID) AS AvgSellingPrice
FROM SalesLT.Product AS P
```

Table 1.85: Displaying records 1 - 10

ProductID	Name	StandardCost	ListPrice	AvgSellingPrice
680	HL Road Frame - Black, 58	1059.3100	1431.50	NA
706	HL Road Frame - Red, 58	1059.3100	1431.50	NA
707	Sport-100 Helmet, Red	13.0863	34.99	20.9940
708	Sport-100 Helmet, Black	13.0863	34.99	20.6441
709	Mountain Bike Socks, M	3.3963	9.50	NA
710	Mountain Bike Socks, L	3.3963	9.50	NA
711	Sport-100 Helmet, Blue	13.0863	34.99	20.7440
712	AWC Logo Cap	6.9223	8.99	5.3740
713	Long-Sleeve Logo Jersey, S	38.4923	49.99	NA
714	Long-Sleeve Logo Jersey, M	38.4923	49.99	29.9940

Table 1.86: Displaying records 1 - 10

ProductID	Name	StandardCost	ListPrice	AvgSellingPrice
712	AWC Logo Cap	6.9223	8.99	5.374
714	Long-Sleeve Logo Jersey, M	38.4923	49.99	29.994
715	Long-Sleeve Logo Jersey, L	38.4923	49.99	29.744
716	Long-Sleeve Logo Jersey, XL	38.4923	49.99	29.994
717	HL Road Frame - Red, 62	868.6342	1431.50	858.900
718	HL Road Frame - Red, 44	868.6342	1431.50	858.900
722	LL Road Frame - Black, 58	204.6251	337.22	202.332
738	LL Road Frame - Black, 52	204.6251	337.22	202.332
792	Road-250 Red, 58	1554.9479	2443.35	1466.010
793	Road-250 Black, 44	1554.9479	2443.35	1466.010

```
ORDER BY P.ProductID;
```

AdventureWorks is interested in finding out which products are costing more than they're being sold for, on average. Filter the query for the previous exercise to include only products where the cost is higher than the average selling price.

```
SELECT ProductID, Name, StandardCost, ListPrice,
    (SELECT AVG(UnitPrice)
     FROM SalesLT.SalesOrderDetail AS SOD
     WHERE P.ProductID = SOD.ProductID) AS AvgSellingPrice
FROM SalesLT.Product AS P
WHERE StandardCost >
    (SELECT AVG(UnitPrice)
     FROM SalesLT.SalesOrderDetail AS SOD
     WHERE P.ProductID = SOD.ProductID)
ORDER BY P.ProductID;
```

The AdventureWorksLT database includes a table-valued user-defined function named dbo.ufnGetCustomerInformation. Use this function to retrieve details of customers based on customer ID values retrieved from tables in the database.

Retrieve the sales order ID, customer ID, first name, last name, and total due for all sales orders from the SalesLT.SalesOrderHeader table and the dbo.ufnGetCustomerInformation function.

Table 1.87: Displaying records 1 - 10

SalesOrderID	CustomerID	FirstName	LastName	TotalDue
71774	29847	David	Hodgson	972.7850
71776	30072	Andrea	Thomsen	87.0851
71780	30113	Raja	Venugopal	42452.6519
71782	29485	Catherine	Abel	43962.7901
71783	29957	Kevin	Liu	92663.5609
71784	29736	Terry	Eminhizer	119960.8240
71796	29660	Anthony	Chor	63686.2708
71797	29796	Jon	Grande	86222.8072
71815	30089	Michael John	Troyer	1261.4440
71816	30027	Joseph	Mitzner	3754.9733

Table 1.88: Displaying records 1 - 10

CustomerID	FirstName	LastName	AddressLine1	City
29485	Catherine	Abel	57251 Serene Blvd	Van Nuys
29486	Kim	Abercrombie	Tanger Factory	Branch
29489	Frances	Adams	6900 Sisk Road	Modesto
29490	Margaret	Smith	Lewiston Mall	Lewiston
29492	Jay	Adams	Blue Ridge Mall	Kansas City
29494	Samuel	Agcaoili	No. 25800-130 King Street West	Toronto
29496	Robert	Ahlering	6500 East Grant Road	Tucson
29497	François	Ferrier	Eastridge Mall	Casper
29499	Amy	Alberts	252851 Rowan Place	Richmond
29502	Paul	Alcorn	White Mountain Mall	Rock Springs

```
SELECT SOH.SalesOrderID, SOH.CustomerID, CI.FirstName, CI.LastName, SOH.TotalDue
FROM SalesLT.SalesOrderHeader AS SOH
CROSS APPLY dbo.ufnGetCustomerInformation(SOH.CustomerID) AS CI
ORDER BY SOH.SalesOrderID;
```

Use the table-valued user-defined function dbo.ufnGetCustomerInformation again to retrieve details of customers based on customer ID values retrieved from tables in the database. Retrieve the customer ID, first name, last name, address line 1 and city for all customers from the SalesLT.Address and SalesLT.CustomerAddress tables, using the dbo.ufnGetCustomerInformation function.

```
SELECT CA.CustomerID, CI.FirstName, CI.LastName, A.AddressLine1, A.City
FROM SalesLT.Address AS A
JOIN SalesLT.CustomerAddress AS CA
ON A.AddressID = CA.AddressID
CROSS APPLY dbo.ufnGetCustomerInformation(CA.CustomerID) AS CI
ORDER BY CA.CustomerID;
```

1.7 Using Table Expressions

We can create tables in a number of ways that can then be re-usd, rather than being a one-off query.

Table 1.89: Displaying records 1 - 10

CustomerID	City
29698	Burnaby
29997	Seattle
29854	Joliet
30027	Oxnard
30023	Peoria
29637	Irving
29545	Bothell
29890	Port Orchard
29772	Austin
29883	Denver

1.7.1 Views

One way we can do this is using a view. Rather than repeatadly joining and selected fields from two tables, we can create a view that contains the join syntax then just select the view. This view can be used and queried as if it were a table. The data is still in the underlying tables, but it is a view or presentation on top of the tables. It's like a named query, which can make this simpler, plus we can add data at the view level, so they can view rights for the view table, but not the underlying data. Note that whilst you are able to use insert to add new rows, you can only do this to one of the underlying tables.

First we create a view

```
-- Create a view
CREATE VIEW SalesLT.vCustomerAddress
AS
SELECT C.CustomerID, FirstName, LastName, AddressLine1, City, StateProvince
FROM
SalesLT.Customer C JOIN SalesLT.CustomerAddress CA
ON C.CustomerID=CA.CustomerID
JOIN SalesLT.Address A
ON CA.AddressID=A.AddressID
```

Then we can query the view.

```
SELECT CustomerID, City
FROM SalesLT.vCustomerAddress
```

And we can join data to that view.

```
SELECT c.StateProvince, c.City, ISNULL(SUM(s.TotalDue), 0.00) AS Revenue
FROM SalesLT.vCustomerAddress AS c
LEFT JOIN SalesLT.SalesOrderHeader AS s
ON s.CustomerID = c.CustomerID
GROUP BY c.StateProvince, c.City
ORDER BY c.StateProvince, Revenue DESC;
```

1.7.2 Using Temporary Tables and Table Variables

Views are persistent database objects, however we might want a temporary table. We prefix the object with a # symbol and they are created in a seperate temporary database called tempdb, rather than the db you

Table 1.90: Displaying records 1 - 10

StateProvince	City	Revenue
Alberta	Calgary	0
Alberta	Edmonton	0
Arizona	Chandler	0
Arizona	Gilbert	0
Arizona	Mesa	0
Arizona	Phoenix	0
Arizona	Scottsdale	0
Arizona	Surprise	0
Arizona	Tucson	0
British Columbia	Vancouver	0

are working in. Typically using a single # will mean the table exists for the current user session. If you wanted it to persistent over multiple sessions, use the ## prefix.

An alternative approach is to use table variables, which were introduced to cause performance problems if used again, as the table has to be re-created each user session, which can become a problem if there are many tables. If we want to use one we prefix it with an @ sign to signify it is a variable e.g. **?** table so we define the variable as a table. This is connected to the batch rather than the session, so as long as we are in the block of code. It only works well on small databases or queries. They are for temporary situations or a temporary holding space.

In the next block of code we show how we create a temporary table then update and query it.

```
CREATE TABLE #Colors
(Color varchar(15));

INSERT INTO #Colors
SELECT DISTINCT Color FROM SalesLT.Product;

SELECT * FROM #Colors;
```

A table variable is similar, with some syntax differences, but this works on the code set (batch of commans ran at the same time) rather than being for the entire session.

```
DECLARE @Colors AS TABLE (Color varchar(15));

INSERT INTO @Colors
SELECT DISTINCT Color FROM SalesLT.Product;

SELECT * FROM @Colors;
```

** Key Points **

- Temporary tables are prefixed with a # symbol (You can also create global temporary tables that can be accessed by other processes by prefixing the name with ##)
- Local temporary tables are automatically deleted when the session in which they were created ends. Global temporary tables are deleted when the last user sessions referencing them is closed.
- Table variables are prefixed with a @ symbol.
- Table variables are scoped to the batch in which they are created.

Table 1.91: 2 records

CustomerID	FirstName	LastName	AddressLine1	City	StateProvince
29559	Robert	Bernacchi	2681 Eagle Peak	Bellevue	Washington
29559	Robert	Bernacchi	25915 140th Ave Ne	Bellevue	Washington

1.7.3 Table Value Functions (TVF)

This is a specific type of function which returns a table. It is a permanently define database object. Unlike views, we can pass values in to a TVF.

So first we create a TVF.

```
CREATE FUNCTION SalesLT.udfCustomersByCity
(@City AS VARCHAR(20))
RETURNS TABLE
AS
RETURN
    (SELECT C.CustomerID, FirstName, LastName, AddressLine1, City, StateProvince
     FROM SalesLT.Customer C JOIN SalesLT.CustomerAddress CA
     ON C.CustomerID=CA.CustomerID
     JOIN SalesLT.Address A ON CA.AddressID=A.AddressID
     WHERE City=@City);
```

Then we can call the TVF passing a parameter - in this instance we prvoide a city name of Bellevue - which will then return the results of the SELECT statement which is a list of customers in that city.

```
SELECT * FROM SalesLT.udfCustomersByCity('Bellevue')
```

1.7.4 Derived Tables

These are derived tables that return a multi column table. They are a virtual table to simplify a query. The table only exists within that SELECT query. It is a programming construct within a SELECT statement, it is used for programming purposes. They can use internal (aka inline) within the derived table element, or external aliases for columns - outside the () of the derived table.

In the example derived table below, the alias ProdCats is provided external to the query (outside the parentheses). Other elements such as Category as defined as internal aliases.

```
SELECT Category, COUNT(ProductID) AS Products
FROM
    (SELECT p.ProductID, p.Name AS Product, c.Name AS Category
     FROM SalesLT.Product AS p
     JOIN SalesLT.ProductCategory AS c
     ON p.ProductCategoryID = c.ProductCategoryID) AS ProdCats
GROUP BY Category
ORDER BY Category;
```

1.7.5 Using Common Table Expressions (CTEs)

Similar to a TVF before we are defining a temporary table object which is used within the scope of the query. But this time aorund we define the CTE first, then make calls to that CTE afterwards. Unlike a derived query however, you can refer to the CTE mutiple times within the same query/code batch, but it won't live

Table 1.92: Displaying records 1 - 10

Category	Products
Bib-Shorts	3
Bike Racks	1
Bike Stands	1
Bottles and Cages	3
Bottom Brackets	3
Brakes	2
Caps	1
Chains	1
Cleaners	1
Cranksets	3

Table 1.93: Displaying records 1 - 10

Category	Products
Bib-Shorts	3
Bike Racks	1
Bike Stands	1
Bottles and Cages	3
Bottom Brackets	3
Brakes	2
Caps	1
Chains	1
Cleaners	1
Cranksets	3

on like a view. We can also use a CTE for recursive elements like loops, so you might want to get everyone within a management tier, then say you want to go through that process three times for the top 3 levels, using the OPTION(MAXRECURSION 3).

So in the following example, we first define the CTE, then we make reference to it. The result here is the same as our derived query previously, but perhaps a little easiter to understand.

```
WITH ProductsByCategory (ProductID, ProductName, Category)
AS
(
    SELECT p.ProductID, p.Name, c.Name AS Category
     FROM SalesLT.Product AS p
     JOIN SalesLT.ProductCategory AS c
     ON p.ProductCategoryID = c.ProductCategoryID
)

SELECT Category, COUNT(ProductID) AS Products
FROM ProductsByCategory
GROUP BY Category
ORDER BY Category;
```

Next we will look at the our organisation chart example as previously mentioned. First let's look at our employee table.

Table 1.94: 9 records

EmployeeID	EmployeeName	ManagerID
1	adventure-works\david8	8
2	adventure-works\garrett1	1
3	adventure-works\jae0	NA
4	adventure-works\jillian0	3
5	adventure-works\josé1	1
6	adventure-works\linda3	3
7	adventure-works\michael9	9
8	adventure-works\pamela0	3
9	adventure-works\shu0	3

Table 1.95: 9 records

ManagerID	EmployeeID	EmployeeName	Level
NA	3	adventure-works\jae0	0
3	4	adventure-works\jillian0	1
3	6	adventure-works\linda3	1
3	8	adventure-works\pamela0	1
3	9	adventure-works\shu0	1
9	7	adventure-works\michael9	2
8	1	adventure-works\david8	2
1	2	adventure-works\garrett1	3
1	5	adventure-works\josé1	3

```
SELECT * FROM SalesLT.Employee
```

We can see that each employee has a manager ID apart from Employee 3 who is the CEO/MD. So we have a hierarchy of employees. We will go 3 levels deep this time (three recursions).

```
WITH OrgReport (ManagerID, EmployeeID, EmployeeName, Level)
AS
(
    -- Anchor query
    SELECT e.ManagerID, e.EmployeeID, EmployeeName, 0
    FROM SalesLT.Employee AS e
    WHERE ManagerID IS NULL

    UNION ALL

    -- Recursive query
    SELECT e.ManagerID, e.EmployeeID, e.EmployeeName, Level + 1
    FROM SalesLT.Employee AS e
    INNER JOIN OrgReport AS o ON e.ManagerID = o.EmployeeID
)

SELECT * FROM OrgReport
OPTION (MAXRECURSION 3);
```

** Key Points **

- A derived table is a subquery that generates a multicolumn rowset. You must use the AS clause to

Table 1.96: Displaying records 1 - 10

ProductID	ProductName	ProductModel	Summary
749	Road-150 Red, 62	Road-150	This bike is ridden by race winners. Developed with the Adventure Wo
750	Road-150 Red, 44	Road-150	This bike is ridden by race winners. Developed with the Adventure Wo
751	Road-150 Red, 48	Road-150	This bike is ridden by race winners. Developed with the Adventure Wo
752	Road-150 Red, 52	Road-150	This bike is ridden by race winners. Developed with the Adventure Wo
753	Road-150 Red, 56	Road-150	This bike is ridden by race winners. Developed with the Adventure Wo
754	Road-450 Red, 58	Road-450	A true multi-sport bike that offers streamlined riding and a revolutiona
755	Road-450 Red, 60	Road-450	A true multi-sport bike that offers streamlined riding and a revolutiona
756	Road-450 Red, 44	Road-450	A true multi-sport bike that offers streamlined riding and a revolutiona
757	Road-450 Red, 48	Road-450	A true multi-sport bike that offers streamlined riding and a revolutiona
758	Road-450 Red, 52	Road-450	A true multi-sport bike that offers streamlined riding and a revolutiona

define an alias for a derived query.

- Common Table Expressions (CTEs) provide a more intuitive syntax or defining rowsets than derived tables, and can be used mulitple times in the same query.
- You can use CTEs to define recursive queries.

1.7.6 Lab Exercises

AdventureWorks sells many products that are variants of the same product model. You must write queries that retrieve information about these products. Retrieve the product ID, product name, product model name, and product model summary for each product from the SalesLT.Product table and the SalesLT.vProductModelCatalogDescription view.

```
SELECT P.ProductID, P.Name AS ProductName, PM.Name AS ProductModel, PM.Summary
FROM SalesLT.Product AS P
JOIN SalesLT.vProductModelCatalogDescription AS PM
ON P.ProductModelID = PM.ProductModelID
ORDER BY ProductID;
```

You are only interested in products which have a listed color in the database. Create a table variable and populate it with a list of distinct colors from the SalesLT.Product table. Then use the table variable to filter a query that returns the product ID, name, and color from the SalesLT.Product table so that only products with a color listed in the table variable are returned.

```
DECLARE @Colors AS TABLE (Color NVARCHAR(15));

INSERT INTO @Colors
SELECT DISTINCT Color FROM SalesLT.Product;

SELECT ProductID, Name, Color
FROM SalesLT.Product
WHERE Color IN (SELECT Color FROM @Colors);
```

The AdventureWorksLT database includes a table-valued function named dbo.ufnGetAllCategories, which returns a table of product categories (e.g. 'Road Bikes') and parent categories (for example 'Bikes'). Write a query that uses this function to return a list of all products including their parent category and their own category.

Table 1.97: Displaying records 1 - 10

ParentCategory	Category	ProductID	ProductName
Accessories	Bike Racks	876	Hitch Rack - 4-Bike
Accessories	Bike Stands	879	All-Purpose Bike Stand
Accessories	Bottles and Cages	871	Mountain Bottle Cage
Accessories	Bottles and Cages	872	Road Bottle Cage
Accessories	Bottles and Cages	870	Water Bottle - 30 oz.
Accessories	Cleaners	877	Bike Wash - Dissolver
Accessories	Fenders	878	Fender Set - Mountain
Accessories	Helmets	708	Sport-100 Helmet, Black
Accessories	Helmets	711	Sport-100 Helmet, Blue
Accessories	Helmets	707	Sport-100 Helmet, Red

Table 1.98: Displaying records 1 - 10

CompanyContact	Revenue
Action Bicycle Specialists (Terry Eminhizer)	119960.8240
Aerobic Exercise Company (Rosmarie Carroll)	2361.6403
Bulk Discount Store (Christopher Beck)	98138.2131
Central Bicycle Specialists (Janeth Esteves)	43.0437
Channel Outlet (Richard Byham)	608.1766
Closest Bicycle Store (Pamala Kotc)	39531.6085
Coalition Bike Company (Donald Blanton)	2669.3183
Discount Tours (Melissa Marple)	3293.7761
Eastside Department Store (Kevin Liu)	92663.5609
Engineered Bike Systems (Joseph Mitzner)	3754.9733

```
SELECT C.ParentProductCategoryName AS ParentCategory,
       C.ProductCategoryName AS Category,
       P.ProductID, P.Name AS ProductName
FROM SalesLT.Product AS P
JOIN dbo.ufnGetAllCategories() AS C
ON P.ProductCategoryID = C.ProductCategoryID
ORDER BY ParentCategory, Category, ProductName;
```

Each AdventureWorks customer is a retail company with a named contact. You must create queries that return the total revenue for each customer, including the company and customer contact names. Retrieve a list of customers in the format Company (Contact Name) together with the total revenue for each customer. Use a derived table or a common table expression to retrieve the details for each sales order, and then query the derived table or CTE to aggregate and group the data.

```
SELECT CompanyContact, SUM(SalesAmount) AS Revenue
FROM
    (SELECT CONCAT(c.CompanyName, CONCAT(' (' + c.FirstName + ' ', c.LastName + ')')), SOH.TotalDue
     FROM SalesLT.SalesOrderHeader AS SOH
     JOIN SalesLT.Customer AS c
     ON SOH.CustomerID = c.CustomerID) AS CustomerSales(CompanyContact, SalesAmount)
GROUP BY CompanyContact
ORDER BY CompanyContact;
```

Table 1.99: Displaying records 1 - 10

ParentProductCategoryName	ProductCategoryName	Products
Accessories	Bike Racks	1
Accessories	Bike Stands	1
Accessories	Bottles and Cages	3
Accessories	Cleaners	1
Accessories	Fenders	1
Accessories	Helmets	3
Accessories	Hydration Packs	1
Accessories	Lights	4
Accessories	Locks	1
Accessories	Panniers	1

1.8 Grouping Sets and Pivoting Data

These are both ways of summarising data - grouping data includes cubing the data.

Grouping sets allows you to define multiple groupings within the same query. So you may haved a number of subtotals for different product types, then a total of all the subtotals combined.

So you would have something like

SELECT FROM

GROUP BY GROUPING SETS (

,

() – empty parentheses if aggregating all rows)

A result line will appear with NULLs in all column(s) which is the grand total.

There are other ways to aggregate the data such as ROLLUP and CUBE. ROLLUP is useful if you haave hierarchy - people in households, towns in regions. Cube will show every possible aggregation, rather than specifying each aggregation.

Both are specified in the GROUP BY X () where x is either ROLLUP or CUBE. You can add in GROUPING_ID

AS fiels to make the interpretation easier when using multiple grouping columns. You can then see the grand total more easily, since it will have a 1 in each groupid column.

So a usual group by might look something like this - with a high level Parent Category followed by a Product Category and a count of items.

```
SELECT cat.ParentProductCategoryName, cat.ProductCategoryName, count(prd.ProductID) AS Products
FROM SalesLT.vGetAllCategories as cat
LEFT JOIN SalesLT.Product AS prd
ON prd.ProductCategoryID = cat.ProductcategoryID
GROUP BY cat.ParentProductCategoryName, cat.ProductCategoryName
ORDER BY cat.ParentProductCategoryName, cat.ProductCategoryName;
```

Here there are no grand total by parent categories. Also, there might be certain items appear in more than one category without totals e.g. bike racks might be in mutiple parent categories.

Alternatively, we might create GROUPING SETS which also allows us to create totals.

Table 1.100: Displaying records 1 - 10

ParentProductCategoryName	ProductCategoryName	Products
NA	NA	296
NA	Bib-Shorts	3
NA	Bike Racks	1
NA	Bike Stands	1
NA	Bottles and Cages	3
NA	Bottom Brackets	3
NA	Brakes	2
NA	Caps	1
NA	Chains	1
NA	Cleaners	1

Table 1.101: Displaying records 1 - 10

ParentProductCategoryName	ProductCategoryName	Products
NA	NA	296
Accessories	NA	30
Accessories	Bike Racks	1
Accessories	Bike Stands	1
Accessories	Bottles and Cages	3
Accessories	Cleaners	1
Accessories	Fenders	1
Accessories	Helmets	3
Accessories	Hydration Packs	1
Accessories	Lights	4

```
SELECT cat.ParentProductCategoryName, cat.ProductCategoryName, count(prd.ProductID) AS Products
FROM SalesLT.vGetAllCategories as cat
LEFT JOIN SalesLT.Product AS prd
ON prd.ProductCategoryID = cat.ProductcategoryID
GROUP BY GROUPING SETS(cat.ParentProductCategoryName, cat.ProductCategoryName, ())
ORDER BY cat.ParentProductCategoryName, cat.ProductCategoryName;
```

We now start getting totals, we can see there is a total of 295 products, followed by 3 bib-shorts (irrespective of parent), and later in the table we get the totals by parent categories. But we have no loss some of the relationships with parent categories with this query. It is possible to write a query where these things might appear, GROUPING SETS is the most flexible of the grouping commands, but it might be easier to use one of the other grouping options as required, next shows the the ROLLUP command.

```
SELECT cat.ParentProductCategoryName, cat.ProductCategoryName, count(prd.ProductID) AS Products
FROM SalesLT.vGetAllCategories as cat
LEFT JOIN SalesLT.Product AS prd
ON prd.ProductCategoryID = cat.ProductcategoryID
GROUP BY ROLLUP (cat.ParentProductCategoryName, cat.ProductCategoryName)
ORDER BY cat.ParentProductCategoryName, cat.ProductCategoryName;
```

It is now giving us totals for parents and products. However, we are now missing situations where items might appear in more than one parent category e.g. bottles and cages that are in a category other than accessories. So we can use CUBE to get these other aggregation options, which will give us all possible aggregations.

Table 1.102: Displaying records 1 - 10

ParentProductCategoryName	ProductCategoryName	Products
NA	NA	296
NA	Bib-Shorts	3
NA	Bike Racks	1
NA	Bike Stands	1
NA	Bottles and Cages	3
NA	Bottom Brackets	3
NA	Brakes	2
NA	Caps	1
NA	Chains	1
NA	Cleaners	1

```
SELECT cat.ParentProductCategoryName, cat.ProductCategoryName, count(prd.ProductID) AS Products
FROM SalesLT.vGetAllCategories as cat
LEFT JOIN SalesLT.Product AS prd
ON prd.ProductCategoryID = cat.ProductcategoryID
GROUP BY CUBE (cat.ParentProductCategoryName, cat.ProductCategoryName)
ORDER BY cat.ParentProductCategoryName, cat.ProductCategoryName;
```

** Key Points ** * Use GROUPING SETS to define custom groupings.
* Use ROLLUP to include subtotals and a grand total for hierarchical groupings.
* Use CUBE to include all possible groupings.

1.8.1 Pivoting Data

Pivoting is another form of summarising data. We take row based items and pivot them in to single, summarised, columns in a new table. We simply use the PIVOT command. It is possible to undue this using UNPIVOT however this may loose some of the detail that existed in the original data. For instance line level data, such as which particular clothing items were purchased and at which value, is now lost as we just get a clothing total, but re-orientated to match the original data format.

We might want to be able to see the totals by colour in our data and by cateory. You have to know which columns you wish to create using the PIVOT table, for instance in the code below we are listing the actual colours our in the PIVOT command. If one colour was missed, the query would work, but you obviously wouldn't get that colour returned or inlcuded in totals.

```
SELECT * FROM
    (SELECT P.ProductID, PC.Name,ISNULL(P.Color, 'Uncolored') AS Color
     FROM saleslt.productcategory AS PC
     JOIN SalesLT.Product AS P
     ON PC.ProductCategoryID=P.ProductCategoryID
     ) AS PPC
PIVOT(COUNT(ProductID) FOR Color IN([Red],[Blue],[Black],[Silver],[Yellow],[Grey], [Multi], [Uncolored])
ORDER BY Name;
```

If we wanted to, we could copy/archive this in to a new table, which can be re-used again or compared and retrieved in the future.

```
CREATE TABLE #ProductColorPivot
(Name varchar(50), Red int, Blue int, Black int, Silver int, Yellow int, Grey int , multi int, uncolored
```

Table 1.103: Displaying records 1 - 10

Name	Red	Blue	Black	Silver	Yellow	Grey	Multi	Uncolored
Bib-Shorts	0	0	0	0	0	0	3	0
Bike Racks	0	0	0	0	0	0	0	1
Bike Stands	0	0	0	0	0	0	0	1
Bottles and Cages	0	0	0	0	0	0	0	3
Bottom Brackets	0	0	0	0	0	0	0	3
Brakes	0	0	0	2	0	0	0	0
Caps	0	0	0	0	0	0	1	0
Chains	0	0	0	1	0	0	0	0
Cleaners	0	0	0	0	0	0	0	1
Cranksets	0	0	3	0	0	0	0	0

```
INSERT INTO #ProductColorPivot
    SELECT * FROM
    (SELECT P.ProductID, PC.Name,ISNULL(P.Color, 'Uncolored') AS Color
     FROM saleslt.productcategory AS PC
     JOIN SalesLT.Product AS P
     ON PC.ProductCategoryID=P.ProductCategoryID
     ) AS PPC
PIVOT(COUNT(ProductID) FOR Color IN([Red],[Blue],[Black],[Silver],[Yellow],[Grey], [Multi], [Uncolored])
ORDER BY Name;
```

If we then wanted to unpvivot the data we would do so as follows.

```
SELECT Name, Color, ProductCount
FROM
    (SELECT Name,
    [Red],[Blue],[Black],[Silver],[Yellow],[Grey], [Multi], [Uncolored]
    FROM #ProductColorPivot) pcp
UNPIVOT
(ProductCount FOR Color IN ([Red],[Blue],[Black],[Silver],[Yellow],[Grey], [Multi], [Uncolored])
) AS ProductCounts
```

** Key Points ** * Use PIVOT to re-orient a rowset by generating mulitple columns from values in a single column.
* Use UNPIVOT to re-orient mulitple columns in a an existing rowset into a single column.

1.8.2 Lab Exercises

AdventureWorks sells products to customers in multiple country/regions around the world.

An existing report uses the query provided in the editor to return total sales revenue grouped by country/region and state/province. Modify the query so that the results include a grand total for all sales revenue and a subtotal for each country/region in addition to the state/province subtotals that are already returned.

```
SELECT a.CountryRegion, a.StateProvince, SUM(soh.TotalDue) AS Revenue
FROM SalesLT.Address AS a
JOIN SalesLT.CustomerAddress AS ca
ON a.AddressID = ca.AddressID
JOIN SalesLT.Customer AS c
```

Table 1.104: 9 records

CountryRegion	StateProvince	Revenue
NA	NA	956303.5949
United Kingdom	NA	572496.5594
United Kingdom	England	572496.5594
United States	NA	383807.0355
United States	California	346517.6072
United States	Colorado	14017.9083
United States	Nevada	7330.8972
United States	New Mexico	15275.1977
United States	Utah	665.4251

Table 1.105: 9 records

CountryRegion	StateProvince	Level	Revenue
NA	NA	Total	956303.5949
United Kingdom	NA	United Kingdom Subtotal	572496.5594
United Kingdom	England	England Subtotal	572496.5594
United States	NA	United States Subtotal	383807.0355
United States	California	California Subtotal	346517.6072
United States	Colorado	Colorado Subtotal	14017.9083
United States	Nevada	Nevada Subtotal	7330.8972
United States	New Mexico	New Mexico Subtotal	15275.1977
United States	Utah	Utah Subtotal	665.4251

```
ON ca.CustomerID = c.CustomerID
JOIN SalesLT.SalesOrderHeader as soh
ON c.CustomerID = soh.CustomerID
GROUP BY ROLLUP (a.CountryRegion, a.StateProvince)
ORDER BY a.CountryRegion, a.StateProvince;
```

Modify your query to include a column named Level that indicates at which level in the total, country/region, and state/province hierarchy the revenue figure in the row is aggregated.

```
SELECT a.CountryRegion, a.StateProvince,
    IIF(GROUPING_ID(a.CountryRegion) = 1 AND GROUPING_ID(a.StateProvince) = 1, 'Total', IIF(GROUPING_ID
SUM(soh.TotalDue) AS Revenue
FROM SalesLT.Address AS a
JOIN SalesLT.CustomerAddress AS ca
ON a.AddressID = ca.AddressID
JOIN SalesLT.Customer AS c
ON ca.CustomerID = c.CustomerID
JOIN SalesLT.SalesOrderHeader as soh
ON c.CustomerID = soh.CustomerID
GROUP BY ROLLUP(a.CountryRegion, a.StateProvince)
ORDER BY a.CountryRegion, a.StateProvince;
```

Or to extend your query to include a grouping for individual cities.

```
SELECT a.CountryRegion, a.StateProvince, a.City,
CHOOSE (1 + GROUPING_ID(a.CountryRegion) + GROUPING_ID(a.StateProvince) + GROUPING_ID(a.City),
        a.City + ' Subtotal', a.StateProvince + ' Subtotal',
```

Table 1.106: Displaying records 1 - 10

CountryRegion	StateProvince	City	Level	Revenue
NA	NA	NA	Total	956303.5949
United Kingdom	NA	NA	United Kingdom Subtotal	572496.5594
United Kingdom	England	NA	England Subtotal	572496.5594
United Kingdom	England	Abingdon	Abingdon Subtotal	45.1995
United Kingdom	England	Cambridge	Cambridge Subtotal	2711.4098
United Kingdom	England	Gloucestershire	Gloucestershire Subtotal	70698.9922
United Kingdom	England	High Wycombe	High Wycombe Subtotal	608.1766
United Kingdom	England	Liverpool	Liverpool Subtotal	86222.8072
United Kingdom	England	London	London Subtotal	206736.1667
United Kingdom	England	Maidenhead	Maidenhead Subtotal	43.0437

```
        a.CountryRegion + ' Subtotal', 'Total') AS Level,
SUM(soh.TotalDue) AS Revenue
FROM SalesLT.Address AS a
JOIN SalesLT.CustomerAddress AS ca
ON a.AddressID = ca.AddressID
JOIN SalesLT.Customer AS c
ON ca.CustomerID = c.CustomerID
JOIN SalesLT.SalesOrderHeader as soh
ON c.CustomerID = soh.CustomerID
GROUP BY ROLLUP(a.CountryRegion, a.StateProvince, a.City)
ORDER BY a.CountryRegion, a.StateProvince, a.City;
```

AdventureWorks products are grouped into categories, which in turn have parent categories (defined in the SalesLT.vGetAllCategories view).

AdventureWorks customers are retail companies, and they may place orders for products of any category. The revenue for each product in an order is recorded as the LineTotal value in the SalesLT.SalesOrderDetail table.

Retrieve a list of customer company names together with their total revenue for each parent category in Accessories, Bikes, Clothing, and Components. Make sure to use the aliases provided, and default column names elsewhere.

```
SELECT * FROM
(SELECT cat.ParentProductCategoryName, cust.CompanyName, sod.LineTotal
 FROM SalesLT.SalesOrderDetail AS sod
 JOIN SalesLT.SalesOrderHeader AS soh ON sod.SalesOrderID = soh.SalesOrderID
 JOIN SalesLT.Customer AS cust ON soh.CustomerID = cust.CustomerID
 JOIN SalesLT.Product AS prod ON sod.ProductID = prod.ProductID
 JOIN SalesLT.vGetAllCategories AS cat ON prod.ProductcategoryID = cat.ProductCategoryID) AS catsales
PIVOT (SUM(LineTotal) FOR ParentProductCategoryName
IN ([Accessories], [Bikes], [Clothing], [Components])) AS pivotedsales
ORDER BY CompanyName;
```

1.9 Modifying Data

This section isn't about querying data, but adding, deleting and updating data in our database.

Table 1.107: Displaying records 1 - 10

CompanyName	Accessories	Bikes	Clothing	Components
Action Bicycle Specialists	1299.885	76613.6518	2461.6573	9494.082
Aerobic Exercise Company	NA	NA	NA	1732.890
Bulk Discount Store	730.464	70597.2840	851.5620	1980.918
Central Bicycle Specialists	NA	NA	NA	31.584
Channel Outlet	216.000	NA	308.6640	NA
Closest Bicycle Store	NA	20389.6680	559.1641	8001.846
Coalition Bike Company	NA	529.4928	124.7760	1201.938
Discount Tours	72.000	2041.1880	341.0580	72.882
Eastside Department Store	1220.236	51096.0548	2772.2296	10594.848
Engineered Bike Systems	NA	2604.7620	178.7460	63.900

To add data we use the INSERT … VALUES command, where you can leave blank columns that allow NULLs and cols with default constraints. You can also leave identify/primary key fields blank since they will be generated automatically, depending on what seed and increment values were specified when the table was set up, however it is possible to overide these values if you wish. You can add a NULL value to a column if you wish.

You can also insert the results of another query in to an existing table using INSERT … SELECT or INSERT … EXEC where EXEC is the execution of a stored query.

Another option is SELECT … INTO will explicity create a new table, inserting data based on query. However this newly created table will not have contraints, defaults, indexes or primary keys, just a set of columns with the results of the query. This is currently not supported in Azure, however you could create the table first with the items needed such as columns and pk, then use the INSERT INTO for Azure.

Our Identity property of a column generates sequential numbers for insertations of new data into a database and we can set the parameters of this identity column. If you insert a new item (row) in to a table, you might want to then get back this identity column for this new item e.g. you automatically generate a new sales ID but a new order, however you might want to then present this back to the customer, this is acheived using **@?**, but this returns the last identity referenced, so if you are adding the sales order first then the individual items in a seperate table, it will retrieve the idenity column from the sales order details rather than sales order header. This command therefore returns the last identity from the current connected session.

An alternative is to use SELECT SCOPE_IDENTITY() AS ORDERID which wil bring back the last identity column for the specified table in the current session, but again this can be problematic. The alternative is to use IDENT_CURRENT(') which passes the last identity inserted into a particular table, however this is across all sessions, so if others are inserting the table this can be problematic. So generally the best option is to use SCOPE_IDENTITY and present that, before then going on to do other things.

If you perhaps wanted to have two different tables for two different order types - say in store purchases and online purchases as two dinstinct tables - but to both use the same sequential list for order ID, you could do this using sequences. They exist independently of tables so can be referenced by mutiple tables, the query will ask the server what the next id is in the sequence and be provided with that. It is like a central store that issues new numbers on request.

So to create a new table for customer call logs we do the following:

```
CREATE TABLE SalesLT.CallLog
(
    CallID int IDENTITY PRIMARY KEY NOT NULL, -- no seed or increment set, so will start at 1 then add
    CallTime datetime NOT NULL DEFAULT GETDATE(), -- if not date and time set, it will get the current
    SalesPerson nvarchar(256) NOT NULL,
    CustomerID int NOT NULL REFERENCES SalesLT.Customer(CustomerID), -- references a foreign key from t
```

```
    PhoneNumber nvarchar(25) NOT NULL,
    Notes nvarchar(max) NULL -- the only column which is allowed to be NULL, since perhaps the customer
);
GO
```

If we then wanted to explicitly add or insert a row in to the datbase we do the following - note the CallID does not need to be given since this is automatically created :

– INT is optional, we state values as we are explicitly stating the values

```
INSERT INTO SalesLT.CallLog
VALUES
('2015-01-01T12:30:00', 'adventure-works\pamela0', 1, '245-555-0173', 'Returning call re: enquiry about
```

If we wantedc to accept the default options, we can do that by using DEFAULT (e.g. for data/time) and if there are no notes we can use NULL.

```
INSERT INTO SalesLT.CallLog
VALUES
(DEFAULT, 'adventure-works\david8', 2, '170-555-0127', NULL);
```

Perhaps we need to re-order our fields rather than accepting the default sequence, perhaps it is a modified table or a table form another source. To do this, we explicitly state the columns.

```
INSERT INTO SalesLT.CallLog (SalesPerson, CustomerID, PhoneNumber)
VALUES
('adventure-works\jillian0', 3, '279-555-0130');
```

If we wanted to add two records, we can do this at the same time, and even use some different options for the different rows.

```
INSERT INTO SalesLT.CallLog
VALUES
(DATEADD(mi,-2, GETDATE()), 'adventure-works\jillian0', 4, '710-555-0173', NULL),
(DEFAULT, 'adventure-works\shu0', 5, '828-555-0186', 'Called to arrange deliver of order 10987');
```

There are more examples with adding query results into tables and getting identity values in the demo files.

** Key Points ** * Use the INSERT statement to insert one or more rows into a table.
* When inserting explicit values, you can omit identity columns, columns that allow NULLs, and columns on which a default constraint is defined.
* Identity columns generate a unique integer identifier for each row. You can also use a sequence to generate unique values that can be used in multiple tables.

1.9.1 Updating and Deleting Data

The UPDATE command will update all the rows in a table or view and can be filtered using WHERE and the columns can be selected using FROM. It is more likely we will want to select a particular set of rows, so we will be using the WHERE clause in most instances. We use the SET clause in the UPDATE statement to assign a value and we can use a calculation and we can specify multiple columns e.g. SET unitprice = (unitprice *1.04), notes = 'price increase by 4% due to inflation'. You can also update values in a table based on the results of a query. Only columns in the set clause are affected. There are a number of examples in the MSDN TSQL pages Transact-SQL Refrence.

A common challenge in data warehousing is what is sometimes referred to as 'upserting' - we are updating records that already exist AND we insert new records if they do not. To do this, we use the MERGE command, which modifies data based on a condition. We look at situations of when the source data matches the target data. So we might check if the ProductID already exists, and where it does we use an UPDATE.

But if there is a record in the source table but on the target table (db) we will INSERT a new product. The code below is an example

```
MERGE INTO Production.Products as P
    USING Production.ProductsStaging as S
    ON P.ProductID=S.ProductID
WHEN MATCHED THEN
    UPDATE SET
    P.UnitPrice = S.UnitPrice, P.Discontinued=S.Discontinued
WHEN NOT MATCHED THEN
    INSERT (ProductName, CategoryID, UnitPrice, Discontinued)
    VALUES (S.ProductName, S.CategoryID, S.UnitPrice, S.Discontinued);
```

We can delete data from a table, however we need to be careful, if we don't specify a WHERE clause we may delete a lot of data!

```
DELETE FROM Sales.OrderDetails
WHERE orderid = 10248;
```

There is an overhead with this approach, since SQL Server will record this transaction. If we are deleting a whole table and wish to deallocate the entire table space, we are based to use TRUNCATE. TRUNCATE TABLE Will fail if the table is referenced by a foreign key constraint in another table, the same will happen with the DELETE command. The table definition remains, but all the data is removed and the space reallocated. This is often used in staging tables, so we might load new data in to a staging table, then use the MERGE command for your target table, then empty the staging table but keep the strucutre, so we use TRUNCATE.

So if we wanted to add a value to where a value is currently NULL, for instance to state that no notes were recorded, we can do this as:

```
UPDATE SalesLT.CallLog
SET Notes = 'No notes'
WHERE Notes IS NULL;
```

Or we can update multiple columns, in the following example we set two empty strings using '' - note this does it for the ENTIRE table, as there is no WHERE clause.

```
UPDATE SalesLT.CallLog
SET SalesPerson = '', PhoneNumber = '';
```

Now we want to re-add the data just deleted, using a query, getting Sales Person and Phone Number from the customer table, where the CustomerID matches.

```
UPDATE SalesLT.CallLog
SET SalesPerson = c.SalesPerson, PhoneNumber = c.Phone
FROM SalesLT.Customer AS c
WHERE c.CustomerID = SalesLT.CallLog.CustomerID;
```

We might want to archive off then delete anything older than this week. This can be achived by using DELETE and WHERE.

```
DELETE FROM SalesLT.CallLog
WHERE CallTime < DATEADD(dd, -7, GETDATE());
```

Or if we want to complete wipe the call log table we use TRUNCATE.

```
TRUNCATE TABLE SalesLT.CallLog;
```

** Key Points ** * Use the UPDATE statement to modify the values of one or more columns in specified rows of a table.

* Use the DELETE statement to delete specified rows in a table.
* Use the MERGE statement to insert, update, and delete rows in a target table based on data in a source table.

1.9.2 Lab Exercises

AdventureWorks has started selling the new product - LED Lights. Insert it into the SalesLT.Product table, using default or NULL values for unspecified columns. Once you've inserted the product, run SELECT SCOPE_IDENTITY(); to get the last identity value that was inserted. Add a query to view the row for the product in the SalesLT.Product table.

```
INSERT INTO SalesLT.Product (Name, ProductNumber, StandardCost, ListPrice, ProductCategoryID, SellStartl
VALUES
('LED Lights', 'LT-L123', 2.56, 12.99, 37, GETDATE());

SELECT SCOPE_IDENTITY();

SELECT * FROM SalesLT.Product
WHERE ProductID = SCOPE_IDENTITY();
```

Insert the two new products with the appropriate ProductCategoryID value, based on the product details above.

Finish the query to join the SalesLT.Product and SalesLT.ProductCategory tables. That way, you can verify that the data has been inserted. Make sure to use the aliases provided, and default column names elsewhere.

```
INSERT INTO SalesLT.ProductCategory (ParentProductCategoryID, Name)
VALUES
(4, 'Bells and Horns');

INSERT INTO SalesLT.Product (Name, ProductNumber, StandardCost, ListPrice, ProductCategoryID, SellStartl
VALUES
('Bicycle Bell', 'BB-RING', 2.47, 4.99, IDENT_CURRENT('SalesLT.ProductCategory'), GETDATE()),
('Bicycle Horn', 'BB-PARP', 1.29, 3.75, IDENT_CURRENT('SalesLT.ProductCategory'), GETDATE());

SELECT c.Name As Category, p.Name AS Product
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory as c ON p.ProductCategoryID = c.ProductCategoryID
WHERE p.ProductCategoryID = IDENT_CURRENT('SalesLT.ProductCategory');
```

You must now update the records you have previously inserted to reflect the correct pricing. The sales manager at AdventureWorks has mandated a 10% price increase for all products in the Bells and Horns category. Update the rows in the SalesLT.Product table for these products to increase their price by 10%.

```
UPDATE SalesLT.Product
SET ListPrice = ListPrice * 1.1
WHERE ProductCategoryID =
  (SELECT ProductCategoryID FROM SalesLT.ProductCategory WHERE Name = 'Bells and Horns');
```

The new LED lights you inserted in the previous challenge are to replace all previous light products. Update the SalesLT.Product table to set the DiscontinuedDate to today's date for all products in the Lights category (ProductCategoryID 37) other than the LED Lights product you inserted previously. So we delete records from the SalesLT.Product table first then delete records from the SalesLT.ProductCategory table.

```
DELETE FROM SalesLT.Product
WHERE ProductCategoryID =
```

```
    (SELECT ProductCategoryID FROM SalesLT.ProductCategory WHERE Name = 'Bells and Horns');

DELETE FROM SalesLT.ProductCategory
WHERE ProductCategoryID =
    (SELECT ProductCategoryID FROM SalesLT.ProductCategory WHERE Name = 'Bells and Horns');
```

```
UPDATE SalesLT.Product
SET DiscontinuedDate = GETDATE()
WHERE ProductCategoryID = 37 AND ProductNumber <> 'LT-L123';
```

The Bells and Horns category has not been successful and it must be deleted from the database. Delete the records for the Bells and Horns category and its products. You must ensure that you delete the records from the tables in the correct order to avoid a foreign-key constraint violation.

1.10 Programming with Transact-SQL

We will start looking at procedural logic in this section.

1.10.1 Batches

Batches are sets of commands to to SQL Server as a unit. We can seperate out elements using the GO command, which will send that batch. GO is not a T-SQL command, it interpretor which uses the GO command to send that batch.

1.10.2 Comments

We can add a code block using /* Comment */ for multiple lines or a comment block, this can come in useful to explain what the code is for or why you are doing something a certain way.

1.10.3 Variables

Variables - usually single scalar values - can be used for many things. We delare the variable, give it a data type, then initialise it with a value if you like, or assign a value later. They are declaring using DECLARE **?** nvarchar(15)

```
DECLARE @City VARCHAR(20)='Toronto';
```

But rather than declaring a variables value, we can have a variable which receives something e.g. the results of a select query.

```
DECLARE @Result money
SELECT @Result=MAX(TotalDue)
FROM SalesLT.SalesOrderHeader

PRINT @Result
```

** Key Points ** * A batch defines a group of Transact-SQL command submitted by a client application for execution. Some commands can only be executed at the start of a new batch, and variable values cannot span batches.
* Use comments to document your Transact-SQL code. Inline comments are prefixed by –, and multi-line comment blocks are enclosed in /* and /.

Declare variables by using the DECLARE keyword, specifying a name (prefixed with @) and a data type. You can optionally assign an initial value.
* Assign values to variables by using the SET keyword or in a SELECT statement.

1.10.4 Conditional Branching

An IF statement is one such an example, do one thing or something else or indeed nothing. We can go further using IF and ELSE. We can also use this for providing messages, as shown below where we print a message if the row is <1 i.e. it is not found.

It is not neccessary to add a BEGIN adn END in all cases, but it can be easier to read and helps when possibly expanding the IF list later.

```
UPDATE SalesLT.Product
SET DiscontinuedDate=getdate()
WHERE ProductID=1;

IF @@ROWCOUNT<1
BEGIN
    PRINT 'Product was not found'
END
ELSE
BEGIN
    PRINT 'Product Updated'
END
```

** Key Points ** * Use the IF keyword to execute a task based on the results of a conditional test.
* Use an ELSE clause if you need to execute an alternative task if the conditional test returns false.
* Enclose mulitple statements in an IF or ELSE clause between BEGIN and END keywords.

1.10.5 Looping

Typically in SQL Server we don't want to loop through our data, we work on sets of data, as looping can be slow, so the ideal is to work with sets. A loop will end where a condition (predicate) evaluates to FALSE or unknown. Either we have a BREAK which stops everything or continue, which will move on to the next set of code. So in the example below 5 rows will be affected, as we set the counter as <= 5, adding a row. Looping is best used when you might need to check for something, where we don't know how many times we might need to do something, a situation which is not that common in SQL Server.

```
DECLARE @Counter int=1

WHILE @Counter <=5

BEGIN
    INSERT SalesLT.DemoTable(Description)
    VALUES ('ROW '+CONVERT(varchar(5),@Counter))
    SET @Counter=@Counter+1
END

SELECT Description FROM SalesLT.DemoTable
```

** Key Points **

- Use a WHILE loop if you need to repeat one or more statements until a specified condition is true.

- Use BREAK and CONTINUE to exit or restart the loop.
- Avoid using loops to iteratively update or retrieve single records - in most cases, you should use set-based operations to retrieve and modify data.

1.10.6 Stored Procedures

This is encapsulated a block of code in to a named block. They are setup to take parameters as inputs and can use conidtional branching (IF ELSE) and other such items. Sounds an awful lot like a function in R. In the example below, if we don't specify a value (the **?**) then we run a query which will select all the products (the IF element). If I do specify a category, then return the products in the category (ELSE).

```
CREATE PROCEDURE SalesLT.GetProductsByCategory (@CategoryID INT = NULL)
AS
IF @CategoryID IS NULL
    SELECT ProductID, Name, Color, Size, ListPrice
    FROM SalesLT.Product
ELSE
    SELECT ProductID, Name, Color, Size, ListPrice
    FROM SalesLT.Product
    WHERE ProductCategoryID = @CategoryID;
```

The first time around, the object - the stored proceedure - will be created. To use it, we must use an EXEC or EXECUTE (UTE is optional) statement.

```
EXEC SalesLT.GetProductsByCategory
```

If we want to pass a paramter e.g. category 6, we do a similar exercise.

```
EXEC SalesLT.GetProductsByCategory 6
```

** Key Points ** * Use stored procedures to encapsulate Transact-SQL code in a reusable database objects.
* You can define parameters for a stored procedure, and use them as variables in the Transact-SQL code it contains.
* Stored procedures can return rowsets (usually the results of a SELECT statement). They can also return output parameters, and they always return a return value, which is used to indicate status.

1.10.7 Lab Exercises

You want to create reusable scripts that make it easy to insert sales orders. You plan to create a script to insert the order header record, and a separate script to insert order detail records for a specified order header.

Both scripts will make use of variables to make them easy to reuse. Your script to insert an order header must enable users to specify values for the order date, due date, and customer ID.

```
DECLARE @OrderDate datetime = GETDATE();
DECLARE @DueDate datetime = DATEADD(dd, 7, GETDATE());
DECLARE @CustomerID int = 1;

INSERT INTO SalesLT.SalesOrderHeader (OrderDate, DueDate, CustomerID, ShipMethod)
VALUES (@OrderDate, @DueDate, @CustomerID, 'CARGO TRANSPORT 5');

PRINT SCOPE_IDENTITY();
```

As a next step, you want to insert an order detail. The script for this must enable users to specify a sales order ID, a product ID, a quantity, and a unit price.

The script should check if the specified sales order ID exists in the SalesLT.SalesOrderHeader table. This can be done with the EXISTS predicate.

If it does, the code should insert the order details into the SalesLT.SalesOrderDetail table (using default values or NULL for unspecified columns).

If the sales order ID does not exist in the SalesLT.SalesOrderHeader table, the code should print the message 'The order does not exist'.

```
-- Code from previous exercise
DECLARE @OrderDate datetime = GETDATE();
DECLARE @DueDate datetime = DATEADD(dd, 7, GETDATE());
DECLARE @CustomerID int = 1;
INSERT INTO SalesLT.SalesOrderHeader (OrderDate, DueDate, CustomerID, ShipMethod)
VALUES (@OrderDate, @DueDate, @CustomerID, 'CARGO TRANSPORT 5');
DECLARE @OrderID int = SCOPE_IDENTITY();

-- Additional script to complete
DECLARE @ProductID int = 760;
DECLARE @Quantity int = 1;
DECLARE @UnitPrice money = 782.99;

IF EXISTS (SELECT * FROM SalesLT.SalesOrderDetail WHERE SalesOrderID = @OrderID)
BEGIN
    INSERT INTO SalesLT.SalesOrderDetail (SalesOrderID, OrderQty, ProductID, UnitPrice)
    VALUES (@OrderID, @Quantity, @ProductID, @UnitPrice)
END
ELSE
BEGIN
    PRINT 'The order does not exist'
END
```

Adventure Works has determined that the market average price for a bike is $2,000, and consumer research has indicated that the maximum price any customer would be likely to pay for a bike is $5,000.

You must write some Transact-SQL logic that incrementally increases the list price for all bike products by 10% until the average list price for a bike is at least the same as the market average, or until the most expensive bike is priced above the acceptable maximum indicated by the consumer research.

The product categories in the Bikes parent category can be determined from the SalesLT.vGetAllCategories view.

** Key Points **

- The loop should execute only if the average list price of a product in the 'Bikes' parent category is less than the market average.
- Update all products that are in the 'Bikes' parent category, increasing the list price by 10%.
- Determine the new average and maximum selling price for products that are in the 'Bikes' parent category.
- If the new maximum price is greater than or equal to the maximum acceptable price, exit the loop; otherwise continue.

```
DECLARE @MarketAverage money = 2000;
DECLARE @MarketMax money = 5000;
DECLARE @AWMax money;
DECLARE @AWAverage money;

SELECT @AWAverage = AVG(ListPrice), @AWMax = MAX(ListPrice)
FROM SalesLT.Product
WHERE ProductCategoryID IN
    (SELECT DISTINCT ProductCategoryID
     FROM SalesLT.vGetAllCategories
     WHERE ParentProductCategoryName = 'Bikes');

WHILE @AWAverage < @MarketAverage
BEGIN
   UPDATE SalesLT.Product
   SET ListPrice = ListPrice * 1.1
   WHERE ProductCategoryID IN
    (SELECT DISTINCT ProductCategoryID
     FROM SalesLT.vGetAllCategories
     WHERE ParentProductCategoryName = 'Bikes');

    SELECT @AWAverage = AVG(ListPrice), @AWMax = MAX(ListPrice)
    FROM SalesLT.Product
    WHERE ProductCategoryID IN
    (SELECT DISTINCT ProductCategoryID
     FROM SalesLT.vGetAllCategories
     WHERE ParentProductCategoryName = 'Bikes');

   IF @AWMax >= @MarketMax
      END
   ELSE
      CONTINUE
END

PRINT 'New average bike price:' + CONVERT(VARCHAR, @AWAverage);
PRINT 'New maximum bike price:' + CONVERT(VARCHAR, @AWMax);
```

1.11 Error Handling and Transactions

You can raise errors yourself because of a condition, and not just some system error. For instance you can introduce business logic, so whilst a transaction in to the database might in principle be ok, you might not see that some business rules are valide i.e. too high of a discount price for instance. You can also pass error mesages back to client applications.

The errors have different levels of severity from 1 to 25 and includes the line number which caused the error. A line with proceedure will state which procedure or sub-proceedure caused the problem. If the error message is particularly high, SQL Server will also udnertake some certain actions to mitigte some of these problems like disconnect from the server if the disk is corrupt. A state is also given, which can be useful when seeking assistance for troubleshooting. Within the database error messages are stored in the sys.messages table in the database and you can add custom messages using sp_addmessage to this table (note this is for SQL Server and not Azure). It is better and more 'current' to throw custom error messages within your code, rather than in the db table, and this approach will work with SQL on Azure.

1.11.1 Raising or Throwing Errors

if you wish to raise an error you can do so using the RAISERROR command - this works in both SQL Server and Azure. You can specify the serverity and state when doing so.

Another way of raising errors which is newer syntax, it will in time replace RAISERROR, which is THROW. Such user defined errors can only be generated for lower levels of severity, so higher level problems don't cause SQL Server to take mitigating actions like disconnecting the server. The customer error message must be above 50000.

In the following example, the SQL query would run, but we introduce a rule based on business logic which will return an error if no rows are affected by the query. Nothing has failed, but nothing actually happended.

```
UPDATE SalesLT.Product
SET DiscontinuedDate = GETDATE()
WHERE ProductID = 0;

IF @@ROWCOUNT < 1
    RAISERROR('The product was not found - no products have been updated', 16, 0);
```

Or we can use the newer THROW command.

```
UPDATE SalesLT.Product
SET DiscontinuedDate = GETDATE()
WHERE ProductID = 0;

IF @@ROWCOUNT < 1
    THROW 50001, 'The product was not found - no products have been updated', 0;
```

** Key Points ** * System errors have pre-defined numbers, messages, severity levels, and other characteristics that you can use to troubleshoot issues.
* You can use RAISERROR and THROW to raise custom errors.

1.11.2 Catching and Handling Errors

This is sometimes called stuctured exception handling, where we have a TRY block of code which is something we want to run, then we have a CATCH block of code so that if the TRY code fails we catch and handle that error. We either deal with the rror and fix it or thrown an error to the compiler. IF the TRY block code works, no error will be thrown. There are a number of system global variables we can call for errors depending on what we are trying to catch. We can throw the original message, or we can throw a custom error.

In the first example below, we update the product table using the ProductNumber as ProductID divided by the weight. If the weight is missing, we add 0 in to the divide, which is likely to cause problems since this divides by infinity. If something does go wrong, we will print an error message of what occured. No red error text will occur, since the application is correctly running - the rror is captured by the code.

```
BEGIN TRY
    UPDATE SalesLT.Product
    SET ProductNumber = ProductID / ISNULL(Weight, 0);
END TRY
BEGIN CATCH
    PRINT 'The following error occurred:';
    PRINT ERROR_MESSAGE();
END CATCH;
```

Table 1.108: Displaying records 1 - 10

ErrorLogID	ErrorTime	UserName	ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLin
1	2018-03-14 12:16:08	dbo	8134	16	1	NA	
2	2018-03-16 13:25:48	dbo	8134	16	1	NA	
3	2018-03-16 13:27:24	dbo	8134	16	1	NA	
4	2018-03-16 13:27:33	dbo	8134	16	1	NA	
5	2018-03-16 13:27:50	dbo	8134	16	1	NA	
6	2018-03-23 04:03:28	dbo	8134	16	1	NA	
7	2018-03-23 04:08:49	dbo	8134	16	1	NA	
8	2018-03-23 04:20:56	dbo	8134	16	1	NA	
9	2018-03-23 04:21:37	dbo	8134	16	1	NA	
10	2018-03-23 04:22:19	dbo	8134	16	1	NA	

Another approach is to return the error to the application, so that it can do it's own error handling. Here we re THROW the error back to the client application. This time the error will be returned in red as an error.

```
BEGIN TRY
    UPDATE SalesLT.Product
    SET ProductNumber = ProductID / ISNULL(Weight, 0);
END TRY
BEGIN CATCH
    PRINT 'The following error occurred:';
    PRINT ERROR_MESSAGE();
    THROW;
END CATCH;
```

Anotehr more advanced way would be to create some variables in the CATCH block and use a stored procedure in the database. This stored procedure will store the error in to an error log table and throws that error back to the stored procedure. This means you can transport the strored proceedure across different databases without having to re-write code and still have a consistent way to handle errors.

```
BEGIN TRY
    UPDATE SalesLT.Product
    SET ProductNumber = ProductID / ISNULL(Weight, 0);
END TRY
BEGIN CATCH
    DECLARE @ErrorLogID as int, @ErrorMsg AS varchar(250);
    EXECUTE dbo.uspLogError @ErrorLogID OUTPUT;
    SET @ErrorMsg = 'The update failed because of an error. View error #'
                    + CAST(@ErrorLogID AS varchar)
                    + ' in the error log for details.';
    THROW 50001, @ErrorMsg, 0;
END CATCH;
```

Then we can view the error log if we want.

```
SELECT * FROM dbo.ErrorLog;
```

** Key Points ** * Use TRY…CATCH blocks in your Transact-SQL code to catch and handle exceptions. * A common exception handling pattern is to log the error, and then if the operation cannot be completed successfully, throw it (or a new custom error) to the calling application.

1.11.3 Transactions

We are concerned about protecting the integrity of the database. A transaction is a group of tasks, possibly inserting in to multiple tables. A transaction can be ACID

- Atomic - one unit of work that everythign contributes towards, you can split it up in to seperate parts
- Consistent - when the transaction is finished, there won't be any orphaned records or rows
- Isolated - whilst that transaction is happening, other transactions or queries are not effected, they won't get to see the in flight data
- Durable - one finished the data will be stored in the database unless we explicity DELETE it

A transaction must either fully succeed or wholly fail, we cannot have a partial succeed. So for instance if we add a new order, we might add summary info in to a order header table then all the prodcuts into the order detail table. We don't want half of this to work - just inserts in to one table - the whole thing, the transaction, must take place for it to be successful. If for some reason it something doesn't work, perhaps as a consequence for a system fail or network issue, we want the operation to fail not partially succeed or complete. SQL Server will handle this by itself, you don't need to add any transaction error codes or error handling, it will lock or isolate the data as required.

However, if there are multiple statements that you want to execute, it is best done as an encapsulated transaction explicitly. This is done by BEGIN TRANSACTION then COMMIT TRANSACTION if there is an error will can throw an error and use ROLLBACK TRANSACTION. These can be nexted TRANSACTIONS which can get complicated. If you ROLLBACK it will ROLLBACK right back to the beggining - so it is Atomic and Consistent. Rather than use ROLLBACK you can use XACT_ABORT which will rollback all transactions if an error is encountered.

So if we had some code as shown below, this would lead to a record being adding to the order header, but no details being added to the product table, since the product we are trying to add (ProductID 9999) does not exist. This would lead to an orphaned sales order header.

```
BEGIN TRY
    INSERT INTO SalesLT.SalesOrderHeader (DueDate, CustomerID, ShipMethod)
    VALUES
    (DATEADD(dd, 7, GETDATE()), 1, 'STD DELIVERY');

    DECLARE @SalesOrderID int = SCOPE_IDENTITY();

    INSERT INTO SalesLT.SalesOrderDetail (SalesOrderID, OrderQty, ProductID, UnitPrice, UnitPriceDiscou
    VALUES
    (@SalesOrderID, 1, 99999, 1431.50, 0.00);
END TRY
BEGIN CATCH
    PRINT ERROR_MESSAGE();
END CATCH;
```

So we want to add this code within a single transaction, so if one part fails the whole thing will. In this instance it wil try to do both parts individually, but the whole thing will fail as the second part - adding in an invalid productID - fails. So in effect it will undo the transaction - the first part which succeeds adding the order header. If you have many rows, this can therefore be time consuming as it might try to add 10,000 rows, then fails at a later stage, then has to go back and undo all those 10,000 rows. So you may want to add some logic at the beggining of the transaction, before this addition and error followed by rollback occurs.

```
BEGIN TRY
  BEGIN TRANSACTION
    INSERT INTO SalesLT.SalesOrderHeader (DueDate, CustomerID, ShipMethod)
```

```
    VALUES
    (DATEADD(dd, 7, GETDATE()), 1, 'STD DELIVERY');

    DECLARE @SalesOrderID int = SCOPE_IDENTITY();

    INSERT INTO SalesLT.SalesOrderDetail (SalesOrderID, OrderQty, ProductID, UnitPrice, UnitPriceDiscou
    VALUES
    (@SalesOrderID, 1, 99999, 1431.50, 0.00);
  COMMIT TRANSACTION
END TRY
BEGIN CATCH
  IF @@TRANCOUNT > 0
  BEGIN
    PRINT XACT_STATE();
    ROLLBACK TRANSACTION;
  END
  PRINT ERROR_MESSAGE();
  THROW 50001,'An insert failed. The transaction was cancelled.', 0;
END CATCH;
```

As previously discussed, we can use XACT_ABORT rather than ROLLBACK. Again we might want to add some logic at the beggining to check for validity before adding and undoing mutiple

```
SET XACT_ABORT ON;
BEGIN TRY
  BEGIN TRANSACTION
    INSERT INTO SalesLT.SalesOrderHeader (DueDate, CustomerID, ShipMethod)
    VALUES
    (DATEADD(dd, 7, GETDATE()), 1, 'STD DELIVERY');

    DECLARE @SalesOrderID int = SCOPE_IDENTITY();

    INSERT INTO SalesLT.SalesOrderDetail (SalesOrderID, OrderQty, ProductID, UnitPrice, UnitPriceDiscou
    VALUES
    (@SalesOrderID, 1, 99999, 1431.50, 0.00);
  COMMIT TRANSACTION
END TRY
BEGIN CATCH
  PRINT ERROR_MESSAGE();
  THROW 50001,'An insert failed. The transaction was cancelled.', 0;
END CATCH;
SET XACT_ABORT OFF;
```

** Key Points ** * Transactions are used to protect data integrity by ensuring that all data changes within a transaction succeed or fail as a unit.
* Individual Transact-SQL statements are inherently treated as transactions, and you can define explicit transactions that encompass mulitple * statements.
* Use the BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK TRANSACTION statements to manage transactions.
* Enable the XACT_ABORT option to automatically rollback all transactions if an exception occurs.
* Use the @**?** system variable and XACT_STATE system function to determine transaction status.

1.11.4 Lab Exercises

You are implementing a Transact-SQL script to delete orders, and you want to handle any errors that occur during the deletion process.

```
DECLARE @OrderID int = 0

-- Declare a custom error if the specified order doesn't exist
DECLARE @error VARCHAR(25) = 'Order #' + cast(@OrderID as VARCHAR) + ' does not exist';

IF NOT NULL (SELECT * FROM SalesLT.SalesOrderHeader WHERE SalesOrderID = @OrderID)
BEGIN
  THROW 50001, @error, 0;
END
ELSE
BEGIN
  SELECT FROM SalesLT.SalesOrderDetail WHERE SalesOrderID = @OrderID;
  DELETE FROM SalesLT.SalesOrderHeader WHERE SalesOrderID = @OrderID;
END
```

Now your code now throws an error if the specified order does not exist. Refine your code to catch this (or any other) error and print the error message to the user interface using the PRINT command. You can use BEGIN TRY, END TRY, BEGIN CATCH and END CATCH for this.

```
DECLARE @OrderID int = 71774
DECLARE @error VARCHAR(25) = 'Order #' + cast(@OrderID as VARCHAR) + ' does not exist';

-- Wrap IF ELSE in a TRY block
TRY CATCH
  IF NOT EXISTS (SELECT * FROM SalesLT.SalesOrderHeader WHERE SalesOrderID = @OrderID)
  BEGIN
    THROW 50001, @error, 0
  END
  ELSE
  BEGIN
    DELETE FROM SalesLT.SalesOrderDetail WHERE SalesOrderID = @OrderID;
    DELETE FROM SalesLT.SalesOrderHeader WHERE SalesOrderID = @OrderID;
  END
END CATCH
-- Add a CATCH block to print out the error
BEGIN CATCH
  PRINT ERROR_MESSAGE();
END CATCH
```

You have implemented error handling logic in some Transact-SQL code that deletes order details and order headers. However, you are concerned that a failure partway through the process will result in data inconsistency in the form of undeleted order headers for which the order details have been deleted.

Your task is to enhance the code you created in the previous challenge so that the two DELETE statements are treated as a single transactional unit of work.

```
DECLARE @OrderID int = 0
DECLARE @error VARCHAR(25) = 'Order #' + cast(@OrderID as VARCHAR) + ' does not exist';

BEGIN TRY
  IF NOT EXISTS (SELECT * FROM SalesLT.SalesOrderHeader WHERE SalesOrderID = @OrderID)
  BEGIN
```

```
    THROW 50001, @error, 0
  END
  ELSE
  BEGIN
    BEGIN TRANSACTION
    DELETE FROM SalesLT.SalesOrderDetail
    WHERE SalesOrderID = @OrderID;
    DELETE FROM SalesLT.SalesOrderHeader
    WHERE SalesOrderID = @OrderID;
    END TRANSACTION
  END
END TRY
BEGIN CATCH
  IF @@TRANCOUNT > 0
  BEGIN
    ROLLBACK TRANSACTION;
  END
  ELSE
  BEGIN
    PRINT ERROR_MESSAGE();
  END
END CATCH
```

1.12 Final Assessment

1.12.1 Section 1

Select the quantity per unit for all products in the Products table.

```
SELECT QuantityPerUnit
FROM Products;
```

Select the unique category IDs from the Products table.

```
SELECT DISTINCT CategoryID
FROM Products;
```

Select the names of products from the Products table which have more than 20 units left in stock.

```
SELECT ProductName
FROM Products
WHERE UnitsInStock > 20;
```

Select the product ID, product name, and unit price of the 10 most expensive products from the Products table.

```
SELECT TOP (10) ProductID, ProductName, UnitPrice
FROM Products
ORDER BY UnitPrice DESC;
```

Select the product ID, product name, and quantity per unit for all products in the Products table. Sort your results alphabetically by product name.

```
SELECT ProductID, ProductName, QuantityPerUnit
FROM Products
```

```
ORDER BY ProductName;
```

Select the product ID, product name, and unit price of all products in the Products table. Sort your results by number of units in stock, from greatest to least.

Skip the first 10 results and get the next 5 after that.

```
SELECT ProductID, ProductName, UnitPrice
FROM Products
ORDER BY UnitsInStock DESC
OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY;
```

Use STR, CONVERT, and NVARCHAR(30) where appropriate to display the first name, employee ID and birthdate (as Unicode in ISO 8601 format) for each employee in the Employees table.

```
SELECT FirstName + ' has an EmployeeID of ' + STR(EmployeeID, 1) + ' and was born ' +
       CONVERT(NVARCHAR(30), BirthDate, 126)
FROM Employees;
```

Select from the Orders table.

The first column of your result should be a single string in exactly the following format:

<> is from <>

If there is no ShipCity, then you should select ShipRegion, and if there is no ShipRegion you should select ShipCountry.

```
SELECT ShipName + ' is from ' +
  CASE
    WHEN ShipCity IS NULL THEN 'ShipRegion'
    ELSE ShipCity
  END
FROM Orders;
```

The above works, but doesn't properly handle ShipCountry, a better option is to use the COALESCE which returns the first non-null value. The AS destination is optional, if not specified the column will not have a title.

```
SELECT
ShipName + ' is from ' +
  COALESCE(ShipCity, ShipRegion,ShipCountry) AS destination
FROM Orders;
```

Select the ship name and ship postal code from the Orders table. If the postal code is missing, display 'unknown'. The following answer may be differnt for other dbs or flavours of SQL.

```
SELECT ShipName, ISNULL(ShipPostalCode, 'unknown')
FROM Orders;
```

Using the Suppliers table, select the company name, and use a simple CASE expression to display 'outdated' if the company has a fax number, or 'modern' if it doesn't.

```
SELECT CompanyName,
  CASE
    WHEN Fax IS NOT NULL THEN 'outdated'
    ELSE 'modern'
  END AS Status
FROM Suppliers;
```

1.12.2 Section 2

Get the order ID and unit price for each order by joining the Orders table and the Order Details table. Note that you need to use [Order Details] since the table name contains whitespace.

```
SELECT o.OrderID, od.UnitPrice
FROM Orders AS o
INNER JOIN [Order Details] AS od
ON o.OrderID = od.OrderID;
```

Get the order ID and first name of the associated employee by joining the Orders and Employees tables.

```
SELECT o.OrderID, e.FirstName
FROM Orders AS o
INNER JOIN Employees as e
ON o.EmployeeID = e.EmployeeID;
```

Get the employee ID and related territory description for each territory an employee is in, by joining the Employees, EmployeeTerritories and Territories tables.

```
SELECT e.EmployeeID, t.TerritoryDescription
FROM Employees AS e
INNER JOIN EmployeeTerritories AS et
ON e.EmployeeID = et.EmployeeID
INNER JOIN Territories AS t
ON et.TerritoryID = t.TerritoryID;
```

Select all the different countries from the Customers table and the Suppliers table using UNION.

```
SELECT Country
FROM Customers
UNION
SELECT Country
FROM Suppliers
ORDER BY Country;
```

Select all the countries, including duplicates, from the Customers table and the Suppliers table using UNION ALL.

```
SELECT Country
FROM Customers
UNION ALL
SELECT Country
FROM Suppliers
ORDER BY Country;
```

Using the Products table, get the unit price of each product, rounded to the nearest dollar.

```
SELECT ROUND(UnitPrice, 0)
FROM Products;
```

Using the Products table, get the total number of units in stock across all products.

```
SELECT SUM(UnitsInStock)
FROM Products;
```

Using the Orders table, get the order ID and year of the order by using YEAR(). Alias the year as OrderYear.

```
SELECT OrderID, YEAR(OrderDate) AS OrderYear
FROM Orders;
```

Using the Orders table, get the order ID and month of the order by using DATENAME(). Alias the month as OrderMonth.

```
SELECT OrderID, DATENAME(mm, OrderDate) AS OrderMonth
FROM Orders;
```

Use LEFT() to get the first two letters of each region description from the Region table.

```
SELECT LEFT(RegionDescription, 2)
FROM Region;
```

Using the Suppliers table, select the city and postal code for each supplier, using WHERE and ISNUMERIC() to select only those postal codes which have no letters in them.

```
SELECT City, PostalCode
FROM Suppliers
WHERE ISNUMERIC(PostalCode) = 1;
```

Use LEFT() and UPPER() to get the first letter (capitalized) of each region description from the Region table.

```
SELECT UPPER(LEFT(RegionDescription, 1))
FROM Region;
```

1.12.3 Section 3

Use a subquery to get the product name and unit price of products from the Products table which have a unit price greater than the average unit price from the Order Details table.

Note that you need to use [Order Details] since the table name contains whitespace.

```
SELECT ProductName, UnitPrice
FROM Products
WHERE UnitPrice >
  (SELECT AVG(UnitPrice)
  FROM [Order Details]);
```

Select from the Employees and Orders tables. Use a subquery to get the first name and employee ID for employees who were associated with orders which shipped from the USA.

```
SELECT FirstName, EmployeeID
FROM Employees
WHERE EmployeeID IN
  (SELECT DISTINCT(EmployeeID)
  FROM Orders
  WHERE ShipCountry = 'USA');
```

Create a new temporary table called ProductNames which has one field called ProductName (a VARCHAR of max length 40). Insert into this table the names of every product from the Products table. Select all columns from the ProductNames table you created.

```
CREATE TABLE #ProductNames
(ProductName VARCHAR(40))

INSERT INTO #ProductNames
SELECT ProductName
FROM Products
```

```
SELECT *
FROM #ProductNames;
```

1.12.4 Section 4

Use CHOOSE() and MONTH() to get the season in which each order was shipped from the Orders table. You should select the order ID, shipped date, and then the season aliased as ShippedSeason. Be careful to filter out any NULL shipped dates.

```
SELECT OrderID, ShippedDate,
  CHOOSE (MONTH(ShippedDate), 'Winter', 'Winter', 'Spring', 'Spring', 'Spring', 'Summer', 'Summer', 'Su
FROM Orders
WHERE ShippedDate IS NOT NULL;
```

Using the Suppliers table, select the company name and use a simple IIF expression to display 'outdated' if a company has a fax number, or 'modern' if it doesn't. Alias the result of the IIF expression to Status.

```
SELECT CompanyName,
IIF(FAX IS NOT NULL, 'outdated', 'modern') AS Status
FROM Suppliers;
```

Select from the Customers, Orders, and Order Details tables. Note that you need to use [Order Details] since the table name contains whitespace.

Use GROUP BY and ROLLUP() to get the total quantity ordered by all countries, while maintaining the total per country in your result set.

Your first column should be the country, and the second column the total quantity ordered by that country, aliased as TotalQuantity.

```
SELECT c.Country, SUM(od.Quantity) AS TotalQuantity
FROM Customers AS c
JOIN Orders as o
ON c.CustomerID = o.CustomerID
JOIN [Order Details] AS od
ON o.OrderID = od.OrderID
GROUP BY ROLLUP (c.Country);
```

From the Customers table, use GROUP BY to select the country, contact title, and count of that contact title aliased as Count, grouped by country and contact title (in that order).

Then use CASE WHEN, GROUPING_ID(), and ROLLUP() to add a column called Legend, which shows one of two things:

When the GROUPING_ID is 0, show '' (i.e., nothing) When the GROUPING_ID is 1, show Subtotal for << Country >>'

```
SELECT Country, ContactTitle, COUNT(ContactTitle) AS Count,
  CASE
    WHEN GROUPING_ID(Country, ContactTitle) = 0 THEN ''
    WHEN GROUPING_ID(Country, ContactTitle) = 1 THEN 'Subtotal for ' + Country
  END AS Legend
FROM Customers
GROUP BY ROLLUP (Country, ContactTitle);
```

Convert the following query to be pivoted, using PIVOT().

SELECT CategoryID, AVG(UnitPrice) FROM Products GROUP BY CategoryID;

Note the first part of the answer defines the item that appears in the row, the second 'Per Category' is the table title.

```
SELECT 'Average Unit Price' AS 'Per Category',
[1], [2], [3], [4], [5], [6], [7], [8]
FROM
(
    SELECT CategoryID, UnitPrice
    FROM Products)
    AS SourceTable
PIVOT
(
    AVG(UnitPrice)
    FOR CategoryID IN ([1], [2], [3], [4], [5], [6], [7], [8]))
    AS PivotTable;
```

So we go from long to wide format as illustrated below.

Insert into the Region table the region ID 5 and the description 'Space'.

Then, in a second query, select the newly inserted data from the table using a WHERE clause.

```
INSERT INTO Region
VALUES
(5, 'Space');

SELECT *
FROM Region
WHERE RegionID = 5;
```

Update the region descriptions in the Region table to be all uppercase, using SET and UPPER().

Next, select all data from the table to view your updates.

```
UPDATE Region
SET RegionDescription = UPPER(RegionDescription);

SELECT *
FROM Region;
```

Declare a custom region **?** called 'Space', of type NVARCHAR(25).

Use IF NOT EXISTS, ELSE, and BEGIN..END to throw an error message 'Error!' if **?** is not in the Region table. Do this check using SELECT *.

If the specified description does exist, you should select all columns for that region from the Region table.

```
DECLARE @region AS NVARCHAR(25) = 'Space';

TRY CATCH
IF NOT EXISTS (SELECT * FROM Region WHERE RegionDescription = @region)
  BEGIN
    PRINT 'Error!'
  END
  ELSE
  BEGIN
    SELECT * FROM Region;
  END
END CATCH;
```

```sql
SELECT CategoryID, AVG(UnitPrice) AS AvgUnitPrice
FROM Products
GROUP BY CategoryID;
```

CategoryID	AvgUnitPrice
1	37.98
2	23.06
3	25.16
4	28.73
5	20.25
6	54.01
7	32.37
8	20.68

```sql
SELECT 'Average Unit Price' AS 'Per Category',
[1], [2], [3], [4], [5], [6], [7], [8]
FROM
(
    SELECT CategoryID, UnitPrice
    FROM Products)
    AS SourceTable
PIVOT
(
    AVG(UnitPrice)
    FOR CategoryID IN ([1], [2], [3], [4], [5], [6], [7], [8]))
    AS PivotTable;
```

Per Category	1	2	3	4	5
Average Unit Price	37.98	23.06	25.16	28.73	20.25

Figure 1.3: Pivot Answer Section 4 Question 5

(#fig:Pivot Answer)

Chapter 2

Supervised Learning In R Classification

Notes taken during/inspired by the DataCamp course 'Supervised Learning In R Classification' by Brett Lantz.

Course Handouts

- Part 1 - k-Nearest Neighbors (kNN)
- Part 2 - Naive Bayes
- Part 3 - Logistic Regression
- Part 4 - Classification Trees

Other useful links

- Introduction to Tree Based Methods from ISL

2.1 k-Nearest Neighbors (kNN)

Machine Learning uses computers to turn data in to insight and action. This course looks at supervised learning, where we train the machine to learn from prior examples. When the concept to be learned is a set of categories, the objective is classification. In autonomous driving, we may want our car to undertake some action, like brake, when certain roads signs are observed. After a period of time observing a drivers behaviour, the computer will build a database of signs and appropriate responses. Some if one stop sign is observed, it will try and place where this sign is in relation to other signs it has seen before, then determine what type or class the sign is and do the appropriate action.

To do this, it calculates the distance between the new sign and past signs, using co-ordinates in feature space. For instance, signs could be classified in three dimensions using RGB, then signs of a similar colour will be located together. Distance is then measured based on the signs location in the co-ordinate space. We could for instance measure Euclidean distance, which is used by many NN algorithms, and can be done in R using the knn function.

First let us load the data.

```
# Load the signs data
traffic_signs <- read.csv("./files/SLinRClassification/knn_traffic_signs.csv", stringsAsFactors = FALSE)
signs <- subset(traffic_signs, sample == "train")
signs$id <- NULL
```

```
signs$sample <- NULL
signs_test <- subset(traffic_signs, sample == "test")
signs_test$id <- NULL
signs_test$sample <- NULL

library(class)
next_sign <- signs[146,]
signs     <- signs[1:145,]
sign_types <- signs$sign_type
signs_actual <- signs_test$sign_type
rm(traffic_signs)

#Load the locations and where9am data
locations <-  read.csv("./files/SLinRClassification/locations.csv", stringsAsFactors = FALSE)
where9am <- subset(locations, hour == 9)
```

After several trips with a human behind the wheel, it is time for the self-driving car to attempt the test course alone.

As it begins to drive away, its camera captures an image.LEt's write some code so that a kNN classifier helps the car recognize the sign.

```
# Load the 'class' package
library(class)

# Create a vector of labels
sign_types <- signs$sign_type

# Classify the next sign observed - the first column of the signs dataset is removed as the class is sp
knn(train = signs[-1], test = next_sign, cl = sign_types)
```

Each previously observed street sign was divided into a 4x4 grid, and the red, green, and blue level for each of the 16 center pixels is recorded. The result is a dataset that records the sign_type as well as 16 x 3 = 48 color properties of each sign.

```
# Examine the structure of the signs dataset
str(signs)
```

```
## 'data.frame':    145 obs. of  49 variables:
##  $ sign_type: chr  "pedestrian" "pedestrian" "pedestrian" "pedestrian" ...
##  $ r1       : int  155 142 57 22 169 75 136 149 13 123 ...
##  $ g1       : int  228 217 54 35 179 67 149 225 34 124 ...
##  $ b1       : int  251 242 50 41 170 60 157 241 28 107 ...
##  $ r2       : int  135 166 187 171 231 131 200 34 5 83 ...
##  $ g2       : int  188 204 201 178 254 89 203 45 21 61 ...
##  $ b2       : int  101 44 68 26 27 53 107 1 11 26 ...
##  $ r3       : int  156 142 51 19 97 214 150 155 123 116 ...
##  $ g3       : int  227 217 51 27 107 144 167 226 154 124 ...
##  $ b3       : int  245 242 45 29 99 75 134 238 140 115 ...
##  $ r4       : int  145 147 59 19 123 156 171 147 21 67 ...
##  $ g4       : int  211 219 62 27 147 169 218 222 46 67 ...
##  $ b4       : int  228 242 65 29 152 190 252 242 41 52 ...
##  $ r5       : int  166 164 156 42 221 67 171 170 36 70 ...
##  $ g5       : int  233 228 171 37 236 50 158 191 60 53 ...
##  $ b5       : int  245 229 50 3 117 36 108 113 26 26 ...
##  $ r6       : int  212 84 254 217 205 37 157 26 75 26 ...
```

```
##  $ g6       : int  254 116 255 228 225 36 186 37 108 26 ...
##  $ b6       : int  52 17 36 19 80 42 11 12 44 21 ...
##  $ r7       : int  212 217 211 221 235 44 26 34 13 52 ...
##  $ g7       : int  254 254 226 235 254 42 35 45 27 45 ...
##  $ b7       : int  11 26 70 20 60 44 10 19 25 27 ...
##  $ r8       : int  188 155 78 181 90 192 180 221 133 117 ...
##  $ g8       : int  229 203 73 183 110 131 211 249 163 109 ...
##  $ b8       : int  117 128 64 73 9 73 236 184 126 83 ...
##  $ r9       : int  170 213 220 237 216 123 129 226 83 110 ...
##  $ g9       : int  216 253 234 234 236 74 109 246 125 74 ...
##  $ b9       : int  120 51 59 44 66 22 73 59 19 12 ...
##  $ r10      : int  211 217 254 251 229 36 161 30 13 98 ...
##  $ g10      : int  254 255 255 254 255 34 190 40 27 70 ...
##  $ b10      : int  3 21 51 2 12 37 10 34 25 26 ...
##  $ r11      : int  212 217 253 235 235 44 161 34 9 20 ...
##  $ g11      : int  254 255 255 243 254 42 190 44 23 21 ...
##  $ b11      : int  19 21 44 12 60 44 6 35 18 20 ...
##  $ r12      : int  172 158 66 19 163 197 187 241 85 113 ...
##  $ g12      : int  235 225 68 27 168 114 215 255 128 76 ...
##  $ b12      : int  244 237 68 29 152 21 236 54 21 14 ...
##  $ r13      : int  172 164 69 20 124 171 141 205 83 106 ...
##  $ g13      : int  235 227 65 29 117 102 142 229 125 69 ...
##  $ b13      : int  244 237 59 34 91 26 140 46 19 9 ...
##  $ r14      : int  172 182 76 64 188 197 189 226 85 102 ...
##  $ g14      : int  228 228 84 61 205 114 171 246 128 67 ...
##  $ b14      : int  235 143 22 4 78 21 140 59 21 6 ...
##  $ r15      : int  177 171 82 211 125 123 214 235 85 106 ...
##  $ g15      : int  235 228 93 222 147 74 221 252 128 69 ...
##  $ b15      : int  244 196 17 78 20 22 201 67 21 9 ...
##  $ r16      : int  22 164 58 19 160 180 188 237 83 43 ...
##  $ g16      : int  52 227 60 27 183 107 211 254 125 29 ...
##  $ b16      : int  53 237 60 29 187 26 227 53 19 11 ...
```

```
# Count the number of signs of each type
table(signs$sign_type)
```

```
##
## pedestrian      speed       stop
##         46         49         50
```

```
# Check r10's average red level by sign type
aggregate(r10 ~ sign_type, data = signs, mean)
```

```
##    sign_type       r10
## 1 pedestrian 113.71739
## 2      speed  80.63265
## 3       stop 131.98000
```

Next we want to try and see how well the predict signs match the actual signs classified by a human. We will also create a confusion matrix to see where it worked and a accuracy measure.

```
# Use kNN to identify the test road signs
sign_types <- signs$sign_type
signs_pred <- knn(train = signs[-1], test = test_signs[-1], cl = sign_types)

# Create a confusion matrix of the actual versus predicted values
```

```
signs_actual <- test_signs$sign_type
table(signs_pred, signs_actual)

# Compute the accuracy
mean(signs_pred == signs_actual)
```

When we use kNN, the K signifies the number of neighbours to consider when making the classification. Unless specified, R will use k = 1 i.e. it will only consider the nearest neighbour. However, as other elements, such as road sign background and lighting, might cause an incorrect road sign to be the nearest based on such factors, this can cause problems. If we use a greater value for K, there is in effect a vote from the nearest neighbours (k) on which sign is the most likely. In the case of a tie, the winner is typically set by random. Setting a high value for K isn't always the best approach, as it can introduce noise in to a pattern. Setting a low value for k might enable it to identify more subtle patterns, but may lead to overfitting and errors.

** Some people suggest setting K to the square root of the number of observations in the training data ** So if we observed 100 road signs, we would set k to 10. A better approach would be to set k to mutiple values, then run the model against some unseen (test) data and see how it performs.

In the following example we set k = 1 (default), 7 then 15 and compare the levels of accuracy.

```
# Compute the accuracy of the baseline model (default k = 1)
k_1 <- knn(train = signs[,-1], test = signs_test[,-1], cl = signs[,1])
mean(k_1 == signs_actual)

# Modify the above to set k = 7
k_7 <- knn(train = signs[,-1], test = signs_test[,-1], cl = signs[,1], k = 7)
mean(k_7 == signs_actual)

# Set k = 15 and compare to the above
k_15 <- knn(train = signs[,-1], test = signs_test[,-1], cl = signs[,1], k = 15)
mean(k_15 == signs_actual)
```

K = 7 gives the highest level of accuracy.

When multiple nearest neighbors hold a vote, it can sometimes be useful to examine whether the voters were unanimous or widely separated. There is a option we can set using prob = TRUE parameter to compute the vote proportions for a kNN model.

```
# Use the prob parameter to get the proportion of votes for the winning class
sign_pred <- knn(train = signs[,-1], test = signs_test[,-1], cl = signs[,1], k = 7, prob = TRUE)

# Get the "prob" attribute from the predicted classes
sign_prob <- attr(sign_pred, "prob")

# Examine the first several predictions
head(sign_pred)

# Examine the proportion of votes for the winning class
head(sign_prob)
```

kNN models calculate distance, therefore kNN assumes the data is in numeric format e.g. we don't have yellow we have RGB values. If something cannot be easily converted to a numeric, we can create dummy variables e.g. to indicate the shape such as triangle = 1, square = 0, circle = 0 might be for one particular sign. This dummy set of vars can then be used in distance calculations.

When we then introduce another variable - the original RGB 0-255 and shape 0-1 - we can have problems

as they are on different scales. The variables with a larger scale can have a disproportionate effect on the calculation of distance, therefore we need to normalise the data. So we change our RGB vlues to go in to the range of 0 to 1. To do this you could create a function yourself, or use one which is included in some packages like caret.

2.2 Naive Bayes

Bayes proposed rules for estimating probabilties in light of historic data. This section will look at applying these methods to phone data to help forecast action. We take the number of times and event happened / all possible events (how many times it could of occured). But we should also incorporate other variables, like time of day, to give a single probability estimate. This therefore becomes the joint probability P(A and B) e.g. P(work and evening) = 1%, P(work and afternoon) = 20%.

We also have the conditional probability P(A¦B) = P(A and B) / P(B) = P (work ¦ evening) = 1/25 = 4% or P (work ¦ afternoon) = 20/25 = 80%. WE can do these calculations using the the naivebayes package in R.

The where9am data frame contains 91 days (thirteen weeks) worth of data in which Brett recorded his location at 9am each day as well as whether the daytype was a weekend or weekday.

```
# Compute P(A)
p_A <- nrow(subset(where9am, location == 'office')) / nrow(where9am)

# Compute P(B)
p_B <- nrow(subset(where9am, daytype == 'weekday')) / nrow(where9am)

# Compute the observed P(A and B)
p_AB <- nrow(subset(where9am, location == 'office' & daytype == 'weekday')) / nrow(where9am)

# Compute then print P(A | B)
p_A_given_B <- p_AB / p_B
p_A_given_B
```

```
## [1] 0.6
```

Next we can use th naive bayes model to predict Brett's location and two different times and days - 9am on a Thursday and 9 am on a Saturday.

```
# Setup our variables for prediction
days <- data.frame(daytype = factor(c("weekday","weekend")))
thursday9am <- subset(days, daytype == 'weekday')
saturday9am <- subset(days, daytype == 'weekend')

# Load the naivebayes package
library(naivebayes)
```

```
## Warning: package 'naivebayes' was built under R version 3.4.3
```

```
# Build the location prediction model
locmodel <- naive_bayes(location ~ daytype, data = where9am)

# Predict Thursday's 9am location
predict(locmodel, thursday9am)
```

```
## [1] office
## Levels: appointment campus home office
```

```
# Predict Saturdays's 9am location
predict(locmodel, saturday9am)
```

```
## [1] home
## Levels: appointment campus home office
```

Brett is most likely at the office at 9am on a Thursday, but at home at the same time on a Saturday. Note that the structure of the variables and their levels matter, for instance if we had a factor variable with just one level and not all possible levels e.g. saturday9am with a daytype of weekend but a factor variable with only the level observed (weekend) and not including the other possability of weekday in the factor levels, we would get a different result - a result of office. So data structure is important.

We can examine the *a priori* (overall) probabilities just by examing the model and some of it's sub-components.

```
locmodel$tables$daytype
```

```
##
## daytype   appointment    campus      home    office
##   weekday   1.0000000 1.0000000 0.3658537 1.0000000
##   weekend   0.0000000 0.0000000 0.6341463 0.0000000
```

```
locmodel$prior
```

```
##
## appointment      campus        home      office
##  0.01098901  0.10989011  0.45054945  0.42857143
```

We can then compare these probabilities to those we obtain when using a model.

```
# Obtain the predicted probabilities for Thursday at 9am
predict(locmodel, thursday9am, type = "prob")
```

```
##      appointment    campus      home office
## [1,]  0.01538462 0.1538462 0.2307692    0.6
```

```
# Obtain the predicted probabilities for Saturday at 9am
predict(locmodel, saturday9am, type = "prob")
```

```
##      appointment campus home office
## [1,]           0      0    1      0
```

So on a Saturday there is 0% chance Brett is at the office and a 100% chance he is at home.

In Bayesian statistics, we say that independent events occur when knowing the outcome of one event does not help predict the other. For example, knowing if it's raining in London doesn't help you predict the weather in Manchester. The weather events in the two cities are independent of each other.

2.2.1 Putting the Naivety in Naive Bayes

With simple conditions where we have one predictor, daytime of the week, the conditional probability is based on the overlap between the two events. As we add more events, the degree of overlap illistrated through Venn diagrams can become complex and messy. The computer will find it inefficient to calculate the overlap, so the computer uses a shortcut to calculate the conditional probability we hope to compute.

Instead of looking for the intersection between all possible events, the computer will make a naive assumption about the data, it assumes that the events are independent. When this occurs, the joint probability can be calculated by mutiplying the individual probabilities. So simpler calculations can be performed based on limited intersections mutiplied together.

One of the problems with this approach is that when an event has not been previously observed - such as going to work on a weekend - we immediatly get calculation problems, since this individual probability is zero, so the results become zero despite how many other conditions in the caulaution may be likely. The solution is to add a small amount, such as 1%, to all the individual components - we call this the Laplace correction. So there will be at least some predicted probability, even if it has never been seen before.

Next we will build a more complicated model using the locations data, which is similar to the where9am data but contains 24hours of data recorded at hourly intervals over 13 weeks and also contains other variables such as hour type.

```
head(locations, n = 10)
```

```
##    month day   weekday daytype hour hourtype location
## 1      1   4 wednesday weekday    0    night     home
## 2      1   4 wednesday weekday    1    night     home
## 3      1   4 wednesday weekday    2    night     home
## 4      1   4 wednesday weekday    3    night     home
## 5      1   4 wednesday weekday    4    night     home
## 6      1   4 wednesday weekday    5    night     home
## 7      1   4 wednesday weekday    6  morning     home
## 8      1   4 wednesday weekday    7  morning     home
## 9      1   4 wednesday weekday    8  morning     home
## 10     1   4 wednesday weekday    9  morning   office
```

Then we setup our variables to be predicted

```
weekday_afternoon <- locations[13,c(4,6,7)]
weekday_evening <- locations[19,c(4,6,7)]
weekend_afternoon <- locations[85,c(4,6,7)]
```

Then we build our mutiple variable model and predict

```
# Build a NB model of location
locmodel <- naive_bayes(location ~ daytype + hourtype, data = locations)

# Predict Brett's location on a weekday afternoon
predict(locmodel, weekday_afternoon)
```

```
## [1] office
## Levels: appointment campus home office restaurant store theater
```

```
# Predict Brett's location on a weekday evening
predict(locmodel, weekday_evening)
```

```
## [1] home
## Levels: appointment campus home office restaurant store theater
```

Or with probabilities

```
# Predict Brett's location on a weekday afternoon
predict(locmodel, weekday_afternoon, type = "prob")
```

```
##      appointment     campus      home    office restaurant       store
## [1,] 0.004300045 0.08385089 0.2482618 0.5848062 0.07304769 0.005733394
##      theater
## [1,]       0
```

```
# Predict Brett's location on a weekday evening
predict(locmodel, weekday_evening, type = "prob")
```

```
##      appointment      campus      home     office restaurant      store
## [1,] 0.004283788 0.004283788 0.7903588 0.05997303 0.04400138 0.09709919
##      theater
## [1,]       0
```

Next we will try to predict where Brett would be on a weekend afternoon. Since there are some locations he has never been to before at this time of day e.g. the campus, office or theatre, we will predict probabilities both with and without the Laplace correction. Adding the Laplace correction allows for the small chance that Brett might go to the office on the weekend in the future. Without the Laplace correction, some potential outcomes may be predicted to be impossible.

```
# Observe the predicted probabilities for a weekend afternoon
predict(locmodel, weekend_afternoon, type = "prob")
```

```
##      appointment campus      home office restaurant      store theater
## [1,]  0.02472535      0 0.8472217      0  0.1115693 0.01648357       0
```

```
# Build a new model using the Laplace correction
locmodel2 <- naive_bayes(location ~ daytype + hourtype, laplace =1, data = locations)

# Observe the new predicted probabilities for a weekend afternoon
predict(locmodel2, weekend_afternoon, type = "prob")
```

```
##      appointment      campus      home      office restaurant      store
## [1,]  0.01107985 0.005752078 0.8527053 0.008023444  0.1032598 0.01608175
##          theater
## [1,] 0.003097769
```

2.2.2 Applying Naive Bayes (NB) to other problems

NB is useful when attirbutes from multiple factors need to be considered at the same time, then evaluated as a whole, a bit like a Doctor looking at a patient then making a reccomendation. There are some considerations to be aware of whan apply NB to other scenarios.

Bayes works by calculating conditional probabilites, it builds frequency tables of the number of times a particular condition overlaps with our condition of interest, then the probabilities are multiplied Naively in a chain of events. Each predictor typically comprises a set of categories, numeric data like time of day or age, are difficult without modification. Unstructured text data also causes problems.

We can bin numeric data - like age bands - to make Bayes handle the data better. Or use quintiles. Text data is best adjusted using the bag of words approach. This in effect creates a table, where each sentenence or document becomes a row, and the columns become some numeric value for the presence of words, usually dummy vars. NB trained with the bag of words approach can be very effective text classifiers.

2.3 Logistic regression - binary predictions with regression

Regression models are one of the commonest forms of machine learning. If we have a binary outcome (1 or 0), using a standard regression may result in results either above or below this range. With logistic regression, we create a curve, a logistic function (s curve). Our result therfore becomes a probability between 0 and 1. We can caulate this using a glm model, setting the family to binomial, which we can then use in the predict model. If we use the type = “response” argument with predict, we get the predicted probabilites rather than the default log-odds. WE then use these probabilites and set some parameters using an ifelse statement e.g. if the predicted probability is greater than 50%, then we might say that is a 1 e.g. pred <- ifelse(prob >0.5, 1, 0).

The donors dataset contains 93,462 examples of people mailed in a fundraising solicitation for paralyzed military veterans. The donated column is 1 if the person made a donation in response to the mailing and 0 otherwise. The remaining columns are features of the prospective donors that may influence their donation behavior - the independent variables.

```
# Load the data
donors <- read.csv("./files/SLinRClassification/donors.csv", stringsAsFactors = TRUE)

# Examine the dataset to identify potential independent variables
str(donors)
```

```
## 'data.frame':    93462 obs. of  13 variables:
##  $ donated          : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ veteran          : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ bad_address      : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ age              : int  60 46 NA 70 78 NA 38 NA NA 65 ...
##  $ has_children     : int  0 1 0 0 1 0 1 0 0 0 ...
##  $ wealth_rating    : int  0 3 1 2 1 0 2 3 1 0 ...
##  $ interest_veterans: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ interest_religion: int  0 0 0 0 1 0 0 0 0 0 ...
##  $ pet_owner        : int  0 0 0 0 0 0 1 0 0 0 ...
##  $ catalog_shopper  : int  0 0 0 0 1 0 0 0 0 0 ...
##  $ recency          : Factor w/ 2 levels "CURRENT","LAPSED": 1 1 1 1 1 1 1 1 1 1 ...
##  $ frequency        : Factor w/ 2 levels "FREQUENT","INFREQUENT": 1 1 1 1 1 2 2 1 2 2 ...
##  $ money            : Factor w/ 2 levels "HIGH","MEDIUM": 2 1 2 2 2 2 2 2 2 2 ...
```

```
# Explore the dependent variable
table(donors$donated)
```

```
##
##     0     1
## 88751  4711
```

```
# Build the donation model
donation_model <- glm(donated ~ bad_address + interest_religion + interest_veterans,
                      data = donors, family = "binomial")

# Summarize the model results
summary(donation_model)
```

```
##
## Call:
## glm(formula = donated ~ bad_address + interest_religion + interest_veterans,
##     family = "binomial", data = donors)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.3480  -0.3192  -0.3192  -0.3192   2.5678
##
## Coefficients:
##                    Estimate Std. Error  z value Pr(>|z|)
## (Intercept)       -2.95139    0.01652 -178.664   <2e-16 ***
## bad_address       -0.30780    0.14348   -2.145   0.0319 *
## interest_religion  0.06724    0.05069    1.327   0.1847
## interest_veterans  0.11009    0.04676    2.354   0.0186 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 37330  on 93461  degrees of freedom
## Residual deviance: 37316  on 93458  degrees of freedom
## AIC: 37324
##
## Number of Fisher Scoring iterations: 5
```

We can anow apply this model. Remember that the default output is the log-odds of an outcome, so we need to use type = "response" to convert this to a probability.

```
# Estimate the donation probability
donors$donation_prob <- predict(donation_model, type = "response")

# Find the donation probability of the average prospect
mean(donors$donated)
```

```
## [1] 0.05040551
```

```
# Predict a donation if probability of donation is greater than average (0.0504)
donors$donation_pred <- ifelse(donors$donation_prob > 0.0504, 1, 0)

# Calculate the model's accuracy
mean(donors$donation_pred == donors$donated)
```

```
## [1] 0.794815
```

So the model appears to be accurate - circa 80%. But, the outcome of interest is quite rare (the second element above, the mean) is only around 5%, so had the model predicted not donated for every person, it would have been accurate 95% of the time.

To visual;ise this trade off between positive and negative predictions we can use an ROC curve. With an ROC curve we want to be away from the diagonal (an Area Under the Curve or AUC of 0.5) and towards the top left hand corner (top left would be an AUC of 1.0). But as curves of different shapes/gradients can have the same AUC, we should also look at the curve as well as the AUC value.

Next we can plot an ROC and calculate the AUC for our model.

```
# Load the pROC package
library(pROC)
```

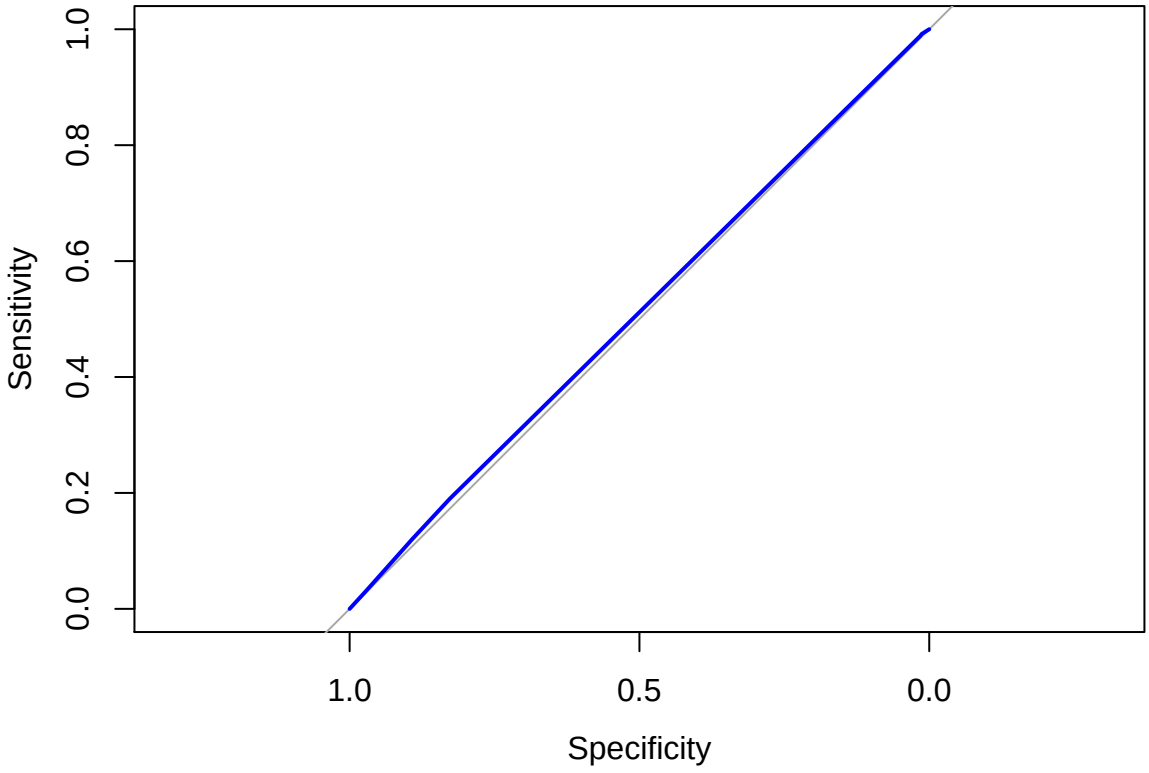
```
## Warning: package 'pROC' was built under R version 3.4.4
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
## Attaching package: 'pROC'
```

```
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
# Create a ROC curve
ROC <- roc(donors$donated, donors$donation_prob)

# Plot the ROC curve
plot(ROC, col = "blue")
```




```
# Calculate the area under the curve (AUC)
auc(ROC)
```

```
## Area under the curve: 0.5102
```

With the line almost on the diagonal and an AUC of just over 0.5 our model isn't doing much better than making predictions at random.

2.3.1 Dummy variables, missing data, and interactions

All of the variables in a regression analysis must be numeric e.g. all categorical data must be represented as a number. With missing data, we can no longer use this information to make predictions.

We use dummy variables - also sometimes called one hot encoding - for logistic regression. The GLM function will automatically convert factor type variables in to dummy vars used in the model. We just use the factor function to the data first, if the data/variable is not already a factor.

If we have missing data, our model will automatically exclude these cases (rows) in the model. You can create a categorical variable option (a factor level) for missing e.g. low, medium, high and missing. However for numerical data we have more options, perhaps we might use the mean or median, or build a more complex model if we like to impute this missing variable. Either way, it is a good idea to add a column/variable to indicate if a variable was imputed. Sometimes, this missing_var (1 = yes) can become an important predictor in a model.

An interaction effect considers that certain variables, when combined, may have more of an influence together than the sum of their individual components. The combination may strenthen, weaken or completly eliminate the strength of the individual predictors. In the model function, we can use the multiplication symbol (*) to indicate an interaction. The result will include individual components as well as combined effects.

In the next example, we classify a variable as a factor (wealth_rating), then set the reference category for the variable to Medium and finally build a model with this variable.

```
# Convert the wealth rating to a factor
donors$wealth_rating <- factor(donors$wealth_rating, levels = c(0,1,2,3), labels = c("Unknown", "Low",

# Use relevel() to change reference category
donors$wealth_rating <- relevel(donors$wealth_rating, ref = "Medium")

# See how our factor coding impacts the model
summary(glm(donated ~ wealth_rating, data = donors, family = "binomial"))
```

```
##
## Call:
## glm(formula = donated ~ wealth_rating, family = "binomial", data = donors)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.3320  -0.3243  -0.3175  -0.3175   2.4582
##
## Coefficients:
##                       Estimate Std. Error z value Pr(>|z|)
## (Intercept)          -2.91894    0.03614 -80.772   <2e-16 ***
## wealth_ratingUnknown -0.04373    0.04243  -1.031    0.303
## wealth_ratingLow     -0.05245    0.05332  -0.984    0.325
## wealth_ratingHigh     0.04804    0.04768   1.008    0.314
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 37330  on 93461  degrees of freedom
## Residual deviance: 37323  on 93458  degrees of freedom
## AIC: 37331
##
## Number of Fisher Scoring iterations: 5
```

As mentioned before, in cases where a variable is NA, R will exclude these cases when building a model. Therefore we should try and attempt to handle such cases, one option is imputation as shown below. We also create a new variable to indicate whether the age value was imputed, as a missing value can be an indicator of something more meaningful going on, it could be related to the outcome.

```
# Find the average age among non-missing values
summary(donors$age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    1.00   48.00   62.00   61.65   75.00   98.00   22546
```

```
# Impute missing age values with mean(age)
donors$imputed_age <- ifelse(is.na(donors$age), round(mean(donors$age, na.rm = TRUE),2), donors$age)

# Create missing value indicator for age
donors$missing_age <- ifelse(is.na(donors$age),1,0)
```

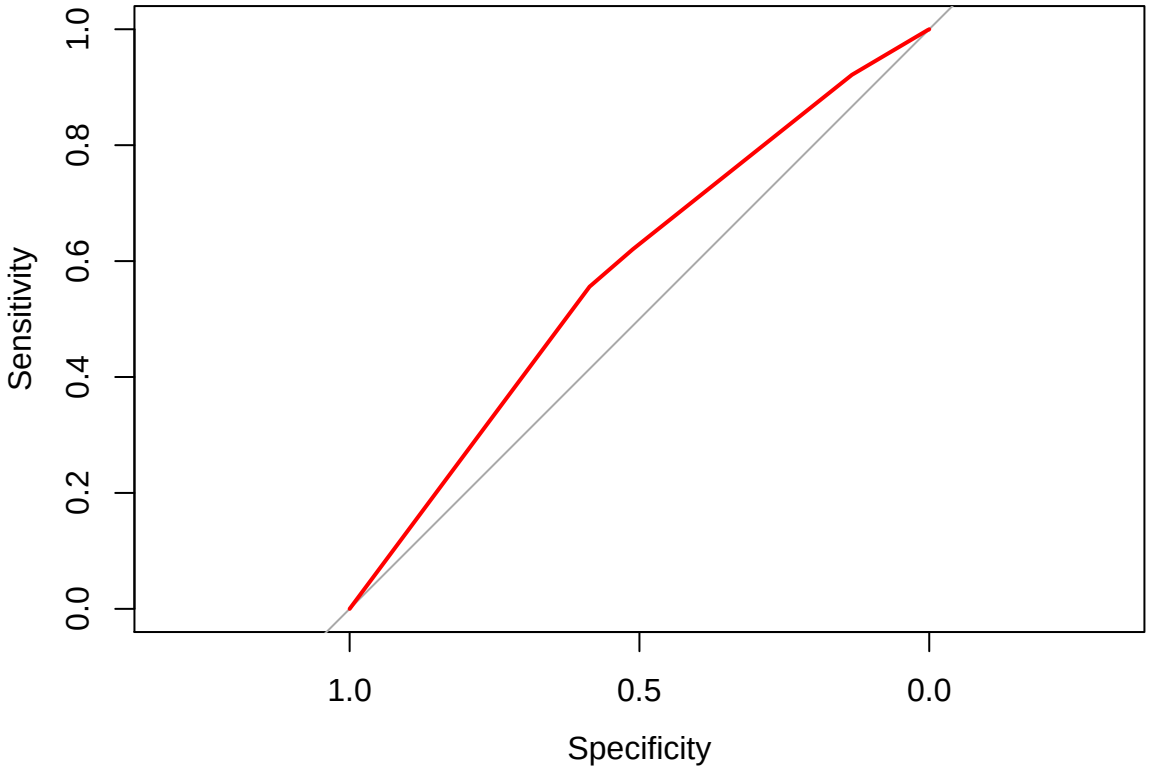
Donors that haven given both recently and frequently may be especially likely to give again; in other words, the combined impact of recency and frequency may be greater than the sum of the separate effects. In Marketing terms this is known as the RFM Model (recency, freqeuncy and money).

Because these predictors together have a greater impact on the dependent variable, their joint effect must be modeled as an interaction.

```
# Build a recency, frequency, and money (RFM) model
rfm_model <- glm(donated ~ money + recency * frequency, data = donors, family = "binomial")

# Summarize the RFM model to see how the parameters were coded
summary(rfm_model)
```

```
##
## Call:
## glm(formula = donated ~ money + recency * frequency, family = "binomial",
##     data = donors)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.3696  -0.3696  -0.2895  -0.2895   2.7924
##
## Coefficients:
##                                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)                       -3.01142    0.04279 -70.375   <2e-16 ***
## moneyMEDIUM                        0.36186    0.04300   8.415   <2e-16 ***
## recencyLAPSED                     -0.86677    0.41434  -2.092   0.0364 *
## frequencyINFREQUENT               -0.50148    0.03107 -16.143   <2e-16 ***
## recencyLAPSED:frequencyINFREQUENT  1.01787    0.51713   1.968   0.0490 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 37330  on 93461  degrees of freedom
## Residual deviance: 36938  on 93457  degrees of freedom
## AIC: 36948
##
## Number of Fisher Scoring iterations: 6
```

```
# Compute predicted probabilities for the RFM model
rfm_prob <- predict(rfm_model, type = "response")

# Plot the ROC curve and find AUC for the new model
library(pROC)
ROC <- roc(donors$donated, rfm_prob)
plot(ROC, col = "red")
```




```
auc(ROC)
```

```
## Area under the curve: 0.5785
```

2.3.2 Automatic feature selection

So far we have been selecting which variables to put in our model manually ourselves. We've had to logically build our model using some fundraising knowledge, thinking about what variables may influence donations. There is a process to speed this up using automatic feature selection called stepwise feature selection.

Stepwise feature selection builds a model each variable at a time and sees which predictor adds value to the final model. Backward deletion begins with a model containing all the features, then variables are deleted as long as the removal of a variable does not negatively impact the models overall ability to predict the outcome. At each step, the predictor that impacts the model the least is removed.

Forward stepwise selection does this in reverse, starting with no predictors, then adding in each predictor by first assessing each predictor to determine which has the greatest ability to predict the outcome at each step and adding that predictor, then re-assessing all remaining predictors.

The results of a forward and backward stepwise process can result in different results AND neither may be the best possible model. This process coul be accused asbeing akin to p-hacking and violates some of our science principles, namely to generate then test a hypothesis - it is atheortical as there is no theory about how things work. But if the goal is prediction, this might not be such a great concern. But it might be best to consider stepwise regression as a tool to start from.

To run stepwise, we set our null model, then our full model, then use step to iteratie either forward or backward between these models.

```
# Specify a null model with no predictors
null_model <- glm(donated ~ 1, data = donors, family = "binomial")

# Specify the full model using all of the potential predictors
full_model <- glm(donated ~ ., data = donors, family = "binomial")

# Use a forward stepwise algorithm to build a parsimonious model
step_model <- step(null_model, scope = list(lower = null_model, upper = full_model), direction = "forwa
```

```
## Start:  AIC=37332.13
## donated ~ 1
```

```
## Warning in add1.glm(fit, scope$add, scale = scale, trace = trace, k = k, :
## using the 70916/93462 rows from a combined fit
```

```
##                     Df Deviance   AIC
## + frequency          1    28502 37122
## + money              1    28621 37241
## + has_children       1    28705 37326
## + age                1    28707 37328
## + imputed_age        1    28707 37328
## + wealth_rating      3    28704 37328
## + interest_veterans  1    28709 37330
## + donation_prob      1    28710 37330
## + donation_pred      1    28710 37330
## + catalog_shopper    1    28710 37330
## + pet_owner          1    28711 37331
## <none>                    28714 37332
## + interest_religion  1    28712 37333
## + recency            1    28713 37333
## + bad_address        1    28714 37334
## + veteran            1    28714 37334
##
## Step:  AIC=37024.77
## donated ~ frequency
```

```
## Warning in add1.glm(fit, scope$add, scale = scale, trace = trace, k = k, :
## using the 70916/93462 rows from a combined fit
```

```
##                     Df Deviance   AIC
## + money              1    28441 36966
## + wealth_rating      3    28490 37019
## + has_children       1    28494 37019
## + donation_prob      1    28498 37023
## + interest_veterans  1    28498 37023
## + catalog_shopper    1    28499 37024
## + donation_pred      1    28499 37024
## + age                1    28499 37024
## + imputed_age        1    28499 37024
## + pet_owner          1    28499 37024
## <none>                    28502 37025
## + interest_religion  1    28501 37026
## + recency            1    28501 37026
## + bad_address        1    28502 37026
## + veteran            1    28502 37027
##
```

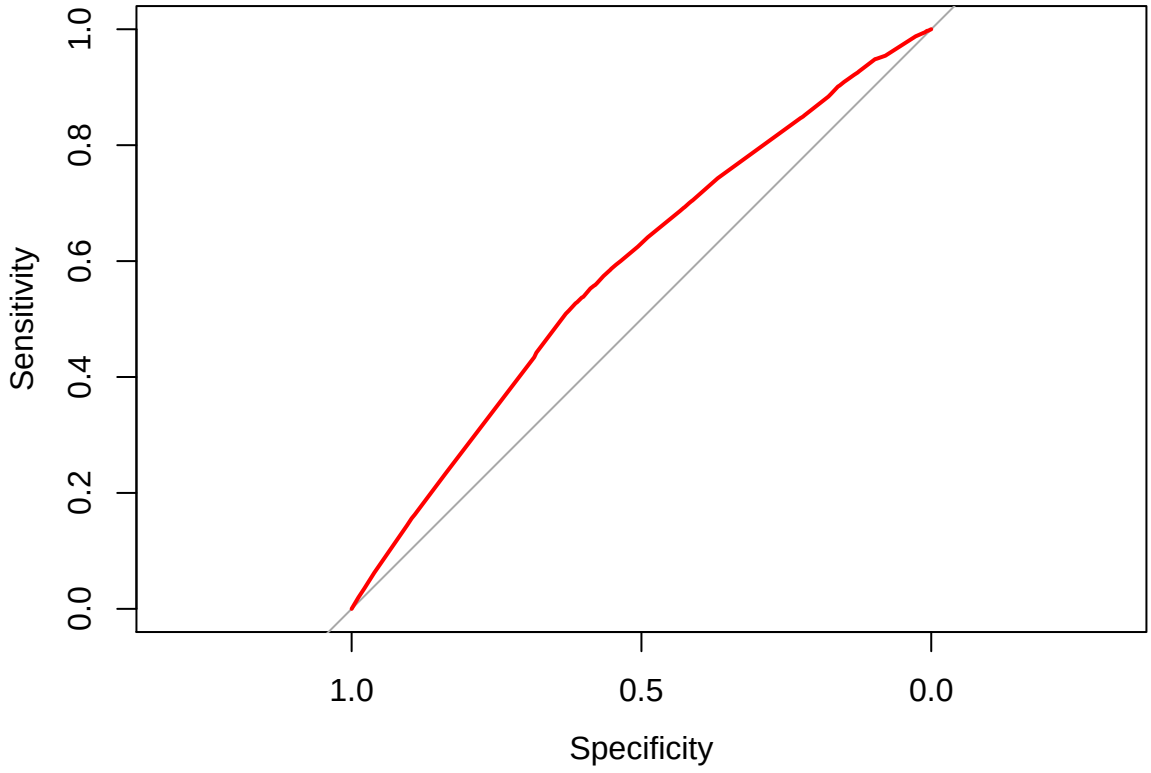
```
## Step:  AIC=36949.71
## donated ~ frequency + money

## Warning in add1.glm(fit, scope$add, scale = scale, trace = trace, k = k, :
## using the 70916/93462 rows from a combined fit

##                     Df Deviance   AIC
## + wealth_rating      3    28427 36942
## + has_children       1    28432 36943
## + interest_veterans  1    28438 36948
## + donation_prob      1    28438 36949
## + catalog_shopper    1    28438 36949
## + donation_pred      1    28439 36949
## + age                1    28439 36949
## + imputed_age        1    28439 36949
## + pet_owner          1    28439 36949
## <none>                    28441 36950
## + interest_religion  1    28440 36951
## + recency            1    28441 36951
## + bad_address        1    28441 36951
## + veteran            1    28441 36952
##
## Step:  AIC=36945.48
## donated ~ frequency + money + wealth_rating

## Warning in add1.glm(fit, scope$add, scale = scale, trace = trace, k = k, :
## using the 70916/93462 rows from a combined fit

##                     Df Deviance   AIC
## + has_children       1    28416 36937
## + age                1    28424 36944
## + imputed_age        1    28424 36944
## + interest_veterans  1    28424 36945
## + donation_prob      1    28424 36945
## + catalog_shopper    1    28425 36945
## + donation_pred      1    28425 36945
## <none>                    28427 36945
## + pet_owner          1    28425 36946
## + interest_religion  1    28426 36947
## + recency            1    28427 36947
## + bad_address        1    28427 36947
## + veteran            1    28427 36947
##
## Step:  AIC=36938.4
## donated ~ frequency + money + wealth_rating + has_children

## Warning in add1.glm(fit, scope$add, scale = scale, trace = trace, k = k, :
## using the 70916/93462 rows from a combined fit

##                     Df Deviance   AIC
## + pet_owner          1    28413 36937
## + donation_prob      1    28413 36937
## + catalog_shopper    1    28413 36937
## + interest_veterans  1    28413 36937
## + donation_pred      1    28414 36938
## <none>                    28416 36938
## + interest_religion  1    28415 36939
```

```
## + age                1    28416 36940
## + imputed_age        1    28416 36940
## + recency            1    28416 36940
## + bad_address        1    28416 36940
## + veteran            1    28416 36940
##
## Step:  AIC=36932.25
## donated ~ frequency + money + wealth_rating + has_children +
##     pet_owner
```

```
## Warning in add1.glm(fit, scope$add, scale = scale, trace = trace, k = k, :
## using the 70916/93462 rows from a combined fit
```

```
##                      Df Deviance   AIC
## <none>                    28413 36932
## + donation_prob      1    28411 36932
## + interest_veterans  1    28411 36932
## + catalog_shopper    1    28412 36933
## + donation_pred      1    28412 36933
## + age                1    28412 36933
## + imputed_age        1    28412 36933
## + recency            1    28413 36934
## + interest_religion  1    28413 36934
## + bad_address        1    28413 36934
## + veteran            1    28413 36934
```

```
# Estimate the stepwise donation probability
step_prob <- predict(step_model, type = "response")

# Plot the ROC of the stepwise model
library(pROC)
ROC <- roc(donors$donated, step_prob)
plot(ROC, col = "red")
```




```
auc(ROC)
```

```
## Area under the curve: 0.5849
```

It is perhaps useful when we lack subject matter expertise and wish to try and find the or some of the most important features which help to predict the outcome of interest.

2.4 Classification Trees

Also known as decisions trees break down data in to a serious of steps or questions (if else) then help to define action. The goal is to model predictors against an outcome of interest. If someone is applying for a loan, we can use past data to determine and build a tree that helps to determine how probable it is that new applicant will repay the debt.

One of those most common R packages is rpart where the part stands for recursive partitioning. There is a good description of how this works in practice in the Introduction to Tree Based Methods from ISL.

The loans dataset contains 11,312 randomly-selected people who were applied for and later received loans from Lending Club, a US-based peer-to-peer lending company.

You will use a decision tree to try to learn patterns in the outcome of these loans (either repaid or default) based on the requested loan amount and credit score at the time of application.

First let's prepare the data.

```
#Load the loans data and subset to just those randomly selected rows
loans_full <-  read.csv("./files/SLinRClassification/loans.csv", stringsAsFactors = TRUE)
loans <- subset(loans_full, keep == 1)
```

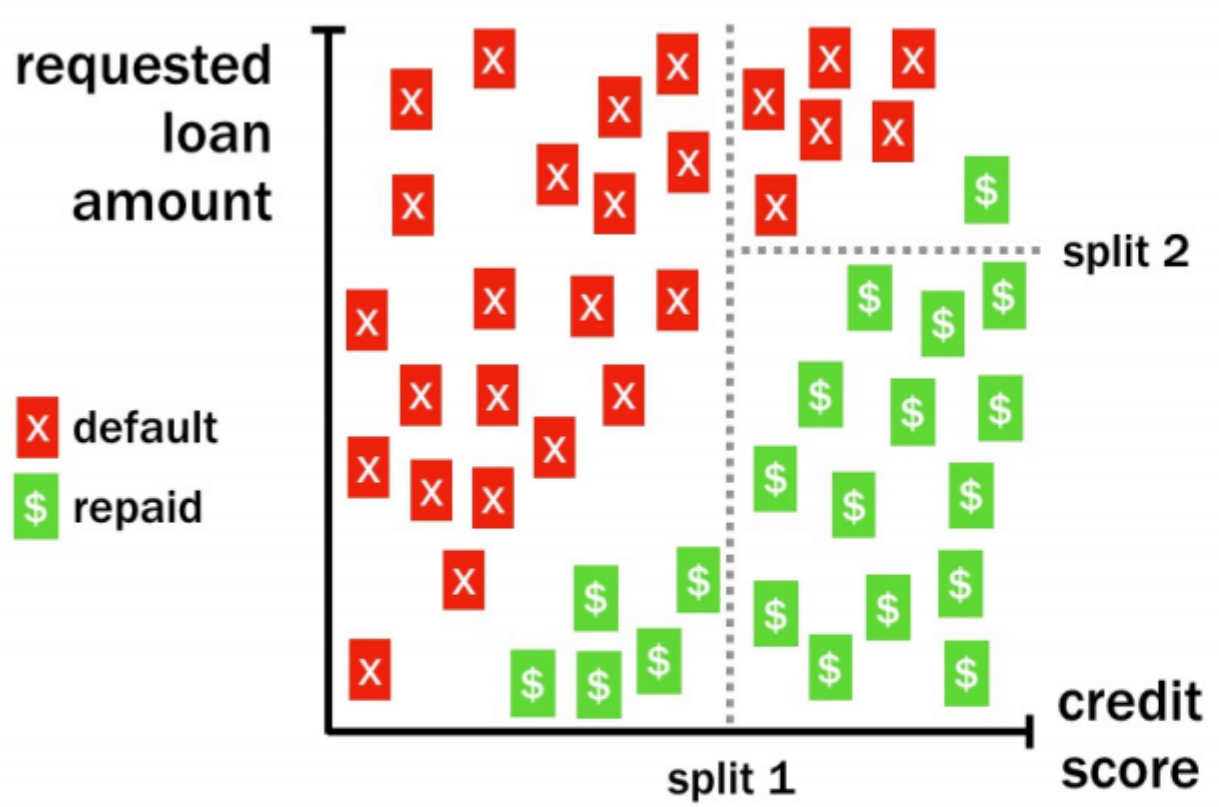


Figure 2.1: Decision Tree

```
# Record default to our variable of interest outcome
loans$outcome[loans$default=="0"] <- "2"
loans$outcome[loans$default=="1"] <- "1"

# Convert the column to a factor and provide names for the levels
loans$outcome <- factor(loans$outcome)
levels(loans$outcome) <- c("default", "repaid")

# Remove the unneccessary columns
loans <- loans[-c(1:3)]
```

Next we do the modelling.

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

# Build a lending model predicting loan outcome versus loan amount and credit score
loan_model <- rpart(outcome ~ loan_amount + credit_score, data = loans, method = "class", control = rpa

# Make a prediction for someone with good credit
# predict(loan_model, good_credit, type = "class")

# Make a prediction for someone with bad credit
# predict(loan_model, bad_credit, type = "class")
```

The structure of classification trees can be depicted visually, which helps to understand how the tree makes its decisions. This might be useful for Transparency reasons.

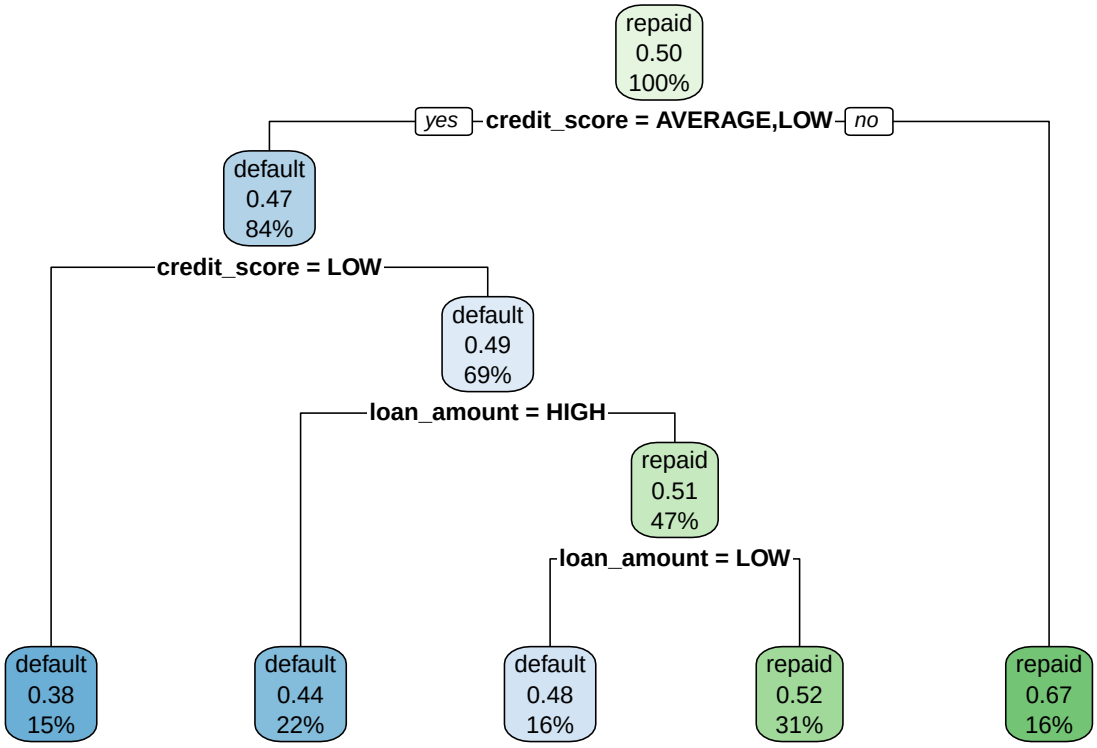
```
# Examine the loan_model object
loan_model
```

```
## n= 11312
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 11312 5654 repaid (0.4998232 0.5001768)
##    2) credit_score=AVERAGE,LOW 9490 4437 default (0.5324552 0.4675448)
##      4) credit_score=LOW 1667  631 default (0.6214757 0.3785243) *
##      5) credit_score=AVERAGE 7823 3806 default (0.5134859 0.4865141)
##       10) loan_amount=HIGH 2472 1079 default (0.5635113 0.4364887) *
##       11) loan_amount=LOW,MEDIUM 5351 2624 repaid (0.4903756 0.5096244)
##         22) loan_amount=LOW 1810  874 default (0.5171271 0.4828729) *
##         23) loan_amount=MEDIUM 3541 1688 repaid (0.4767015 0.5232985) *
##    3) credit_score=HIGH 1822  601 repaid (0.3298573 0.6701427) *
```

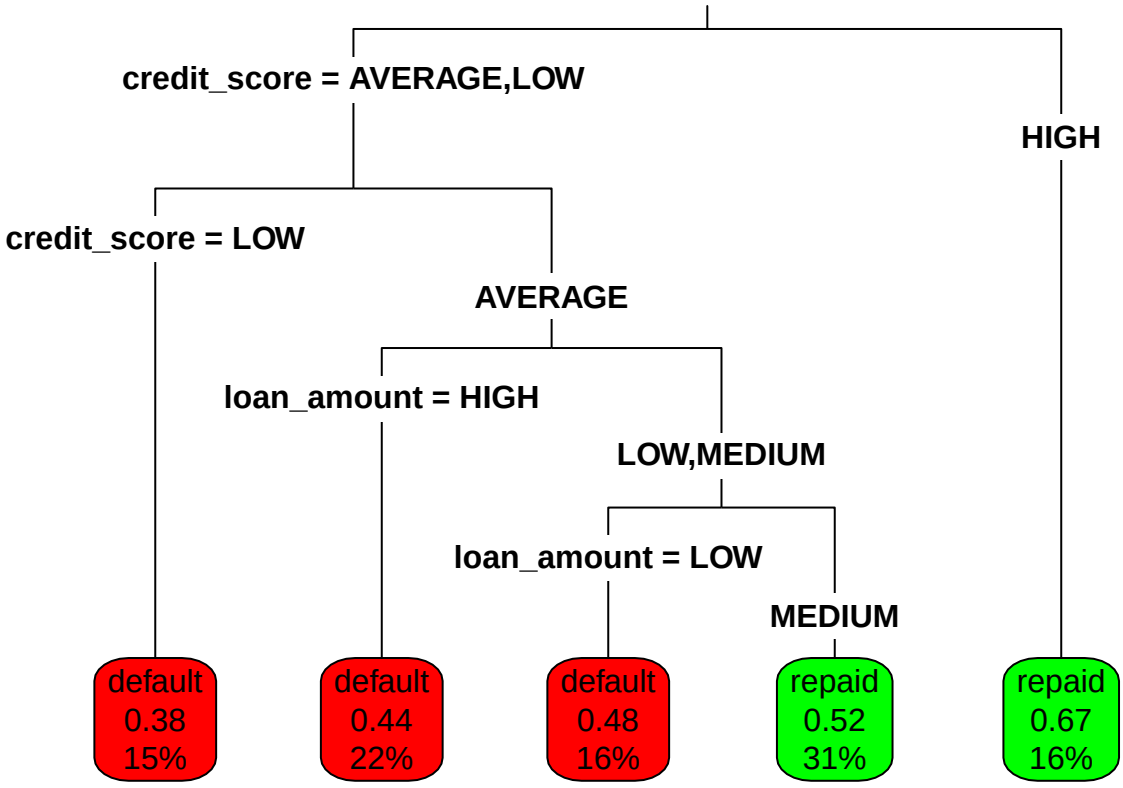
```
# Load the rpart.plot package
library(rpart.plot)
```

```
## Warning: package 'rpart.plot' was built under R version 3.4.4
```

```
# Plot the loan_model with default settings
rpart.plot(loan_model)
```




```
# Plot the loan_model with customized settings
rpart.plot(loan_model, type = 3, box.palette = c("red", "green"), fallen.leaves = TRUE)
```



When choosing between different split options - for instance choosing whether to split on loan amount or credit score - the decision tree will provide a split for both, then look and how homogenous the resulting options are. In the diagram below, even though split B (based on loan amount) produces a very similar group for one partition (14/15 defaulted) it is much more mixed for the other partition (19/32 defaulted). In comparison, split A is more pure, so will be choosen first. It will then proceed to look at the next split which results in the most homogenous/similar partition.

As the tree begins to grow, it results in smaller and more homogeous partions (below left). An easy option would be to draw a diagonal line (below right) however these requires, in this example, consideration of two different variables, which is not possible with the 'divide and conquer' process. A decision tree creates what are called axis parallel splits, which can mean for some patterns in data they can become overly complex.

Decision trees can divide and conquer until it either runs out of features or cases to classify, which can result in large trees which are over-fitted. So we can split out a test set for evaluation - we create a hold out group or set. The sample() function can be used to generate a random sample of rows to include in the training set. Simply supply it the total number of observations and the number needed for training. We use the resulting vector of row IDs to subset the loans into training and testing datasets.

```
# Determine the number of rows for training
train <- nrow(loans) * .75

# Create a random sample of row IDs
sample_rows <- sample(nrow(loans), train)

# Create the training dataset
loans_train <- loans[sample_rows,]

# Create the test dataset
```

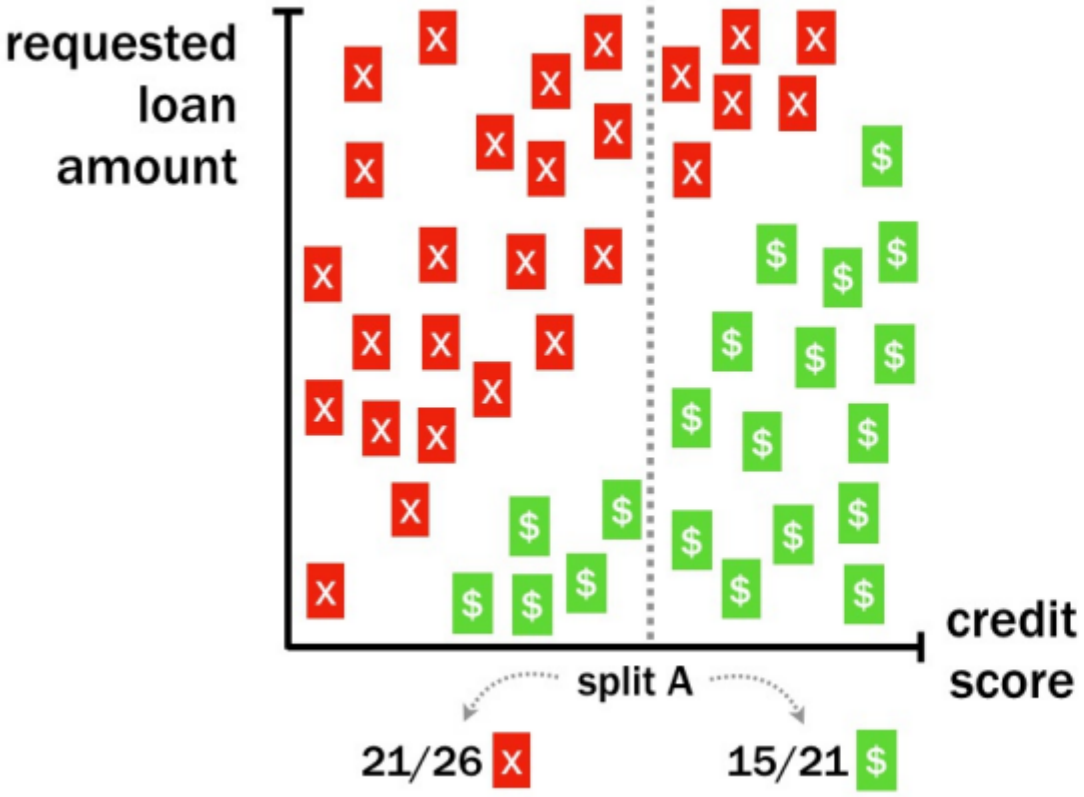


Figure 2.2: Tree Split Choice

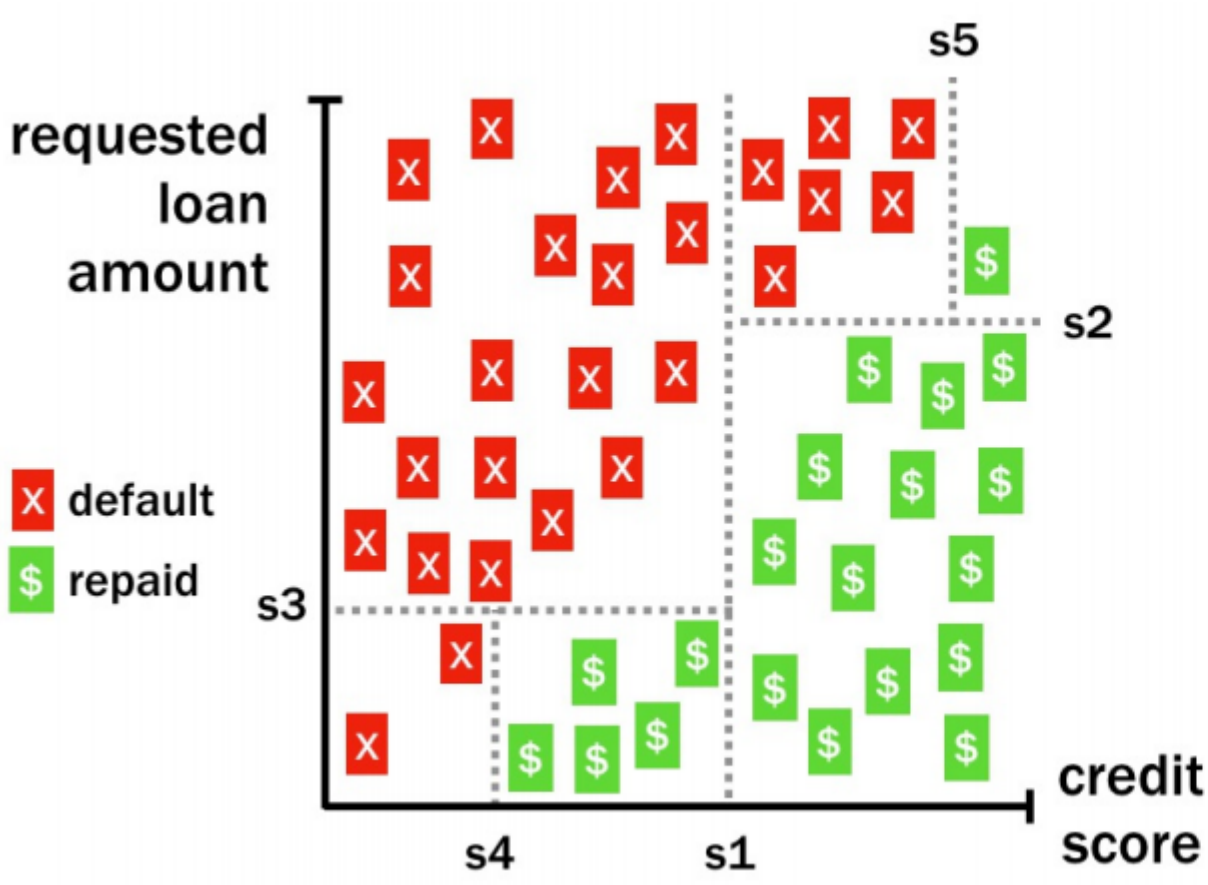


Figure 2.3: Axis parallel split

(#fig:Axis Splits)

```
loans_test <- loans[-sample_rows,]
```

So far we have onlt built our model based on a couple of the available variables. Next we can use all of the available applicant data to build a more sophisticated lending model using the random training dataset created previously. Then, we use this model to make predictions on the testing dataset to estimate the performance of the model on future loan applications.

```
# Grow a tree using all of the available applicant data
loan_model <- rpart(outcome ~ ., data = loans_train, method = "class", control = rpart.control(cp = 0))

# Make predictions on the test dataset
loans_test$pred <- predict(loan_model, loans_test, type = "class")

# Examine the confusion matrix
table(loans_test$pred, loans_test$outcome)
```

```
##
##           default repaid
##   default     805    579
##   repaid      635    809
```

```
# Compute the accuracy on the test dataset
mean(loans_test$pred == loans_test$outcome)
```

```
## [1] 0.5707214
```

2.4.1 Tending to classification trees

As classification trees can overfit, we may need to prune the trees. One method of achieving this is stopping the growing method early by limiting the tree depth and is known as pre-pruning. e.g. we have a max tree depth of 7 levels or branches. Another option is to set the minimum number of observations that can occur in any one branch, perhaps the tree if stopped if there are few than 10 observations on any one branch. However, trees stopped early may miss some patterns that might have been discovered later.

An alternative is to post-pruning where a complex, over-fitted tree is built first, then pruned back to reduce the size. After the tree is built we remove nodes or branches that have little impact on the overall accuracy. To do this we can plot the degree of error reduction on the vertical axis against the tree depth (aka complexity) as shown below and look for a 'dog leg' or kink in the curve. As the tree grows, each successive branch or node improves the accuracy quite a lot, however later branches or nodes only improve the accuracy by a smaller amount. So we can try and find the point at which the curve flattens.

The rpart packages provides this chart and options for pre and post pruning. The pre-pruning is done using the rpart.control function which is then passed to the model as the control. For post pruning, we create the model then use the plotcp(model) to identify the error rate to tree depth trade off, then use the prune(model, cp = n) function to prune the tree, when cp is the complexity parameter.

In the example below, we apply pre-pruning to our applicant data, first by setting the max tree depth to 6, then by setting the min. number of observations at a node to 500.

```
# Grow a tree with maxdepth of 6
loan_model <- rpart(outcome ~ ., data = loans_train, method = "class", control = rpart.control(maxdepth

# Compute the accuracy of the simpler tree
loans_test$pred <- predict(loan_model, loans_test, type = "class")
mean(loans_test$pred == loans_test$outcome)
```

```
## [1] 0.5823904
```

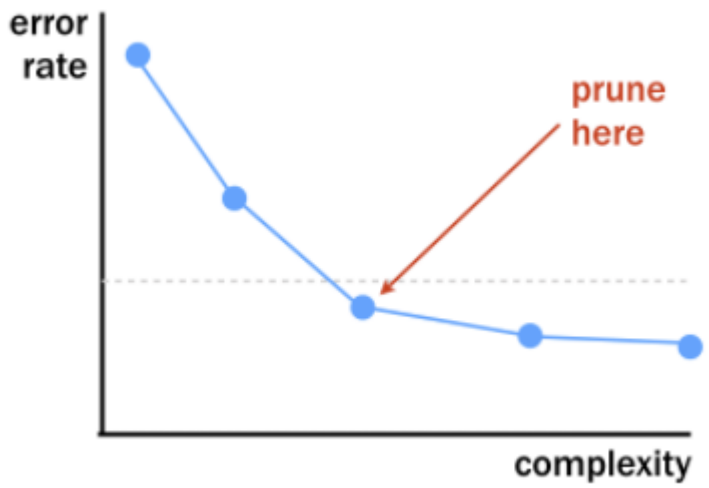


Figure 2.4: Dog Leg

(#fig:Dog Leg)

```
# Grow a tree with minsplit of 500
loan_model2 <- rpart(outcome ~ ., data = loans_train, method = "class", control = rpart.control(minsplit
# Compute the accuracy of the simpler tree
loans_test$pred2 <- predict(loan_model2, loans_test, type = "class")
mean(loans_test$pred2 == loans_test$outcome)
```

```
## [1] 0.5862801
```

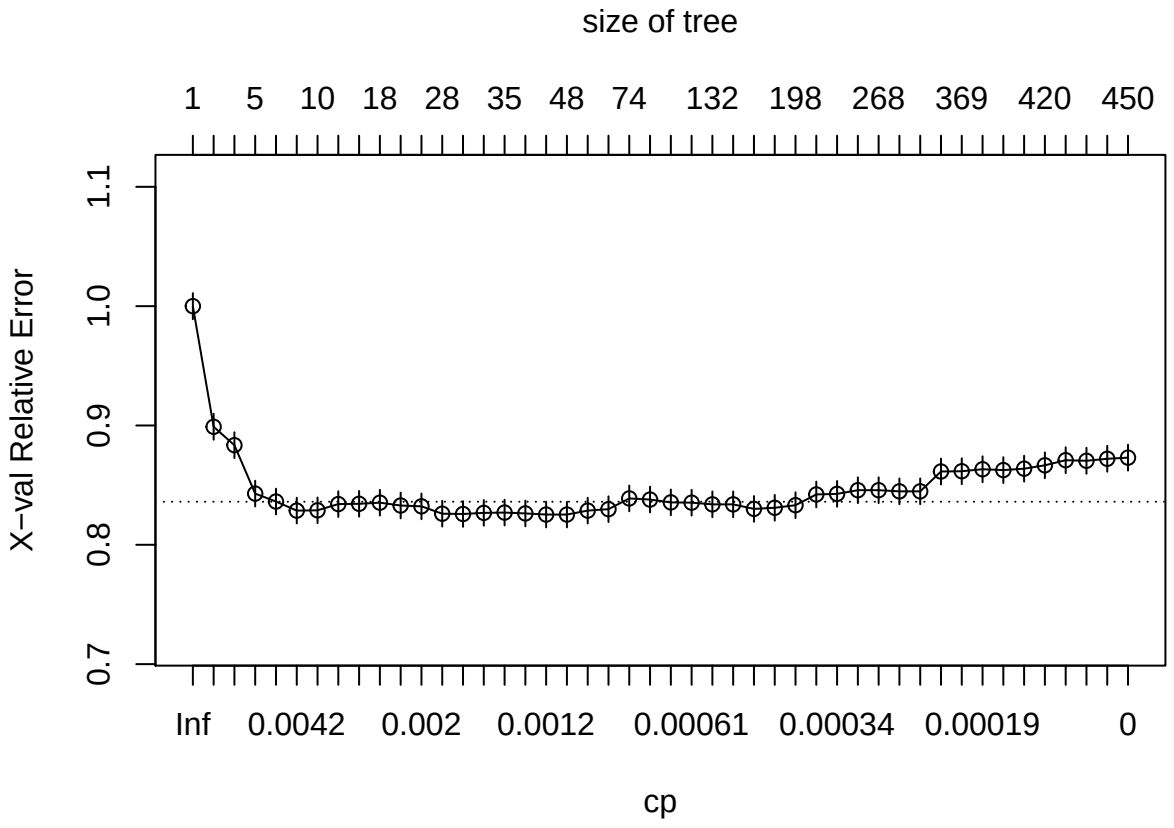
In both these cases, we see the mean accuracy on the test data, despite fitting a simpler tree, is actually higher than the unpruned tree.

Next we use post-pruning and see that this too improves the predictive accuracy on the test data, despite being a simpler tree.

```
# Grow an overly complex tree
loan_model <- rpart(outcome ~ ., data = loans_train, method = "class", control = rpart.control(cp = 0))
# Compute the accuracy of the unpruned tree
loans_test$pred1 <- predict(loan_model, loans_test, type = "class")
mean(loans_test$pred1 == loans_test$outcome)
```

```
## [1] 0.5707214
```

```
# Examine the complexity plot
plotcp(loan_model)
```




```
# Prune the tree
loan_model_pruned <- prune(loan_model, cp = 0.0014)

# Compute the accuracy of the pruned tree
loans_test$pred <- predict(loan_model_pruned, loans_test, type = "class")
mean(loans_test$pred == loans_test$outcome)
```

```
## [1] 0.5997171
```

A number of classification trees can be combined together to create a forest of classification trees. Each of the trees is diverse but simple and by combining them together we can help to udnerstand the complexity in the underlying data. But growing different trees requires differing conditions for each tree, otherwise growing 100 trees on the same data would result in 100 identical trees. To do this, we allocate each tree a random subset of data, we do this using the random forest approach. So each tree is give a small random sample which grows a simple tree, and is then combine. This can seem counter intuitive, since we might think havign a single complex tree is more accurate. A bit like an effective team, it is better to have specialised skills. Combining multiple learners together is known as ensemble models, where each tree or model is given a vote on a particular observation. The teamwork-based approach of the random forest may help it find important trends a single tree may miss.

We use the randomForest package in R and specify the ntree parameter for the size of the forest and mtry is the number of features selected at random for each tree. We can typically use the default number of features which is sqrt(p) where p is the total number of parameters.

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

```
## randomForest 4.6-12
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
# Build a random forest model, first with default values and second with 500 trees and each tree with s
loan_model_rf <- randomForest(outcome ~ ., data = loans_train)
loan_model_rf2 <- randomForest(outcome ~ ., data = loans, ntree = 500, mtry = round(sqrt(ncol(loans)-1))

# Compute the accuracy of the random forest
loans_test$pred <- predict(loan_model_rf, loans_test, type = "class")
loans_test$pred2 <- predict(loan_model_rf2, loans_test, type = "class")
mean(loans_test$pred == loans_test$outcome)
```

```
## [1] 0.594413
```

```
mean(loans_test$pred2 == loans_test$outcome)
```

```
## [1] 0.9105375
```

Chapter 3

Supervised Learning In R Regression

Notes taken during/inspired by the DataCamp course 'Supervised Learning In R Classification' by Nina Zumel and John Mount.

Course Handouts

- Part 1 - What is Regression?
- [Part 2 - Training and Evaluating Regression Models] (./files/SLinRRegression/chapter2.pdf)
- [Part 3 - Issues to Consider] (./files/SLinRRegression/chapter3.pdf)
- [Part 4 - Dealing with Non-Linear Responses] (./files/SLinRRegression/chapter4.pdf)
- [Part 5 - Tree-Based Methods] (./files/SLinRRegression/chapter5.pdf)

Other useful links

- The Basics of Encoding Categorical Data for Predictive Models

3.1 What is Regression?

In this course we are interested in predicting numerical outcomes (rather than discreet) from a set of input variables. Classification on the other hand is the task of making discreet predictions. WE can break down the modelling process in to two camps:

- Scientific mindset - we try to understand causal mechanisms
- Engineering mindset - we try to accuratley predict

Machine Learning is more in line with the Engineering mindset.

In the first task, the goal is to predict the rate of female unemployment from the observed rate of male unemployment. The outcome is female_unemployment, and the input is male_unemployment.

```
# Load the unemployment data
unemployment <- readRDS("./files/SLinRRegression/unemployment.rds")

# unemployment is loaded in the workspace
summary(unemployment)
```

```
##  male_unemployment female_unemployment
##  Min.   :2.900     Min.   :4.000
##  1st Qu.:4.900     1st Qu.:4.400
##  Median :6.000     Median :5.200
```

```
##  Mean   :5.954     Mean   :5.569
##  3rd Qu.:6.700     3rd Qu.:6.100
##  Max.   :9.800     Max.   :7.900
```

```
# Define a formula to express female_unemployment as a function of male_unemployment
fmla <- as.formula("female_unemployment ~ male_unemployment")

# Print it
fmla
```

```
## female_unemployment ~ male_unemployment
```

```
# Use the formula to fit a model: unemployment_model
unemployment_model <- lm(formula = fmla, data = unemployment)

# Print it
unemployment_model
```

```
##
## Call:
## lm(formula = fmla, data = unemployment)
##
## Coefficients:
##       (Intercept)  male_unemployment
##            1.4341             0.6945
```

We can see the relationship is positive - female unemployment increases as male unemployment does. We can then use some diagnostics on the model. There are different ways and packages to achieve this.

```
# Print unemployment_model
unemployment_model
```

```
##
## Call:
## lm(formula = fmla, data = unemployment)
##
## Coefficients:
##       (Intercept)  male_unemployment
##            1.4341             0.6945
```

```
# Call summary() on unemployment_model to get more details
summary(unemployment_model)
```

```
##
## Call:
## lm(formula = fmla, data = unemployment)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.77621 -0.34050 -0.09004  0.27911  1.31254
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)        1.43411    0.60340   2.377   0.0367 *
## male_unemployment  0.69453    0.09767   7.111 1.97e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 0.5803 on 11 degrees of freedom
## Multiple R-squared:  0.8213, Adjusted R-squared:  0.8051
## F-statistic: 50.56 on 1 and 11 DF,  p-value: 1.966e-05
```

```
# Call glance() on unemployment_model to see the details in a tidier form
broom::glance(unemployment_model)
```

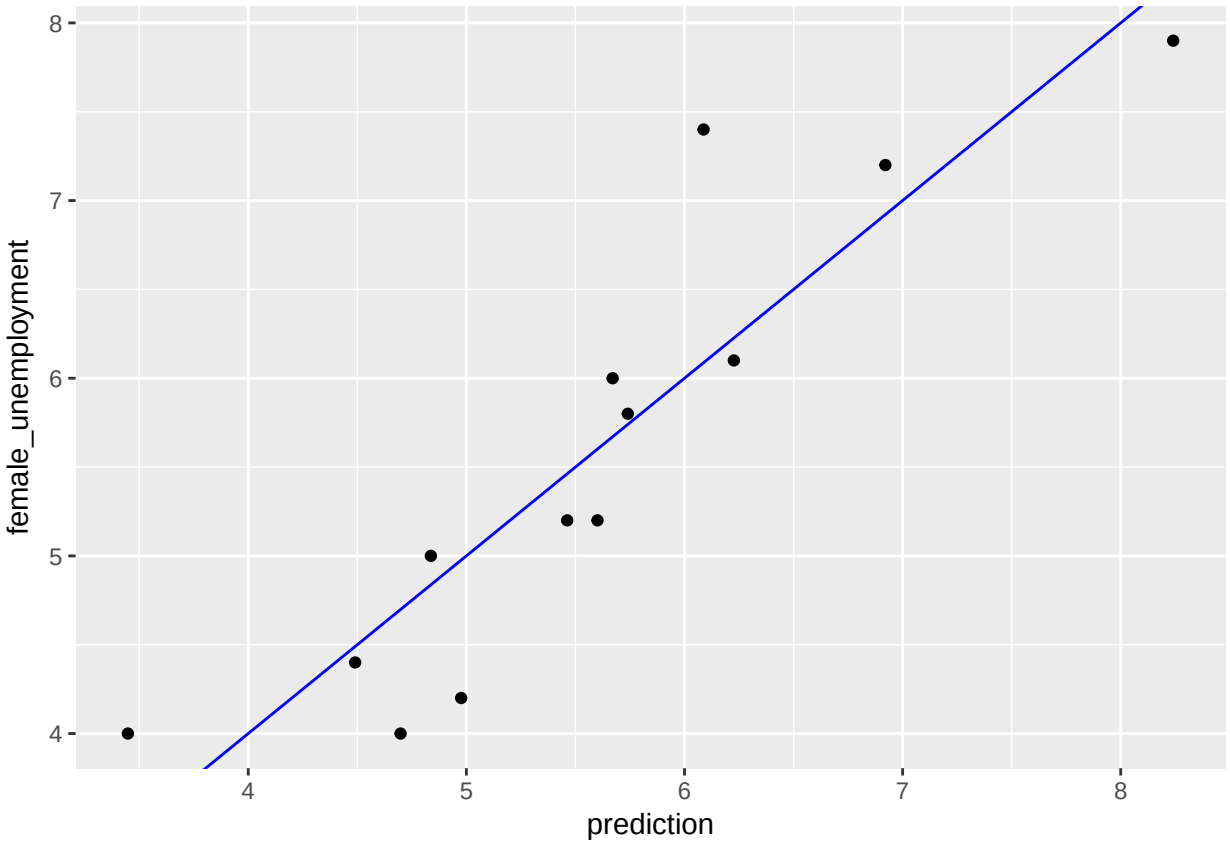
```
##   r.squared adj.r.squared     sigma statistic      p.value df    logLik
## 1 0.8213157     0.8050716 0.5802596  50.56108 1.965985e-05  2 -10.28471
##        AIC      BIC deviance df.residual
## 1 26.56943 28.26428 3.703714          11
```

```
# Call wrapFTest() on unemployment_model to see the most relevant details
sigr::wrapFTest(unemployment_model)
```

```
## [1] "F Test summary: (R2=0.821, F(1,11)=50.6, p=2e-05)."
```

In the next section we use the unemployment model unemployment_model to make predictions from the unemployment data, and compare predicted female unemployment rates to the actual observed female unemployment rates on the training data, unemployment. You will also use your model to predict on the new data in newrates.

```
newrates <- data.frame(male_unemployment = 5)

# Predict female unemployment in the unemployment data set
unemployment$prediction <-  predict(unemployment_model, unemployment)

# load the ggplot2 package
library(ggplot2)

# Make a plot to compare predictions to actual (prediction on x axis).
ggplot(unemployment, aes(x = prediction, y = female_unemployment)) +
  geom_point() +
  geom_abline(color = "blue")
```




```
# Predict female unemployment rate when male unemployment is 5%
pred <- predict(unemployment_model, newrates)
# Print it
pred
```

```
##        1
## 4.906757
```

Next we will look at a model of blood pressure as a function of age and weight.

```
# Load the blood data
bloodpressure <- readRDS("./files/SLinRRegression/bloodpressure.rds")

# bloodpressure is in the workspace
summary(bloodpressure)
```

```
##  blood_pressure       age             weight
##  Min.   :128.0   Min.   :46.00   Min.   :167
##  1st Qu.:140.0   1st Qu.:56.50   1st Qu.:186
##  Median :153.0   Median :64.00   Median :194
##  Mean   :150.1   Mean   :62.45   Mean   :195
##  3rd Qu.:160.5   3rd Qu.:69.50   3rd Qu.:209
##  Max.   :168.0   Max.   :74.00   Max.   :220
```

```
# Create the formula and print it
fmla <- as.formula("blood_pressure ~ age + weight")
fmla
```

```
## blood_pressure ~ age + weight
```

```
# Fit the model: bloodpressure_model
bloodpressure_model <- lm(fmla, bloodpressure)

# Print bloodpressure_model and call summary()
bloodpressure_model
```

```
##
## Call:
## lm(formula = fmla, data = bloodpressure)
##
## Coefficients:
## (Intercept)          age       weight
##     30.9941       0.8614       0.3349
```

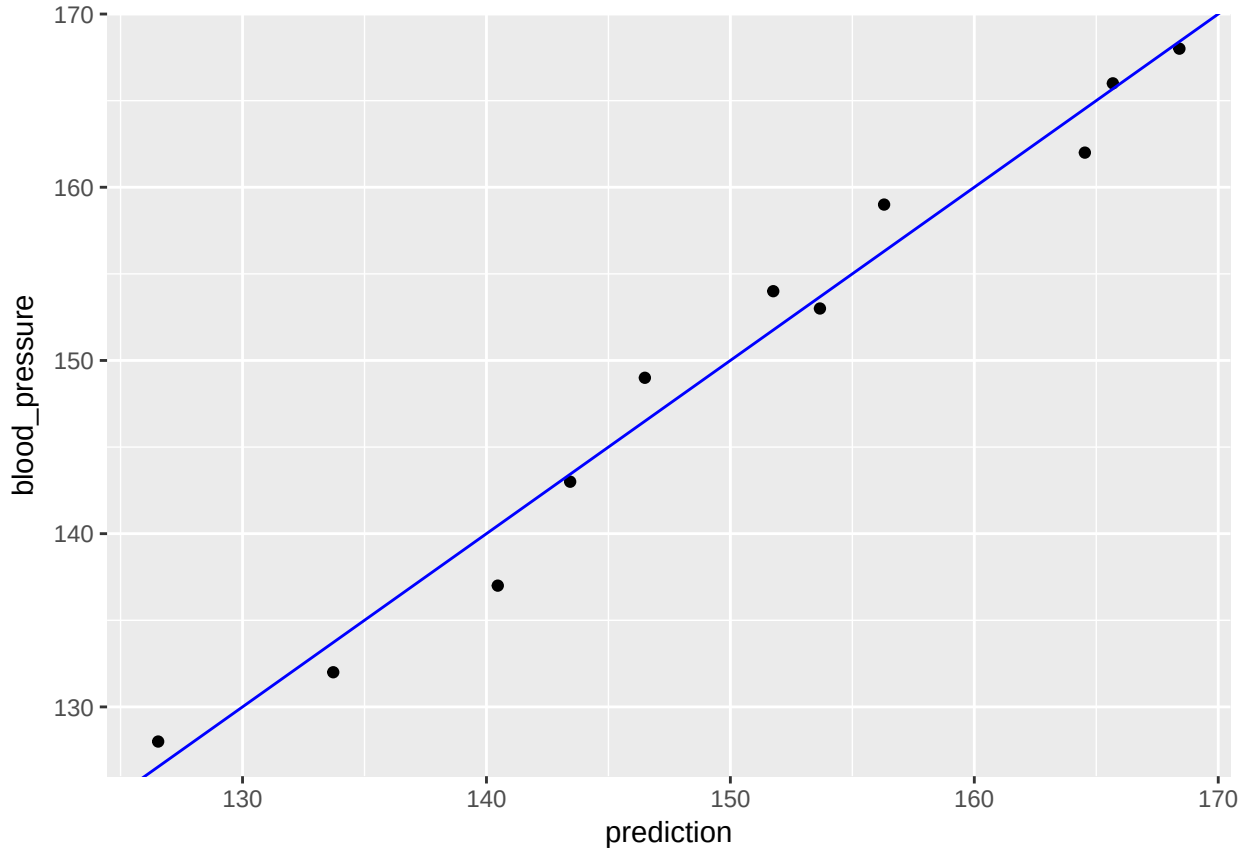
```
summary(bloodpressure_model)
```

```
##
## Call:
## lm(formula = fmla, data = bloodpressure)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.4640 -1.1949 -0.4078  1.8511  2.6981
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  30.9941    11.9438   2.595  0.03186 *
## age           0.8614     0.2482   3.470  0.00844 **
## weight        0.3349     0.1307   2.563  0.03351 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.318 on 8 degrees of freedom
## Multiple R-squared:  0.9768, Adjusted R-squared:  0.9711
## F-statistic: 168.8 on 2 and 8 DF,  p-value: 2.874e-07
```

Both variables are positive suggesting that increases in them also lead to increases in blood pressure.

Next we can use the model to make preditions.

```
# predict blood pressure using bloodpressure_model :prediction
bloodpressure$prediction <- predict(bloodpressure_model, bloodpressure)

# plot the results
ggplot(bloodpressure, aes(x = prediction, y = blood_pressure)) +
    geom_point() +
    geom_abline(color = "blue")
```



** Key Points **

- Linear models are easy to fit and can be less prone to overfitting.

- They are also generally easier to understand
- However they cannot express complex relationships in the data
- In addition we can get collinearity with some variables e.g. weight increases with age

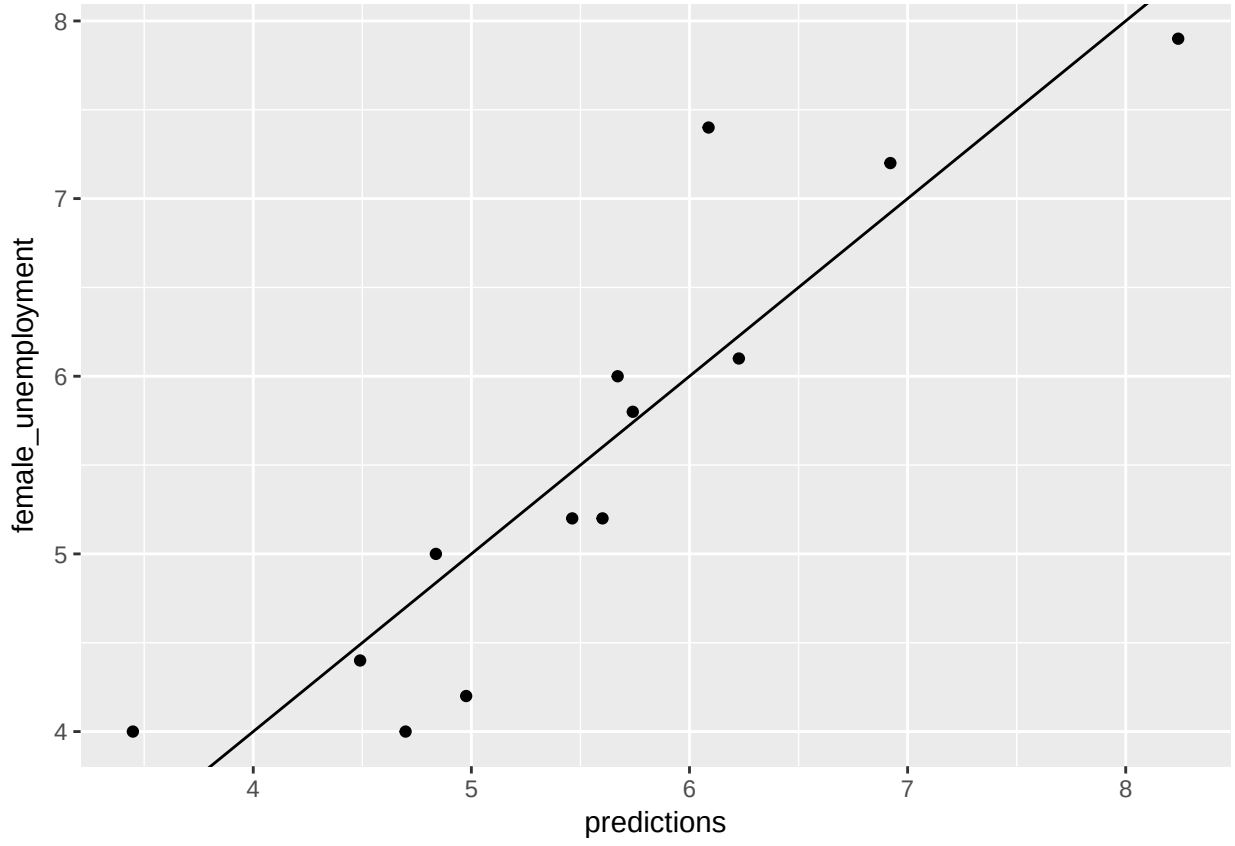
3.2 Training and Evaluating Regression Models

A model with a good fit will have its points close to the line. Models which don't fit well will have points which systemtically don't fit welland are all over the place. This may mean you are missing variables in your model. A residual plot can help understand if there are any systematic errors, we are typically looking for a random cloud of residuals rather than anything which may resemble a trend. We can also use a Gain Curve Plot

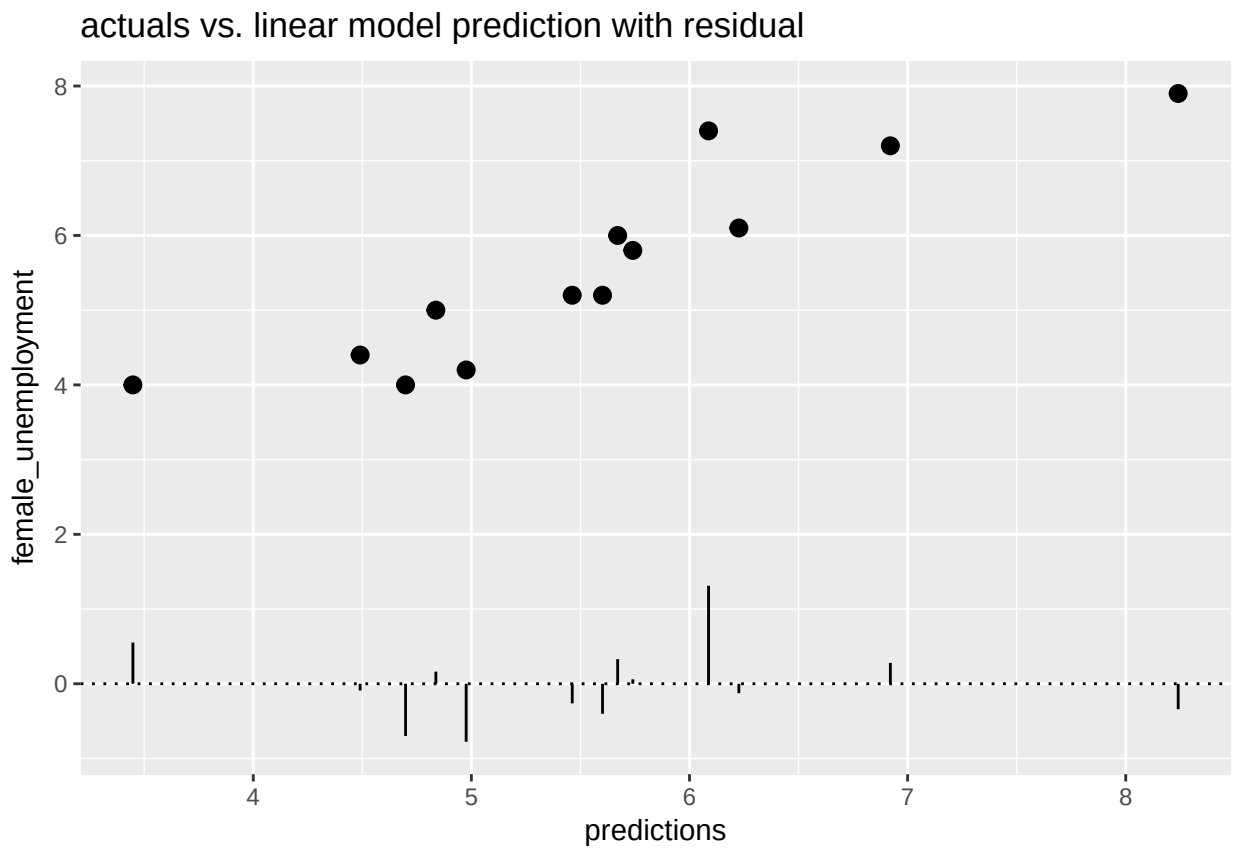
```
summary(unemployment)
```

```
##  male_unemployment female_unemployment   prediction
##  Min.   :2.900     Min.   :4.000       Min.   :3.448
##  1st Qu.:4.900     1st Qu.:4.400       1st Qu.:4.837
##  Median :6.000     Median :5.200       Median :5.601
##  Mean   :5.954     Mean   :5.569       Mean   :5.569
##  3rd Qu.:6.700     3rd Qu.:6.100       3rd Qu.:6.087
##  Max.   :9.800     Max.   :7.900       Max.   :8.240
```

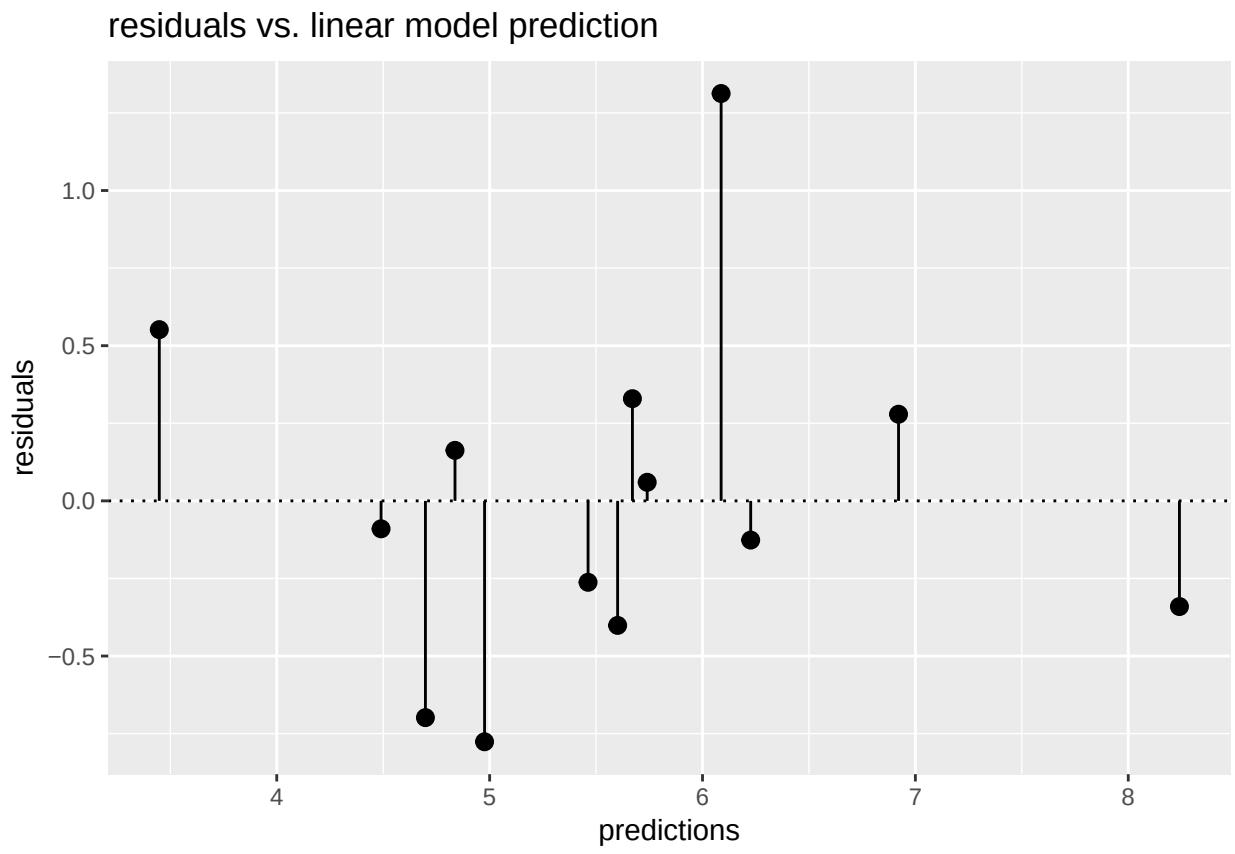
```r
# Make predictions from the model
unemployment$predictions <- predict(unemployment_model, unemployment)

# Plot predictions (on x-axis) versus the female_unemployment rates
ggplot(unemployment, aes(x = predictions, y = female_unemployment)) +
  geom_point() +
  geom_abline()
```




```r
# Calculate residuals
unemployment$residuals <- unemployment$female_unemployment - unemployment$predictions

# Predictions (on x-axis) versus the actuals with residuals
ggplot(unemployment, aes(x = predictions, y = female_unemployment)) + geom_point() +
  geom_pointrange(aes(ymin = 0, ymax = residuals)) +
  geom_hline(yintercept = 0, linetype = 3) +
  ggtitle("actuals vs. linear model prediction with residual")
```




```
# Fill in the blanks to plot predictions (on x-axis) versus the residuals
ggplot(unemployment, aes(x = predictions, y = residuals)) +
  geom_pointrange(aes(ymin = 0, ymax = residuals)) +
  geom_hline(yintercept = 0, linetype = 3) +
  ggtitle("residuals vs. linear model prediction")
```



We can also plot a gain curve using the WVPlots package. The syntax is:

GainCurvePlot(frame, xvar, truthvar, title)

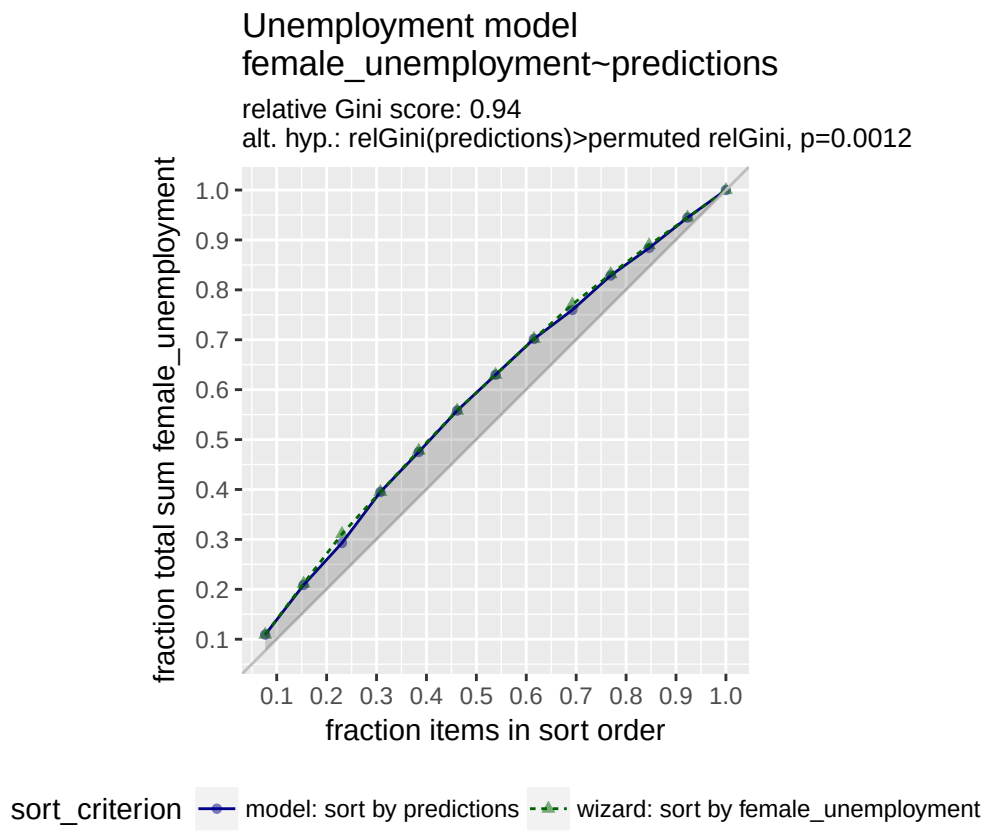
where

- frame is a data frame
- xvar and truthvar are strings naming the prediction and actual outcome columns of frame
- title is the title of the plot

```
library(WVPlots)
```

```
## Warning: package 'WVPlots' was built under R version 3.4.4
```

```
# Plot the Gain Curve
GainCurvePlot(unemployment, "predictions", "female_unemployment", "Unemployment model")
```



A relative gini coefficient close to one shows that the model correctly sorts high unemployment situations from lower ones.

Another way to evaluate our model is to use the RMSE. You want RMSE to be small. One heuristic is to compare the RMSE to the standard deviation of the outcome. With a good model, the RMSE should be smaller.

```
# For convenience put the residuals in the variable res
res <- unemployment$residuals

# Calculate RMSE, assign it to the variable rmse and print it
(rmse <- sqrt(mean(res^2)))
```

```
## [1] 0.5337612
```

```
# Calculate the standard deviation of female_unemployment and print it
(sd_unemployment <- sd(unemployment$female_unemployment))
```

```
## [1] 1.314271
```

```
# Calculate it as a fraction < 1 is good
rmse / sd_unemployment
```

```
## [1] 0.4061273
```

An RMSE much smaller than the outcome's standard deviation suggests a model that predicts well.

Another way of evaluating a model is R squared. We can calculate this as the 1 - RSS / TSS RSS = Residual Sum of Squares TSS is Total Sum of Squares.

Then the total sum of squares tss ("total variance") of the data is:

$$tss = \sum (y - \bar{y})^2$$

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

The residual sum of squared errors of the model, rss is:

$$rss = \sum res^2$$

R^2 (R-Squared), the "variance explained" by the model, is then:

$$1 - \frac{rss}{tss}$$

Figure 3.1: R Squared

Most packages or regression functions come with built in R squared metrics, the summary will print it out or we can use the glance() function from broom. Here we calculate them manually.

```
# Calculate mean female_unemployment: fe_mean. Print it
(fe_mean <- mean(unemployment$female_unemployment))
```

```
## [1] 5.569231
```

```
# Calculate total sum of squares: tss. Print it
(tss <- sum((unemployment$female_unemployment - fe_mean)^2))
```

```
## [1] 20.72769
```

```
# Calculate residual sum of squares: rss. Print it
(rss <- sum((unemployment$residuals)^2))
```

```
## [1] 3.703714
```

```
# Calculate R-squared: rsq. Print it. Is it a good fit?
(rsq <- 1 - (rss/tss))
```

```
## [1] 0.8213157
```

```
# Get R-squared from glance. Print it
(rsq_glance <- broom::glance(unemployment_model)$r.squared)
```

```
## [1] 0.8213157
```

Correlation or rho shows the strength of the linear relationship between two variables.

```
# Get the correlation between the prediction and true outcome: rho and print it
(rho <- cor(unemployment$predictions, unemployment$female_unemployment))
```

```
## [1] 0.9062647
```

```
rho
```

```
## [1] 0.9062647
```

```
# Square rho: rho2 and print it
(rho2 <- rho ^ 2)
```

```
## [1] 0.8213157
```

```
# Get R-squared from glance and print it
(rsq_glance <- broom::glance(unemployment_model)$r.squared)
```

```
## [1] 0.8213157
```

So far we have only looked at how well our data fits the observed variables, without new data. A better way of evaluating our model is to split out data in to a training and test set of data, then see how well the model predicts to the test data, having built the model on the train data. WE want our RMSE on the test data to be similar to the RMSE on the train data, if it is much lower than we may have overfitted our model. If the number of observations is too small to do a split, we can use cross validation to acheive a similar result.

In the following code we will split mpg into a training set mpg_train (75% of the data) and a test set mpg_test (25% of the data). One way to do this is to generate a column of uniform random numbers between 0 and 1, using the function runif(). There other ways, such as sample (see Supervised LEarning In R - Classification notes, do search for 0.75).

If using run if, we do:

- Generate a vector of uniform random numbers: gp = runif(N).
- dframe[gp < X,] will be about the right size.
- dframe[gp >= X,] will be the complement.

```
# Load the mpg data
mpg <- ggplot2::mpg

# take a look at the data
summary(mpg)
```

```
##  manufacturer          model               displ            year
##  Length:234         Length:234         Min.   :1.600   Min.   :1999
##  Class :character   Class :character   1st Qu.:2.400   1st Qu.:1999
##  Mode  :character   Mode  :character   Median :3.300   Median :2004
##                                        Mean   :3.472   Mean   :2004
##                                        3rd Qu.:4.600   3rd Qu.:2008
##                                        Max.   :7.000   Max.   :2008
##       cyl           trans               drv                 cty
##  Min.   :4.000   Length:234         Length:234         Min.   : 9.00
##  1st Qu.:4.000   Class :character   Class :character   1st Qu.:14.00
##  Median :6.000   Mode  :character   Mode  :character   Median :17.00
##  Mean   :5.889                                         Mean   :16.86
##  3rd Qu.:8.000                                         3rd Qu.:19.00
##  Max.   :8.000                                         Max.   :35.00
##       hwy             fl                 class
##  Min.   :12.00   Length:234         Length:234
##  1st Qu.:18.00   Class :character   Class :character
##  Median :24.00   Mode  :character   Mode  :character
##  Mean   :23.44
##  3rd Qu.:27.00
##  Max.   :44.00
```

```
dim(mpg)
```

```
## [1] 234  11
```

```
# Use nrow to get the number of rows in mpg (N) and print it
(N <- nrow(mpg))
```

```
## [1] 234
```

```
N
```

```
## [1] 234
```

```
# Calculate how many rows 75% of N should be and print it
# Hint: use round() to get an integer
(target <- round(N * 0.75))
```

```
## [1] 176
```

```
target
```

```
## [1] 176
```

```
# Create the vector of N uniform random variables: gp
gp <- runif(N)

# Use gp to create the training set: mpg_train (75% of data) and mpg_test (25% of data)
mpg_train <- mpg[gp < 0.75, ]
mpg_test <- mpg[gp >= 0.75, ]

# Use nrow() to examine mpg_train and mpg_test
nrow(mpg_train)
```

```
## [1] 167
```

```
nrow(mpg_test)
```

```
## [1] 67
```

It is likely our target number of rows will slightly different than what is sampled, but they should be close enough.

Next we use these datasets to create models for prediction.

```
# mpg_train is in the workspace
summary(mpg_train)
```

```
##  manufacturer          model               displ            year
##  Length:167         Length:167         Min.   :1.600   Min.   :1999
##  Class :character   Class :character   1st Qu.:2.400   1st Qu.:1999
##  Mode  :character   Mode  :character   Median :3.300   Median :2008
##                                        Mean   :3.464   Mean   :2004
##                                        3rd Qu.:4.600   3rd Qu.:2008
##                                        Max.   :7.000   Max.   :2008
##       cyl           trans               drv                 cty
##  Min.   :4.000   Length:167         Length:167         Min.   : 9.00
##  1st Qu.:4.000   Class :character   Class :character   1st Qu.:14.00
##  Median :6.000   Mode  :character   Mode  :character   Median :17.00
##  Mean   :5.898                                         Mean   :16.84
##  3rd Qu.:8.000                                         3rd Qu.:19.00
##  Max.   :8.000                                         Max.   :35.00
##       hwy             fl                 class
##  Min.   :12.00   Length:167         Length:167
```

```
##  1st Qu.:18.00   Class :character   Class :character
##  Median :25.00   Mode  :character   Mode  :character
##  Mean   :23.44
##  3rd Qu.:27.00
##  Max.   :44.00
```

```
# Create a formula to express cty as a function of hwy: fmla and print it.
(fmla <- as.formula("cty ~ hwy"))
```

```
## cty ~ hwy
```

```
# Now use lm() to build a model mpg_model from mpg_train that predicts cty from hwy
mpg_model <- lm(fmla, data = mpg_train)

# Use summary() to examine the model
summary(mpg_model)
```

```
##
## Call:
## lm(formula = fmla, data = mpg_train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.8877 -0.7091 -0.1591  0.6195  4.3552
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.10203    0.37749   2.919    0.004 **
## hwy          0.67143    0.01561  43.006   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.199 on 165 degrees of freedom
## Multiple R-squared:  0.9181, Adjusted R-squared:  0.9176
## F-statistic:  1850 on 1 and 165 DF,  p-value: < 2.2e-16
```

Next we will test the model mpg_model on the test data, mpg_test. We will use two functions rather than calculate the rmse and r_squared manually:

- Metrics::rmse(predcol, ycol)
- tsensembler::r_squared(r_squared(y, y_hat)

```
# predict cty from hwy for the training set
mpg_train$pred <- predict(mpg_model, mpg_train)

# predict cty from hwy for the test set
mpg_test$pred <- predict(mpg_model, mpg_test)

# Evaluate the rmse on both training and test data and print them
(rmse_train <- Metrics::rmse(mpg_train$pred, mpg_train$cty))
```

```
## [1] 1.191911
```

```
(rmse_test <-  Metrics::rmse(mpg_test$pred, mpg_test$cty))
```

```
## [1] 1.381538
```

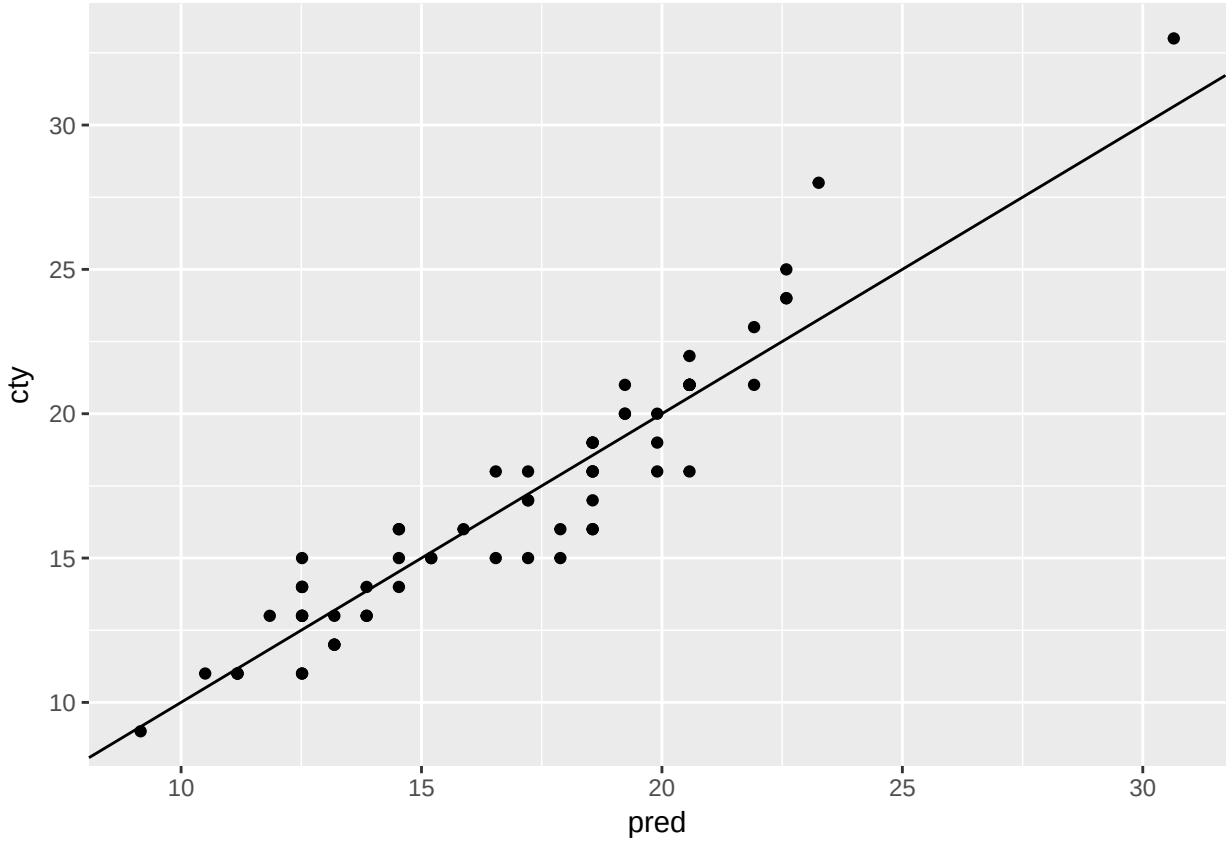
```
# Evaluate the r-squared on both training and test data.and print them
(rsq_train <- tsensembler::r_squared(mpg_train$pred, mpg_train$cty))
```

```
## [1] 0.9107881
```

```
(rsq_test <- tsensembler::r_squared(mpg_test$pred, mpg_test$cty))
```

```
## [1] 0.879934
```

```
# Plot the predictions (on the x-axis) against the outcome (cty) on the test data
ggplot(mpg_test, aes(x = pred, y = cty)) +
  geom_point() +
  geom_abline()
```



There are a number of ways to create cross fold validation - the caret package being one of the most flexible. Here we use the vtreat package and KWayCrossValidation(). We first create our CV plan.

```
# Load the package vtreat
library(vtreat)

# Get the number of rows in mpg
nRows <- nrow(mpg)

# Implement the 5-fold cross-fold plan with vtreat
splitPlan <- kWayCrossValidation(nRows = nRows, nSplits = 5, dframe = NULL, y = NULL)

# Examine the split plan
str(splitPlan)
```

```
## List of 5
##  $ :List of 2
##   ..$ train: int [1:188] 1 2 3 4 5 7 8 9 10 13 ...
##   ..$ app  : int [1:46] 210 198 97 127 142 84 22 173 187 38 ...
##  $ :List of 2
##   ..$ train: int [1:187] 1 3 4 5 6 8 9 11 12 13 ...
##   ..$ app  : int [1:47] 7 111 50 202 141 208 220 178 55 158 ...
##  $ :List of 2
##   ..$ train: int [1:187] 2 3 4 5 6 7 8 10 11 12 ...
##   ..$ app  : int [1:47] 9 125 126 114 51 143 129 205 32 28 ...
##  $ :List of 2
##   ..$ train: int [1:187] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ app  : int [1:47] 30 138 76 113 73 149 168 197 53 70 ...
##  $ :List of 2
##   ..$ train: int [1:187] 1 2 6 7 9 10 11 12 13 14 ...
##   ..$ app  : int [1:47] 75 195 4 162 79 17 177 181 31 77 ...
##  - attr(*, "splitmethod")= chr "kwaycross"
```

Next we iterate through our training plan, creating a new model for each split or fold.

```
# mpg is in the workspace
summary(mpg)
```

```
##  manufacturer          model               displ            year
##  Length:234         Length:234         Min.   :1.600   Min.   :1999
##  Class :character   Class :character   1st Qu.:2.400   1st Qu.:1999
##  Mode  :character   Mode  :character   Median :3.300   Median :2004
##                                        Mean   :3.472   Mean   :2004
##                                        3rd Qu.:4.600   3rd Qu.:2008
##                                        Max.   :7.000   Max.   :2008
##       cyl           trans               drv                 cty
##  Min.   :4.000   Length:234         Length:234         Min.   : 9.00
##  1st Qu.:4.000   Class :character   Class :character   1st Qu.:14.00
##  Median :6.000   Mode  :character   Mode  :character   Median :17.00
##  Mean   :5.889                                         Mean   :16.86
##  3rd Qu.:8.000                                         3rd Qu.:19.00
##  Max.   :8.000                                         Max.   :35.00
##       hwy             fl                class
##  Min.   :12.00   Length:234         Length:234
##  1st Qu.:18.00   Class :character   Class :character
##  Median :24.00   Mode  :character   Mode  :character
##  Mean   :23.44
##  3rd Qu.:27.00
##  Max.   :44.00
```

```
# splitPlan is in the workspace
str(splitPlan)
```

```
## List of 5
##  $ :List of 2
##   ..$ train: int [1:188] 1 2 3 4 5 7 8 9 10 13 ...
##   ..$ app  : int [1:46] 210 198 97 127 142 84 22 173 187 38 ...
##  $ :List of 2
##   ..$ train: int [1:187] 1 3 4 5 6 8 9 11 12 13 ...
##   ..$ app  : int [1:47] 7 111 50 202 141 208 220 178 55 158 ...
##  $ :List of 2
```

```
##   ..$ train: int [1:187] 2 3 4 5 6 7 8 10 11 12 ...
##   ..$ app  : int [1:47] 9 125 126 114 51 143 129 205 32 28 ...
##  $ :List of 2
##   ..$ train: int [1:187] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ app  : int [1:47] 30 138 76 113 73 149 168 197 53 70 ...
##  $ :List of 2
##   ..$ train: int [1:187] 1 2 6 7 9 10 11 12 13 14 ...
##   ..$ app  : int [1:47] 75 195 4 162 79 17 177 181 31 77 ...
##  - attr(*, "splitmethod")= chr "kwaycross"
```

```
# Run the 5-fold cross validation plan from splitPlan
k <- 5 # Number of folds
mpg$pred.cv <- 0
for(i in 1:k) {
  split <- splitPlan[[i]]
  model <- lm(cty ~ hwy, data = mpg[split$train,])
  mpg$pred.cv[split$app] <- predict(model, newdata = mpg[split$app,])
}

# Predict from a full model
mpg$pred <- predict(lm(cty ~ hwy, data = mpg))

# Get the rmse of the full model's predictions
Metrics::rmse(mpg$pred, mpg$cty)
```

```
## [1] 1.247045
```

```
# Get the rmse of the cross-validation predictions
Metrics::rmse(mpg$pred.cv, mpg$cty)
```

```
## [1] 1.262551
```

This has now calculated the models out of sample error using cross validation. CV validates the modelling process, not whether the model is a good one or not. Here we we the full model RMSE is very similar to the CV RMSE suggesting we are not over fitting the data.

3.3 Issues to Consider

When using categorical variables, n - 1 variables are created and coded as dummy vars (0 or 1), one one variable is left out as the reference model. This is usually referred to as dummy variable creation or one hot encoding. So our model then compares how the presence of one variable (one categorical var or response) affects the outcome, ceteris parabus, against the baseline cateogrical variable or response. This is best done where the number of variables is quite small, to avoid overfitting.

Some functions/models do this one hot or dummy variable creation automatically 'under the hood' whereas others need more pre-procesing prior to modelling. For some approaches (tree models) we can leave them as nominal e.g. 1,2 and 3. BUT this won't work where the data/calculations are geometric, such as linear regression (geometric), PCA and some clustering methods (eigen space calculations). More information is available in The Basics of Encoding Categorical Data for Predictive Models

```
# Load the data from Sleuth and modify for the purposes of demostration
library(Sleuth3)
```

```
## Warning: package 'Sleuth3' was built under R version 3.4.3
```

```
flowers <- print(case0901)
```

```
##    Flowers Time Intensity
## 1     62.3    1       150
## 2     77.4    1       150
## 3     55.3    1       300
## 4     54.2    1       300
## 5     49.6    1       450
## 6     61.9    1       450
## 7     39.4    1       600
## 8     45.7    1       600
## 9     31.3    1       750
## 10    44.9    1       750
## 11    36.8    1       900
## 12    41.9    1       900
## 13    77.8    2       150
## 14    75.6    2       150
## 15    69.1    2       300
## 16    78.0    2       300
## 17    57.0    2       450
## 18    71.1    2       450
## 19    62.9    2       600
## 20    52.2    2       600
## 21    60.3    2       750
## 22    45.6    2       750
## 23    52.6    2       900
## 24    44.4    2       900
```

```
flowers$Time[flowers$Time==1] <- "Late"
flowers$Time[flowers$Time==2] <- "Early"

# Call str on flowers to see the types of each column
str(flowers)
```

```
## 'data.frame':    24 obs. of  3 variables:
##  $ Flowers  : num  62.3 77.4 55.3 54.2 49.6 61.9 39.4 45.7 31.3 44.9 ...
##  $ Time     : chr  "Late" "Late" "Late" "Late" ...
##  $ Intensity: int  150 150 300 300 450 450 600 600 750 750 ...
```

```
# Use unique() to see how many possible values Time takes
unique(flowers$Time)
```

```
## [1] "Late"  "Early"
```

```
# Build a formula to express Flowers as a function of Intensity and Time: fmla. Print it
(fmla <- as.formula("Flowers ~ Intensity + Time"))
```

```
## Flowers ~ Intensity + Time
```

```
# Use fmla and model.matrix to see how the data is represented for modeling
mmat <- model.matrix(fmla, flowers)

# Examine the first 20 lines of flowers
head(flowers, n = 20)
```

```
##    Flowers  Time Intensity
## 1     62.3  Late       150
```

```
## 2     77.4  Late       150
## 3     55.3  Late       300
## 4     54.2  Late       300
## 5     49.6  Late       450
## 6     61.9  Late       450
## 7     39.4  Late       600
## 8     45.7  Late       600
## 9     31.3  Late       750
## 10    44.9  Late       750
## 11    36.8  Late       900
## 12    41.9  Late       900
## 13    77.8 Early       150
## 14    75.6 Early       150
## 15    69.1 Early       300
## 16    78.0 Early       300
## 17    57.0 Early       450
## 18    71.1 Early       450
## 19    62.9 Early       600
## 20    52.2 Early       600
```

```
# Examine the first 20 lines of mmat
head(mmat, n = 20)
```

```
##    (Intercept) Intensity TimeLate
## 1            1       150        1
## 2            1       150        1
## 3            1       300        1
## 4            1       300        1
## 5            1       450        1
## 6            1       450        1
## 7            1       600        1
## 8            1       600        1
## 9            1       750        1
## 10           1       750        1
## 11           1       900        1
## 12           1       900        1
## 13           1       150        0
## 14           1       150        0
## 15           1       300        0
## 16           1       300        0
## 17           1       450        0
## 18           1       450        0
## 19           1       600        0
## 20           1       600        0
```

Next we will fit a linear model to the flowers data, to predict Flowers as a function of Time and Intensity.

```
# flowers in is the workspace
str(flowers)
```

```
## 'data.frame':    24 obs. of  3 variables:
##  $ Flowers  : num  62.3 77.4 55.3 54.2 49.6 61.9 39.4 45.7 31.3 44.9 ...
##  $ Time     : chr  "Late" "Late" "Late" "Late" ...
##  $ Intensity: int  150 150 300 300 450 450 600 600 750 750 ...
```

```
# fmla is in the workspace
fmla
```

```
## Flowers ~ Intensity + Time
```

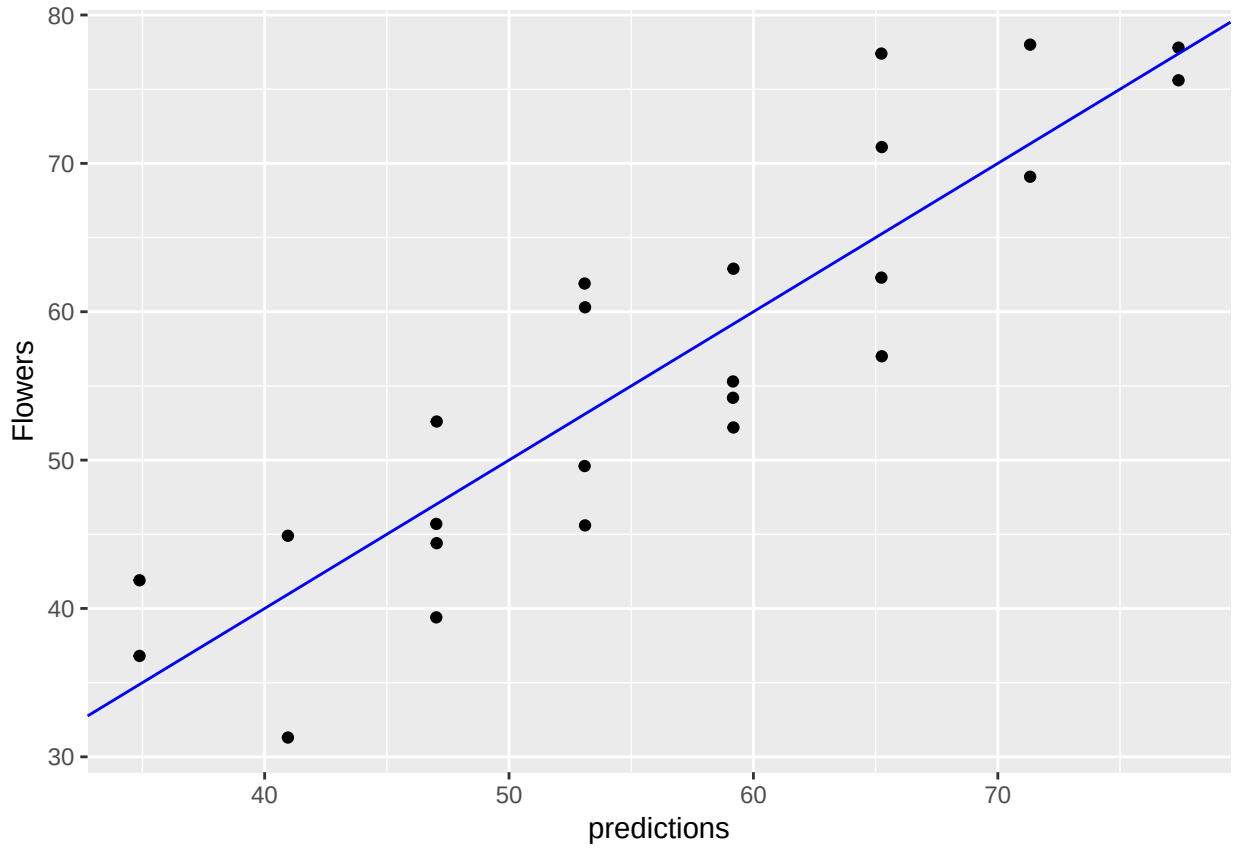
```
# Fit a model to predict Flowers from Intensity and Time : flower_model
flower_model <- lm(fmla, data = flowers)

# Use summary on mmat to remind yourself of its structure
summary(mmat)
```

```
##   (Intercept)   Intensity      TimeLate
##  Min.   :1    Min.   :150   Min.   :0.0
##  1st Qu.:1    1st Qu.:300   1st Qu.:0.0
##  Median :1    Median :525   Median :0.5
##  Mean   :1    Mean   :525   Mean   :0.5
##  3rd Qu.:1    3rd Qu.:750   3rd Qu.:1.0
##  Max.   :1    Max.   :900   Max.   :1.0
```

```
# Use summary to examine flower_model
summary(flower_model)
```

```
##
## Call:
## lm(formula = fmla, data = flowers)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -9.652 -4.139 -1.558  5.632 12.165
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  83.464167   3.273772  25.495  < 2e-16 ***
## Intensity    -0.040471   0.005132  -7.886 1.04e-07 ***
## TimeLate    -12.158333   2.629557  -4.624 0.000146 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.441 on 21 degrees of freedom
## Multiple R-squared:  0.7992, Adjusted R-squared:   0.78
## F-statistic: 41.78 on 2 and 21 DF,  p-value: 4.786e-08
```

```
# Predict the number of flowers on each plant
flowers$predictions <- predict(flower_model, data = flowers)

# Plot predictions vs actual flowers (predictions on x-axis)
ggplot(flowers, aes(x = predictions, y = Flowers)) +
  geom_point() +
  geom_abline(color = "blue")
```



3.3.1 Interactions

In linear models we assume that the variables affect the model linearly and additively. But sometimes this is not the case - with interactions variables, wehn combinded together, can be more than the sum of their parts. If for instance, the effect one one variable is dependent on the level of some other variable, we say their is an interaction. The simultaneous effect on the outcome variable is no longer additive. If we want to model this in R we have different options

- Interaction - Colon(:) e.g. y ~ a:b
- Main effects and interaction - Asterisk() *e.g.* $y \sim a$ b which is the same as y ~ a + b + a:b
- The product of two variables - I function (I) e.g. y ~ I(a*b)

In the following code shows how to use interactions to model the effect of gender and gastric activity on alcohol metabolism.

The data frame alcohol has columns:

- Metabol: the alcohol metabolism rate
- Gastric: the rate of gastric alcohol dehydrogenase activity
- Sex: the sex of the drinker (Male or Female)

```
# Create the formula with main effects only
(fmla_add <- as.formula("Metabol ~ Gastric + Sex") )

# Create the formula with interactions
(fmla_interaction <- as.formula("Metabol ~ Gastric:Sex + Gastric") )
```

```
# Fit the main effects only model
model_add <- lm(fmla_add, data = alcohol)

# Fit the interaction model
model_interaction <- lm(fmla_interaction, data = alcohol)

# Call summary on both models and compare
summary(model_add)
summary(model_interaction)
```

The following code compares the performance of the interaction model previously fit to the performance of a main-effects only model. Because this data set is small, we use cross-validation to simulate making predictions on out-of-sample data.

```
# Create the splitting plan for 3-fold cross validation
set.seed(34245)  # set the seed for reproducibility
splitPlan <- kWayCrossValidation(nrow(alcohol), nSplits = 3, dframe = NULL, y = NULL)

# Sample code: Get cross-val predictions for main-effects only model
alcohol$pred_add <- 0  # initialize the prediction vector
for(i in 1:3) {
  split <- splitPlan[[i]]
  model_add <- lm(fmla_add, data = alcohol[split$train, ])
  alcohol$pred_add[split$app] <- predict(model_add, newdata = alcohol[split$app, ])
}

# Get the cross-val predictions for the model with interactions
alcohol$pred_interaction <- 0 # initialize the prediction vector
for(i in 1:3) {
  split <- splitPlan[[i]]
  model_interaction <- lm(fmla_interaction, data = alcohol[split$train, ])
  alcohol$pred_interaction[split$app] <- predict(model_interaction, newdata = alcohol[split$app, ])
}

# Get RMSE using gather from dplyr
alcohol %>%
  gather(key = modeltype, value = pred, pred_add, pred_interaction) %>%
  mutate(residuals = Metabol - pred) %>%
  group_by(modeltype) %>%
  summarize(rmse = sqrt(mean(residuals^2)))
```

3.3.2 Transforming the response before modeling

Sometimes better models are achieved by transforming the output rather than predicting the output directly. We often want to log transform monetary values like income. This can be achied using log(y) or specify the same function within a model itself, predict using that model, then transform the data back to it's original format i.e not transformed e.g.

1. model <- lm(log(y) ~ x, data = train)

2. logpred <- predict(model, data = test)
3. pred <- exp(logpred)

As the error is relative to the size of the outcome, we should use the root mean squared relative error to

compare two models with one not being log transformed and one being log transformed. We do this by dividing the mean error / variable in quetion (log or non-log).

The next code section uses some toy data to demostrate how we make calculations for relative error.

```
# Load the demostration data
fdata <- read.csv("./files/SLinRRegression/fdata.csv")
str(fdata)
```

```
## 'data.frame':    100 obs. of  3 variables:
##  $ y    : num  9.15 1.9 -3.86 2.39 1.54 ...
##  $ pred : num  6.43 3.47 1.59 3.76 9.51 ...
##  $ label: Factor w/ 2 levels "large purchases",..: 2 2 2 2 2 2 2 2 2 2 ...
```

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
# fdata is in the workspace
summary(fdata)
```

```
##        y                 pred                       label
##  Min.   :  -5.894   Min.   :   1.072   large purchases:50
##  1st Qu.:   5.407   1st Qu.:   6.373   small purchases:50
##  Median :  57.374   Median :  55.693
##  Mean   : 306.204   Mean   : 305.905
##  3rd Qu.: 550.903   3rd Qu.: 547.886
##  Max.   :1101.619   Max.   :1098.896
```

```
# Examine the data: generate the summaries for the groups large and small:
fdata %>%
    group_by(label) %>%     # group by small/large purchases
    summarize(min  = min(y),   # min of y
              mean = mean(y),   # mean of y
              max  = max(y))   # max of y
```

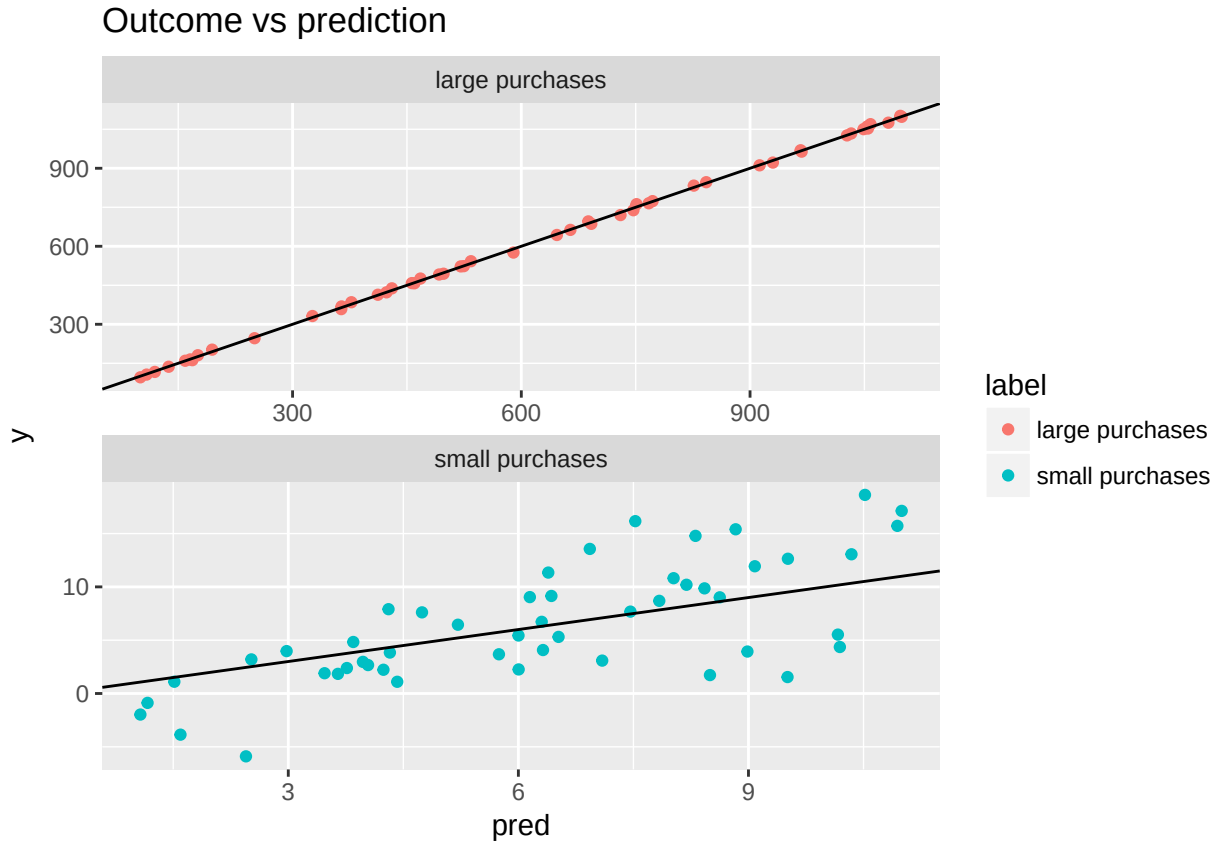
```
## # A tibble: 2 x 4
##             label       min       mean        max
##            <fctr>     <dbl>      <dbl>      <dbl>
## 1 large purchases 96.119814 605.928673 1101.61864
## 2 small purchases -5.893499   6.478254   18.62829
```

```
# Fill in the blanks to add error columns
fdata2 <- fdata %>%
         group_by(label) %>%       # group by label
           mutate(residual = pred - y,  # Residual
                  relerr   = residual / y)  # Relative error

# Compare the rmse and rmse.rel of the large and small groups:
fdata2 %>%
```

```
  group_by(label) %>%
  summarize(rmse     = sqrt(mean(residual^2)),    # RMSE
            rmse.rel = sqrt(mean(relerr^2)))    # Root mean squared relative error
```

```
## # A tibble: 2 x 3
##             label     rmse   rmse.rel
##            <fctr>    <dbl>      <dbl>
## 1 large purchases 5.544439 0.01473322
## 2 small purchases 4.014969 1.24965673
```

```
# Plot the predictions for both groups of purchases
ggplot(fdata2, aes(x = pred, y = y, color = label)) +
  geom_point() +
  geom_abline() +
  facet_wrap(~ label, ncol = 1, scales = "free") +
  ggtitle("Outcome vs prediction")
```



From this example how a model with larger RMSE might still be better as can be observed using the chart, if relative errors are more important than absolute errors as the relative error is much smaller for large purchases using the table.

The follwoing code uses data loaded which records subjects' incomes in 2005, as well as the results of several aptitude tests taken by the subjects in 1981. The data is split into training and test sets. The code demostrates building a model of log(income) from the inputs, and then convert log(income) back into income.

When you transform the output before modeling, you have to 'reverse transform' the resulting predictions after applying the model.

```
# Load the unemployment data
load("./files/SLinRRegression/Income.RData")

# Examine Income2005 in the training set
summary(incometrain$Income2005)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      63   23000   39000   49894   61500  703637
```

```
# Write the formula for log income as a function of the tests and print it
(fmla.log <- as.formula("log(Income2005) ~  Arith + Word + Parag + Math + AFQT"))
```

```
## log(Income2005) ~ Arith + Word + Parag + Math + AFQT
```

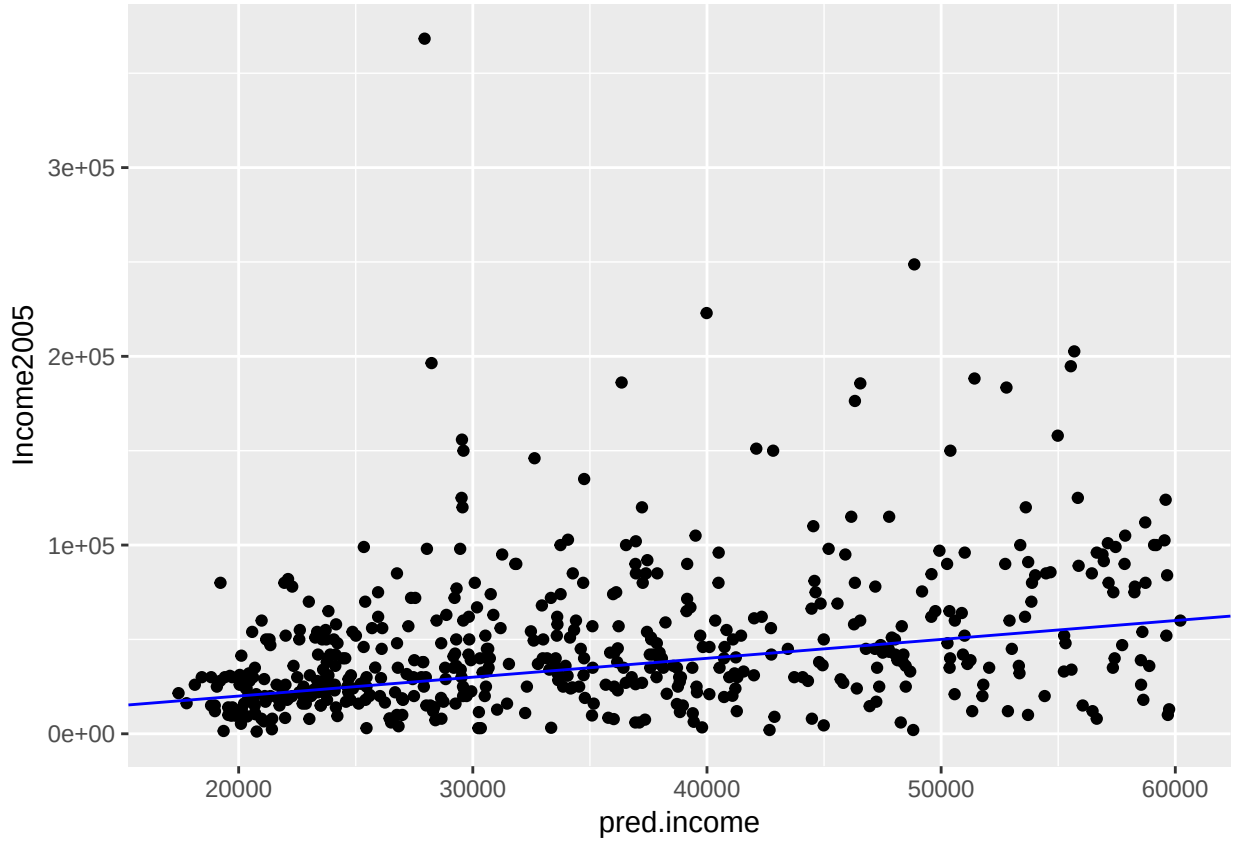
```
# Fit the linear model
model.log <-  lm(fmla.log, data = incometrain)

# Make predictions on income_test
incometest$logpred <- predict(model.log, incometest)
summary(incometest$logpred)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   9.766  10.133  10.423  10.419  10.705  11.006
```

```
# Convert the predictions to monetary units
incometest$pred.income <- exp(incometest$logpred)
summary(incometest$pred.income)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   17432   25167   33615   35363   44566   60217
```

```
#  Plot predicted income (x axis) vs income
ggplot(incometest, aes(x = pred.income, y = Income2005)) +
  geom_point() +
  geom_abline(color = "blue")
```



Next we show that log-transforming a monetary output before modeling improves mean relative error (but increases RMSE) compared to modeling the monetary output directly. We compare two models, the log transmormed model from before to an absolute (i.e. not transformed) model called model.abs.

```
library(tidyr) # for gather commands

# Write our model formula
(fmla.abs <- as.formula("Income2005 ~ Arith + Word + Parag + Math + AFQT"))

# Buil the model
model.abs <- lm(formula = fmla.abs, data = incometrain)

# Add predictions to the test set
income_test <- incometest %>%
  dplyr::mutate(pred.absmodel = predict(model.abs, incometest),        # predictions from model.abs
         pred.logmodel = exp(predict(model.log, incometest)))          # predictions from model.log

# Gather the predictions and calculate residuals and relative error
income_long <- incometest %>%
  tidyr::gather(key = modeltype, value = pred, pred.absmodel, pred.logmodel) %>%
  dplyr::mutate(residual = pred - Income2005,   # residuals
         relerr   = residual / Income2005)   # relative error

# Calculate RMSE and relative RMSE and compare
income_long %>%
  group_by(modeltype) %>%      # group by modeltype
  summarize(rmse     = sqrt(mean(residual^2)),    # RMSE
```

```
            rmse.rel = sqrt(mean(relerr^2)))    # Root mean squared relative error
```

Modeling log(income) can reduce the relative error of the fit, at the cost of increased RMSE. Which tradeoff to make depends on the goals of the project.

3.3.3 Transforming Input variables

So far we've looked at transforming the output variables, but there are instance where we might want to transform the input variables. Usually, this is because you have some domain knowledge about the subject which suggests this is the case. You often want to transform monetary values as we say before. We might also want to create power relationships in our variables. If we do this and we don't know which power is best, based on domain knowledge, we might try various powers then see which yields the lowest prediction error and out of sample/CV error.

In the next section of code we will build a model to predict price from a measure of the house's size (surface area).

Because ^ is also a symbol to express interactions, we use the function I() to treat the expression x^2 "as is": that is, as the square of x rather than the interaction of x with itself.

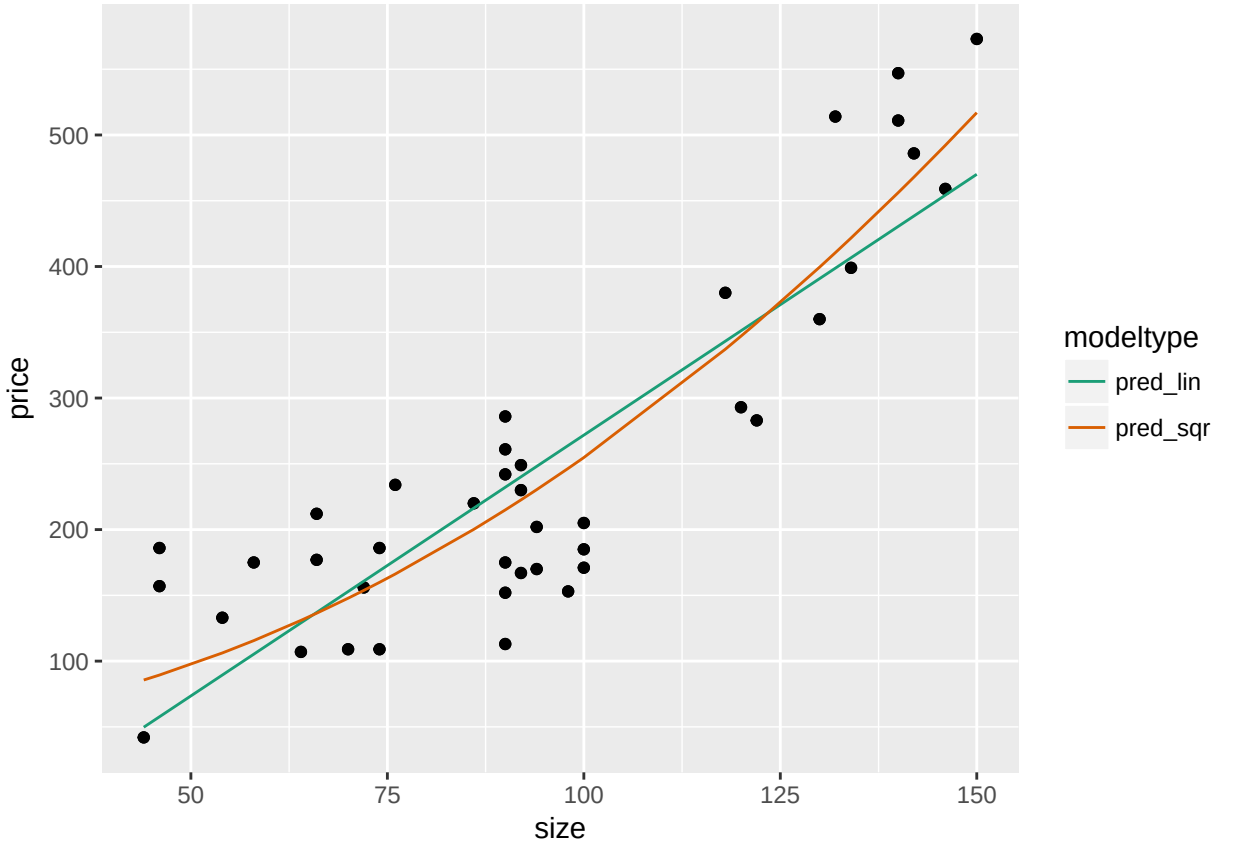
```
# Load the house price data
houseprice <- readRDS("./files/SLinRRegression/houseprice.rds")

# explore the data
summary(houseprice)
```

```
##       size           price
##  Min.   : 44.0   Min.   : 42.0
##  1st Qu.: 73.5   1st Qu.:164.5
##  Median : 91.0   Median :203.5
##  Mean   : 94.3   Mean   :249.2
##  3rd Qu.:118.5   3rd Qu.:287.8
##  Max.   :150.0   Max.   :573.0
```

```
# Create the formula for price as a function of squared size
(fmla_sqr <- as.formula("price ~ I(size^2)"))
```

```
## price ~ I(size^2)
```

```
# Fit a model of price as a function of squared size (use fmla_sqr)
model_sqr <- lm(fmla_sqr, data = houseprice)

# Fit a model of price as a linear function of size
model_lin <- lm(price ~ size, data = houseprice)

# Make predictions and compare
houseprice %>%
    dplyr::mutate(pred_lin = predict(model_lin),       # predictions from linear model
           pred_sqr = predict(model_sqr)) %>%   # predictions from quadratic model
    tidyr::gather(key = modeltype, value = pred, pred_lin, pred_sqr) %>% # gather the predictions
    ggplot2::ggplot(aes(x = size)) +
       geom_point(aes(y = price)) +                    # actual prices
       geom_line(aes(y = pred, color = modeltype)) + # the predictions
       scale_color_brewer(palette = "Dark2")
```



As it appears that the quadratic model better fits the house price data, we will next confirm whether this is the case when using out of sample data. As the data is small, we will use cross-validation. We will also compare the results from a linear model.

```
# houseprice is in the workspace
summary(houseprice)
```

```
##       size           price
##  Min.   : 44.0   Min.   : 42.0
##  1st Qu.: 73.5   1st Qu.:164.5
##  Median : 91.0   Median :203.5
##  Mean   : 94.3   Mean   :249.2
##  3rd Qu.:118.5   3rd Qu.:287.8
##  Max.   :150.0   Max.   :573.0
```

```
# fmla_sqr is in the workspace
fmla_sqr
```

```
## price ~ I(size^2)
```

```
# Create a splitting plan for 3-fold cross validation
set.seed(34245)  # set the seed for reproducibility
splitPlan <- kWayCrossValidation(nRows = nrow(houseprice), nSplits = 3, dframe = NULL, y = NULL)

# Sample code: get cross-val predictions for price ~ size
houseprice$pred_lin <- 0  # initialize the prediction vector
for(i in 1:3) {
  split <- splitPlan[[i]]
```

```
  model_lin <- lm(price ~ size, data = houseprice[split$train,])
  houseprice$pred_lin[split$app] <- predict(model_lin, newdata = houseprice[split$app,])
}

# Get cross-val predictions for price as a function of size^2 (use fmla_sqr)
houseprice$pred_sqr <- 0 # initialize the prediction vector
for(i in 1:3) {
  split <- splitPlan[[i]]
  model_sqr <- lm(fmla_sqr, data = houseprice[split$train, ])
  houseprice$pred_sqr[split$app] <- predict(model_sqr, newdata = houseprice[split$app, ])
}

# Gather the predictions and calculate the residuals
houseprice_long <- houseprice %>%
  tidyr::gather(key = modeltype, value = pred, pred_lin, pred_sqr) %>%
  mutate(residuals = pred - price)

# Compare the cross-validated RMSE for the two models
houseprice_long %>%
  group_by(modeltype) %>% # group by modeltype
  summarize(rmse = sqrt(mean(residuals^2)))
```

```
## # A tibble: 2 x 2
##   modeltype     rmse
##       <chr>    <dbl>
## 1  pred_lin 74.29993
## 2  pred_sqr 63.69409
```

3.4 Dealing with Non-Linear Responses

First we will look at predicting the probability that an event occurs, based on a binary response yes/no. To do this we will use a logistic regression, so that the probabilities are in the 0 to 1 range. This builds a log-odds model which is simply a log transformed ratio that the probability occurs to the probability that it does not -

- for the model we have - glm(formula, data, family = binomial)
- for the prediction we have - predict(model, newdata = test, type = "response")

When assessing accuracy we can use a deviance calculation (pseudo R squared) or a chi-squared test. We can also use a Gain Curve or ROC curce.

We will estimate the probability that a sparrow survives a severe winter storm, based on physical characteristics of the sparrow. The dataset sparrow is loaded into your workspace. The outcome to be predicted is status.

```
# Load the sparrow data
sparrow <- readRDS("./files/SLinRRegression/sparrow.rds")

# Take a look at the data
head(sparrow)
```

```
##     status   age total_length wingspan weight beak_head humerus femur
## 1 Survived adult          154      241   24.5      31.2    0.69  0.67
## 2 Survived adult          160      252   26.9      30.8    0.74  0.71
```

```
## 3 Survived adult          155      243   26.9      30.6    0.73  0.70
## 4 Survived adult          154      245   24.3      31.7    0.74  0.69
## 5 Survived adult          156      247   24.1      31.5    0.71  0.71
## 6 Survived adult          161      253   26.5      31.8    0.78  0.74
##   legbone skull sternum
## 1    1.02  0.59    0.83
## 2    1.18  0.60    0.84
## 3    1.15  0.60    0.85
## 4    1.15  0.58    0.84
## 5    1.13  0.57    0.82
## 6    1.14  0.61    0.89
```

```
# sparrow is in the workspace
summary(sparrow)
```

```
##       status       age             total_length      wingspan
##  Perished:36   Length:87          Min.   :153.0   Min.   :236.0
##  Survived:51   Class :character   1st Qu.:158.0   1st Qu.:245.0
##                Mode  :character   Median :160.0   Median :247.0
##                                   Mean   :160.4   Mean   :247.5
##                                   3rd Qu.:162.5   3rd Qu.:251.0
##                                   Max.   :167.0   Max.   :256.0
##      weight       beak_head        humerus           femur
##  Min.   :23.2   Min.   :29.80   Min.   :0.6600   Min.   :0.6500
##  1st Qu.:24.7   1st Qu.:31.40   1st Qu.:0.7250   1st Qu.:0.7000
##  Median :25.8   Median :31.70   Median :0.7400   Median :0.7100
##  Mean   :25.8   Mean   :31.64   Mean   :0.7353   Mean   :0.7134
##  3rd Qu.:26.7   3rd Qu.:32.10   3rd Qu.:0.7500   3rd Qu.:0.7300
##  Max.   :31.0   Max.   :33.00   Max.   :0.7800   Max.   :0.7600
##     legbone          skull           sternum
##  Min.   :1.010   Min.   :0.5600   Min.   :0.7700
##  1st Qu.:1.110   1st Qu.:0.5900   1st Qu.:0.8300
##  Median :1.130   Median :0.6000   Median :0.8500
##  Mean   :1.131   Mean   :0.6032   Mean   :0.8511
##  3rd Qu.:1.160   3rd Qu.:0.6100   3rd Qu.:0.8800
##  Max.   :1.230   Max.   :0.6400   Max.   :0.9300
```

```
# Create the survived column
sparrow$survived <- sparrow$status == "Survived"

# Create the formula
(fmla <- as.formula("survived ~ total_length + weight + humerus"))
```

```
## survived ~ total_length + weight + humerus
```

```
# Fit the logistic regression model
sparrow_model <- glm(fmla, data = sparrow, family = binomial)

# Call summary
summary(sparrow_model)
```

```
##
## Call:
## glm(formula = fmla, family = binomial, data = sparrow)
##
## Deviance Residuals:
```

```
##     Min       1Q   Median       3Q      Max
## -2.1117  -0.6026   0.2871   0.6577   1.7082
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)   46.8813    16.9631   2.764 0.005715 **
## total_length  -0.5435     0.1409  -3.858 0.000115 ***
## weight        -0.5689     0.2771  -2.053 0.040060 *
## humerus       75.4610    19.1586   3.939 8.19e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 118.008  on 86  degrees of freedom
## Residual deviance:  75.094  on 83  degrees of freedom
## AIC: 83.094
##
## Number of Fisher Scoring iterations: 5
```

```
# Call glance
(perf <- broom::glance(sparrow_model))
```

```
##   null.deviance df.null    logLik      AIC      BIC deviance df.residual
## 1      118.0084      86 -37.54718 83.09436 92.95799 75.09436          83
```

```
# Calculate pseudo-R-squared
(pseudoR2 <- (1 - (perf$deviance / perf$null.deviance)))
```

```
## [1] 0.3636526
```

So our pseudo R squared so far is quite low at 0.36. Next we will predict with the model and show a gain curve. The gain curve show be as close to the ideal (the 'wizzard curve' or green line) as we can get.

```
# sparrow is in the workspace
summary(sparrow)
```

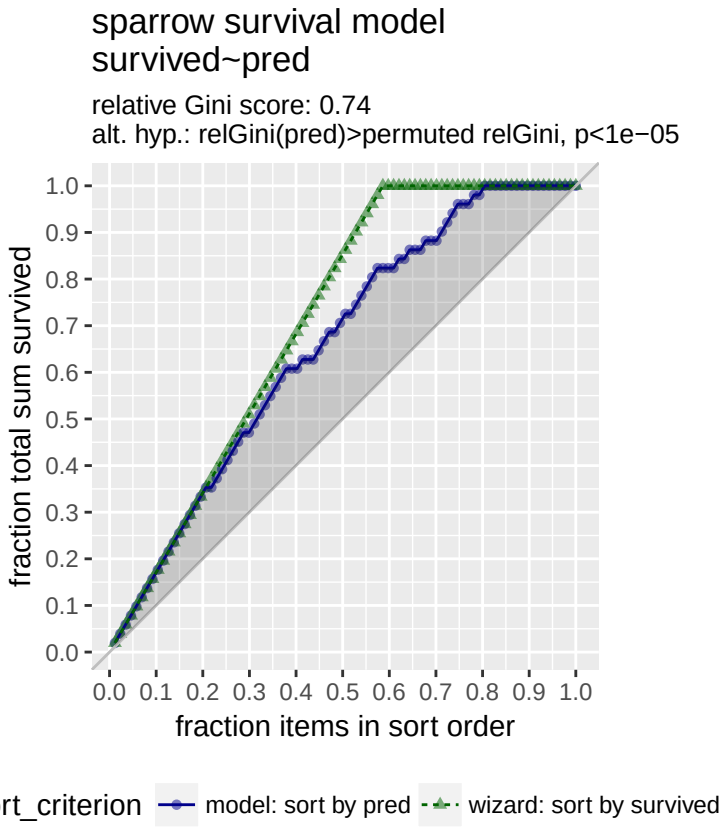
```
##       status       age             total_length      wingspan
##  Perished:36   Length:87          Min.   :153.0   Min.   :236.0
##  Survived:51   Class :character   1st Qu.:158.0   1st Qu.:245.0
##                Mode  :character   Median :160.0   Median :247.0
##                                   Mean   :160.4   Mean   :247.5
##                                   3rd Qu.:162.5   3rd Qu.:251.0
##                                   Max.   :167.0   Max.   :256.0
##      weight       beak_head        humerus           femur
##  Min.   :23.2   Min.   :29.80   Min.   :0.6600   Min.   :0.6500
##  1st Qu.:24.7   1st Qu.:31.40   1st Qu.:0.7250   1st Qu.:0.7000
##  Median :25.8   Median :31.70   Median :0.7400   Median :0.7100
##  Mean   :25.8   Mean   :31.64   Mean   :0.7353   Mean   :0.7134
##  3rd Qu.:26.7   3rd Qu.:32.10   3rd Qu.:0.7500   3rd Qu.:0.7300
##  Max.   :31.0   Max.   :33.00   Max.   :0.7800   Max.   :0.7600
##     legbone          skull           sternum        survived
##  Min.   :1.010   Min.   :0.5600   Min.   :0.7700   Mode :logical
##  1st Qu.:1.110   1st Qu.:0.5900   1st Qu.:0.8300   FALSE:36
##  Median :1.130   Median :0.6000   Median :0.8500   TRUE :51
##  Mean   :1.131   Mean   :0.6032   Mean   :0.8511
##  3rd Qu.:1.160   3rd Qu.:0.6100   3rd Qu.:0.8800
```

```
##  Max.   :1.230   Max.   :0.6400   Max.   :0.9300
```

```
# sparrow_model is in the workspace
summary(sparrow_model)
```

```
##
## Call:
## glm(formula = fmla, family = binomial, data = sparrow)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1117  -0.6026   0.2871   0.6577   1.7082
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)   46.8813    16.9631   2.764 0.005715 **
## total_length  -0.5435     0.1409  -3.858 0.000115 ***
## weight        -0.5689     0.2771  -2.053 0.040060 *
## humerus       75.4610    19.1586   3.939 8.19e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 118.008  on 86  degrees of freedom
## Residual deviance:  75.094  on 83  degrees of freedom
## AIC: 83.094
##
## Number of Fisher Scoring iterations: 5
```

```
# Make predictions
sparrow$pred <- predict(sparrow_model, type = "response")

# Look at gain curve
GainCurvePlot(sparrow, "pred", "survived", "sparrow survival model")
```



From the gain curve that the model follows the wizard curve for about the first 30% of the data, identifying about 45% of the surviving sparrows with only a few false positives.

3.4.1 Count data with poisson and quasipoisson regression

Predicting counts is a non linear problem because counts are restricted to being non negative and integers. To predict counts we do poisson or quasipoisson regression. It is a generalised linear model (GLM) in so much as it assumes the inputs are additive and linear with respect to the log of the outcome, we use family = "poisson" or "quasipoisson". We can use such models for things like predicting the number of website hits, the actual predict model we not predict an integer, but a rate per day. In poisson regression it is assumed the mean = variance, if this is not the case, we should use quasipoisson. More technically we could say that the event we are counting is Poisson distributed: the average count per unit is the same variance of the count, the same meaning that the mean and the variance should be of a similar order of magnitude. When the variance is much larger than the mean, the Poisson assumption doesn't apply and we should use quasipoisson.

NOTE: If the counts we are trying to predict are very large, regular regression may be appropriate method also.

In this exercise we will build a model to predict the number of bikes rented in an hour as a function of the weather, the type of day (holiday, working day, or weekend), and the time of day. You will train the model on data from the month of July.

The data frame has the columns:

- cnt: the number of bikes rented in that hour (the outcome)

- hr: the hour of the day (0-23, as a factor)
- holiday: TRUE/FALSE
- workingday: TRUE if neither a holiday nor a weekend, else FALSE
- weathersit: categorical, “Clear to partly cloudy”/“Light Precipitation”/“Misty”
- temp: normalized temperature in Celsius
- atemp: normalized “feeling” temperature in Celsius
- hum: normalized humidity
- windspeed: normalized windspeed
- instant: the time index – number of hours since beginning of data set (not a variable)
- mnth and yr: month and year indices (not variables)

We fit a quasipoisson model as the mean and variance are quite different, as calculated below. As with a logistic model, you hope for a pseudo-R2 near to 1.

```
# Load the bikes data
load("./files/SLinRRegression/Bikes.RData")
outcome <- "cnt"
vars <- names(bikesJuly)[1:8]

str(bikesJuly)
```

```
## 'data.frame':    744 obs. of  12 variables:
##  $ hr        : Factor w/ 24 levels "0","1","2","3",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ holiday   : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ workingday: logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ weathersit: chr  "Clear to partly cloudy" "Clear to partly cloudy" "Clear to partly cloudy" "Clea
##  $ temp      : num  0.76 0.74 0.72 0.72 0.7 0.68 0.7 0.74 0.78 0.82 ...
##  $ atemp     : num  0.727 0.697 0.697 0.712 0.667 ...
##  $ hum       : num  0.66 0.7 0.74 0.84 0.79 0.79 0.79 0.7 0.62 0.56 ...
##  $ windspeed : num  0 0.1343 0.0896 0.1343 0.194 ...
##  $ cnt       : int  149 93 90 33 4 10 27 50 142 219 ...
##  $ instant   : int  13004 13005 13006 13007 13008 13009 13010 13011 13012 13013 ...
##  $ mnth      : int  7 7 7 7 7 7 7 7 7 7 ...
##  $ yr        : int  1 1 1 1 1 1 1 1 1 1 ...
```

```
# The outcome column
outcome
```

```
## [1] "cnt"
```

```
# The inputs to use
vars
```

```
## [1] "hr"         "holiday"    "workingday" "weathersit" "temp"
## [6] "atemp"      "hum"        "windspeed"
```

```
# Create the formula string for bikes rented as a function of the inputs
(fmla <- paste(outcome, "~", paste(vars, collapse = " + ")))
```

```
## [1] "cnt ~ hr + holiday + workingday + weathersit + temp + atemp + hum + windspeed"
```

```
# Calculate the mean and variance of the outcome
(mean_bikes <- mean(bikesJuly$cnt))
```

```
## [1] 273.6653
```

```
(var_bikes <- var(bikesJuly$cnt))
```

```
## [1] 45863.84
```

```
# Fit the model
bike_model <- glm(fmla, data = bikesJuly, family = quasipoisson)

# Call glance
(perf <- broom::glance(bike_model))
```

```
##   null.deviance df.null logLik AIC BIC deviance df.residual
## 1      133364.9     743     NA  NA  NA  28774.9         712
```

```
# Calculate pseudo-R-squared
(pseudoR2 <- (1 - (perf$deviance / perf$null.deviance)))
```

```
## [1] 0.7842393
```

Next we will use the model you built in the previous exercise to make predictions for the month of August. The data set bikesAugust has the same columns as bikesJuly. You must specify type = "response" with predict() when predicting counts from a glm poisson or quasipoisson model.

```
# bike_model is in the workspace
summary(bike_model)
```

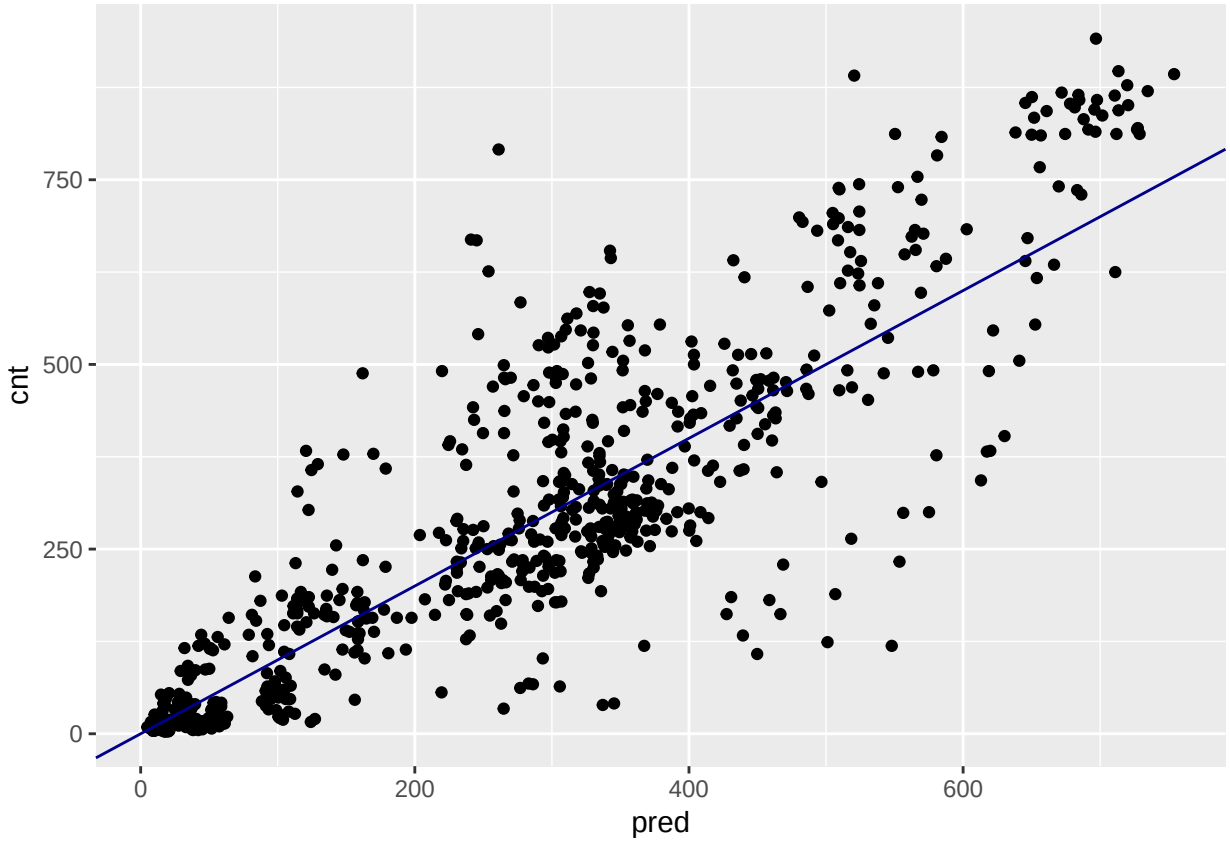
```
##
## Call:
## glm(formula = fmla, family = quasipoisson, data = bikesJuly)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -21.6117   -4.3121   -0.7223    3.5507   16.5079
##
## Coefficients:
##                              Estimate Std. Error t value Pr(>|t|)
## (Intercept)                  5.934986   0.439027  13.519  < 2e-16 ***
## hr1                         -0.580055   0.193354  -3.000 0.002794 **
## hr2                         -0.892314   0.215452  -4.142 3.86e-05 ***
## hr3                         -1.662342   0.290658  -5.719 1.58e-08 ***
## hr4                         -2.350204   0.393560  -5.972 3.71e-09 ***
## hr5                         -1.084289   0.230130  -4.712 2.96e-06 ***
## hr6                          0.211945   0.156476   1.354 0.176012
## hr7                          1.211135   0.132332   9.152  < 2e-16 ***
## hr8                          1.648361   0.127177  12.961  < 2e-16 ***
## hr9                          1.155669   0.133927   8.629  < 2e-16 ***
## hr10                         0.993913   0.137096   7.250 1.09e-12 ***
## hr11                         1.116547   0.136300   8.192 1.19e-15 ***
## hr12                         1.282685   0.134769   9.518  < 2e-16 ***
## hr13                         1.273010   0.135872   9.369  < 2e-16 ***
## hr14                         1.237721   0.136386   9.075  < 2e-16 ***
## hr15                         1.260647   0.136144   9.260  < 2e-16 ***
```

```
## hr16                           1.515893   0.132727  11.421  < 2e-16 ***
## hr17                           1.948404   0.128080  15.212  < 2e-16 ***
## hr18                           1.893915   0.127812  14.818  < 2e-16 ***
## hr19                           1.669277   0.128471  12.993  < 2e-16 ***
## hr20                           1.420732   0.131004  10.845  < 2e-16 ***
## hr21                           1.146763   0.134042   8.555  < 2e-16 ***
## hr22                           0.856182   0.138982   6.160 1.21e-09 ***
## hr23                           0.479197   0.148051   3.237 0.001265 **
## holidayTRUE                    0.201598   0.079039   2.551 0.010961 *
## workingdayTRUE                 0.116798   0.033510   3.485 0.000521 ***
## weathersitLight Precipitation -0.214801   0.072699  -2.955 0.003233 **
## weathersitMisty               -0.010757   0.038600  -0.279 0.780572
## temp                          -3.246001   1.148270  -2.827 0.004833 **
## atemp                          2.042314   0.953772   2.141 0.032589 *
## hum                           -0.748557   0.236015  -3.172 0.001581 **
## windspeed                      0.003277   0.148814   0.022 0.982439
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for quasipoisson family taken to be 38.98949)
##
##     Null deviance: 133365  on 743  degrees of freedom
## Residual deviance:  28775  on 712  degrees of freedom
## AIC: NA
##
## Number of Fisher Scoring iterations: 5
```

```
# Make predictions on August data
bikesAugust$pred  <- predict(bike_model, type = "response", newdata = bikesAugust)

# Calculate the RMSE
bikesAugust %>%
  mutate(residual = pred - cnt) %>%
  summarize(rmse  = sqrt(mean(residual^2)))
```

```
##       rmse
## 1 112.5815
```

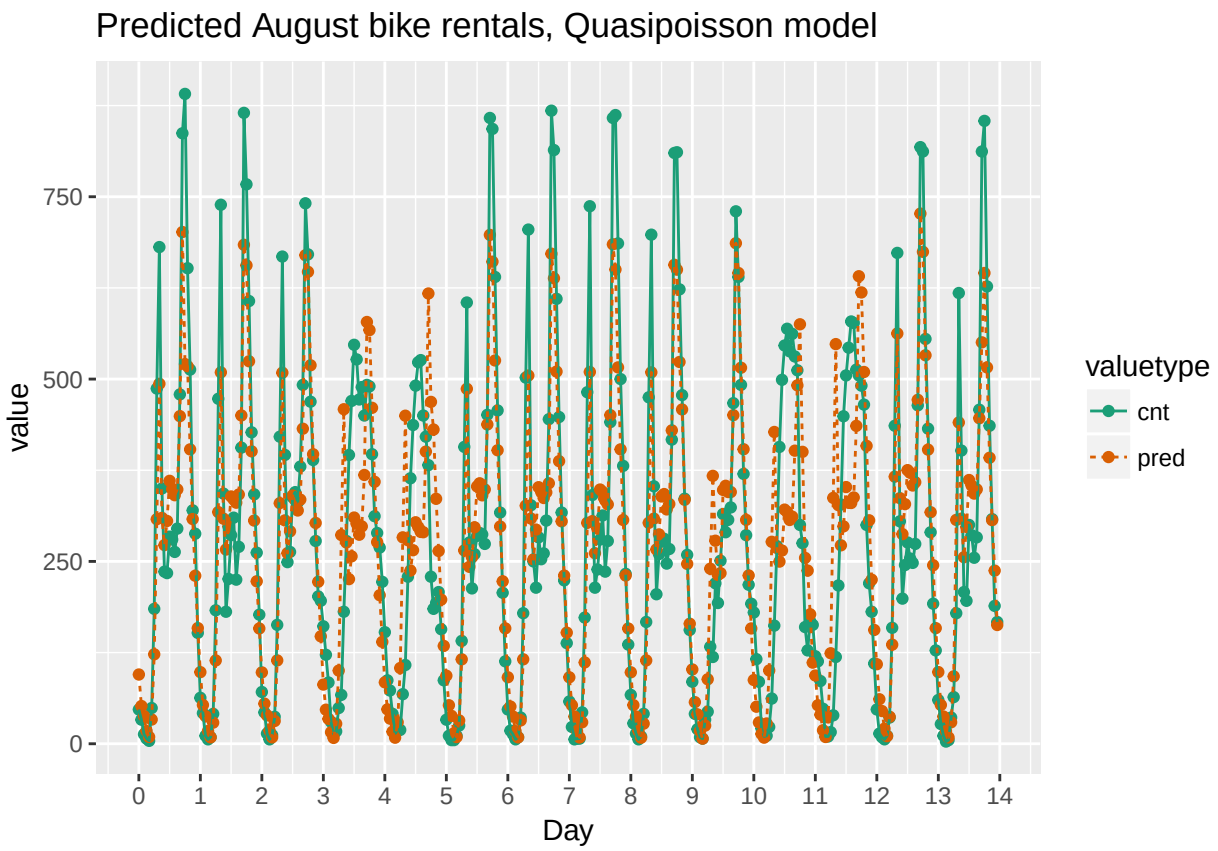
```
# Plot predictions vs cnt (pred on x-axis)
ggplot(bikesAugust, aes(x = pred, y = cnt)) +
  geom_point() +
  geom_abline(color = "darkblue")
```



As the bike rental data is time series data, you might be interested in how the model performs as a function of time. Next we will compare the predictions and actual rentals on an hourly basis, for the first 14 days of August.

To create the plot we use the function tidyr::gather() to consolidate the predicted and actual values from bikesAugust in a single column.

```
library(tidyr)

# Plot predictions and cnt by date/time
Quasipoissonmodel <- bikesAugust %>%
  # set start to 0, convert unit to days
  mutate(instant = (instant - min(instant))/24) %>%
  # gather cnt and pred into a value column
  gather(key = valuetype, value = value, cnt, pred) %>%
  filter(instant < 14) %>% # restric to first 14 days
  # plot value by instant
  ggplot(aes(x = instant, y = value, color = valuetype, linetype = valuetype)) +
  geom_point() +
  geom_line() +
  scale_x_continuous("Day", breaks = 0:14, labels = 0:14) +
  scale_color_brewer(palette = "Dark2") +
  ggtitle("Predicted August bike rentals, Quasipoisson model")
Quasipoissonmodel
```



Using the chart it appears that this model mostly identifies the slow and busy hours of the day, although it often underestimates peak demand.

3.4.2 Generalised Additive Model (GAM)

GAM can be used to automatically learn input variable transformations. In GAM, it depends on unknown smoothed functions of the input variables. A GAM learns what best fits the data e.g. quadratic, cubic and so on. We use the mgcv package for GAM models, which has similar functions at the GLM models we saw previously, including a family variable where gaussian (default) is used for regular regression, binomial for probabilities and poisson/quasipoisson for counts. GAMS are prone to overfitting, so are best used on large sets of data. If we want to model the data as a non-linear relationship we use the s notation e.g. anx ~ s(hassles) - this is best done with 10 or more unique values, since a spline function is fitted, so not good for categorical data.

NOTE: GAM is useful for when you don't have the domain knowledge to determine the best model type so is used as a proxy of the 'best model' in the absence of this knowledge.

If you do have categorical variables, you can still use them in GAM, but you don't specify the s function for those, e.g. if diet and sex are categorical, but age and BMI are continous, we would have:

$$\text{Wtloss} \sim \text{Diet} + \text{Sex} + s(\text{Age}) + s(\text{BMI})$$

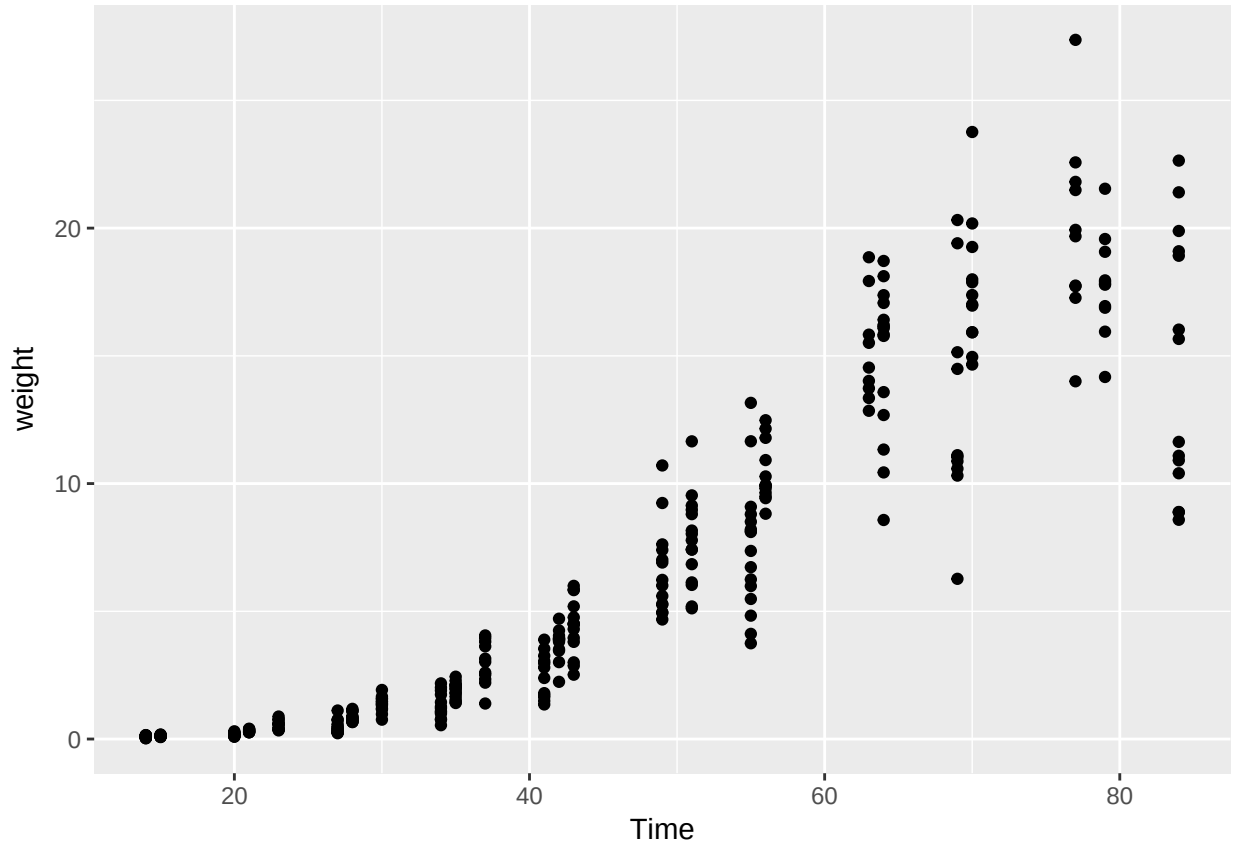
Next we will model the average leaf weight on a soybean plant as a function of time (after planting). As you will see, the soybean plant doesn't grow at a steady rate, but rather has a "growth spurt" that eventually tapers off. Hence, leaf weight is not well described by a linear model.

```
# Load the Soybean data
load("./files/SLinRRegression/Soybean.RData")
```

```
# soybean_train is in the workspace
summary(soybean_train)
```

```
##       Plot     Variety   Year          Time           weight
##  1988F6 : 10   F:161   1988:124   Min.   :14.00   Min.   : 0.0290
##  1988F7 :  9   P:169   1989:102   1st Qu.:27.00   1st Qu.: 0.6663
##  1988P1 :  9           1990:104   Median :42.00   Median : 3.5233
##  1988P8 :  9                      Mean   :43.56   Mean   : 6.1645
##  1988P2 :  9                      3rd Qu.:56.00   3rd Qu.:10.3808
##  1988F3 :  8                      Max.   :84.00   Max.   :27.3700
##  (Other):276
```

```
# Plot weight vs Time (Time on x axis)
ggplot(soybean_train, aes(x = Time, y = weight)) +
  geom_point()
```




```
# Load the package mgcv
library(mgcv)
```

```
## Loading required package: nlme
```

```
##
## Attaching package: 'nlme'
```

```
## The following object is masked from 'package:dplyr':
##
##     collapse
```

```
## This is mgcv 1.8-20. For overview type 'help("mgcv-package")'.
```

```
# Create the gam/non-linear formula
(fmla.gam <- as.formula("weight ~ s(Time)"))
```

```
## weight ~ s(Time)
```

```
# Create the linear formula
(fmla.lin <- as.formula("weight ~ Time"))
```

```
## weight ~ Time
```

```
# Fit the GAM Model
model.gam <- gam(fmla.gam, family = gaussian, data = soybean_train)

# Create the linear formula
model.lin <- lm(formula = fmla.lin, data = soybean_train)

# Call summary() on model.lin and look for R-squared
summary(model.lin)
```

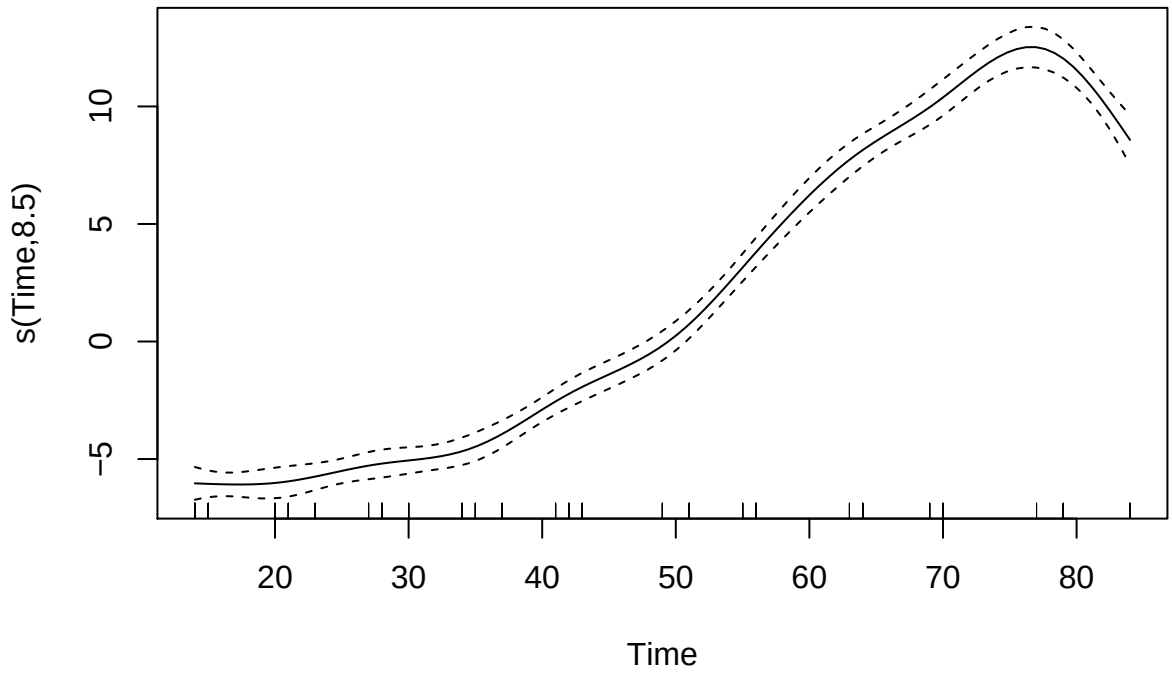
```
##
## Call:
## lm(formula = fmla.lin, data = soybean_train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -9.3933 -1.7100 -0.3909  1.9056 11.4381
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -6.559283   0.358527  -18.30   <2e-16 ***
## Time         0.292094   0.007444   39.24   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.778 on 328 degrees of freedom
## Multiple R-squared:  0.8244, Adjusted R-squared:  0.8238
## F-statistic:  1540 on 1 and 328 DF,  p-value: < 2.2e-16
```

```
# Call summary() on model.gam and look for R-squared
summary(model.gam)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## weight ~ s(Time)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   6.1645     0.1143   53.93   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
```

```
##           edf Ref.df     F p-value
## s(Time) 8.495   8.93 338.2  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.902   Deviance explained = 90.4%
## GCV = 4.4395  Scale est. = 4.3117    n = 330
```

```
# Call plot() on model.gam
plot(model.gam)
```



So we see that the non-linear GAM model fits the data better, with a higher R squared measurement than the linear model.

Next we will predict with both the linear and gam model. As GAM models return a matrix for predictions, we use as.numeric to convert the output.

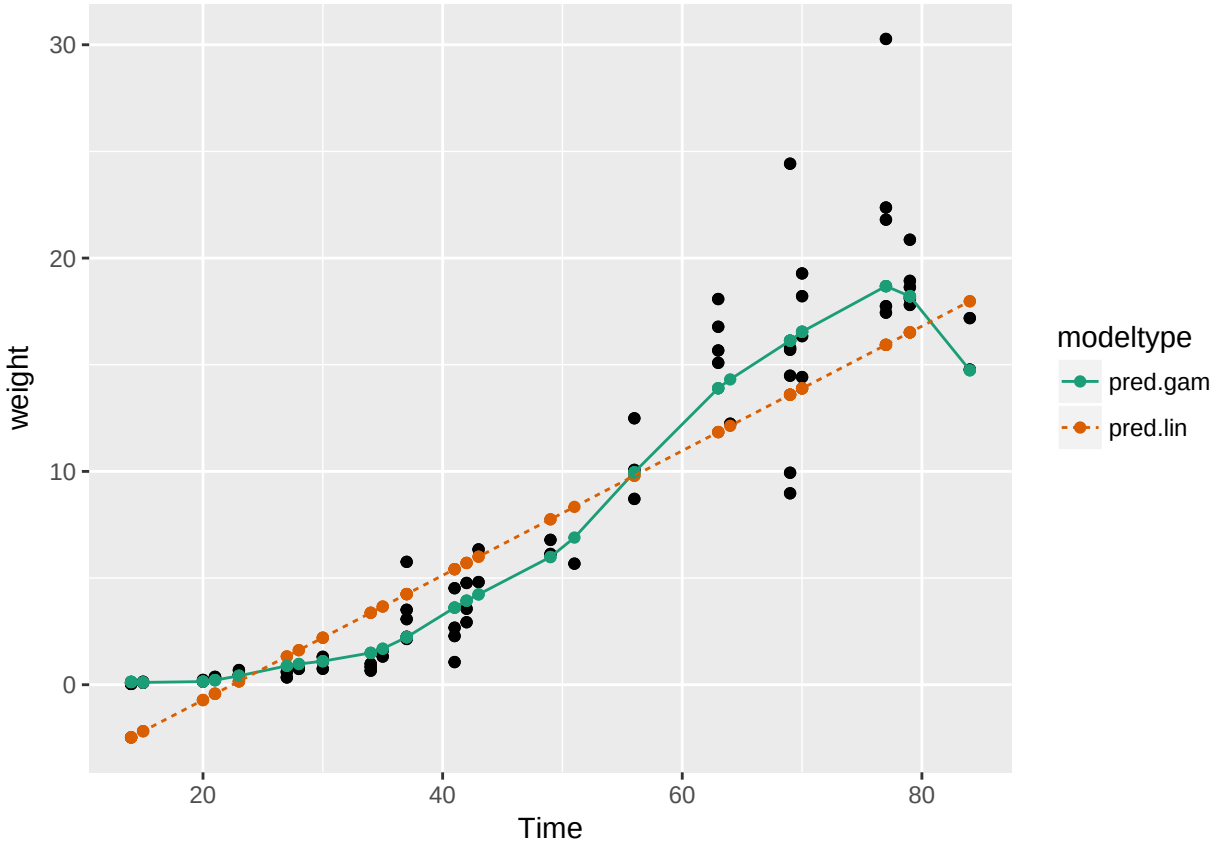
```
# soybean_test is in the workspace
summary(soybean_test)
```

```
##       Plot    Variety   Year         Time           weight
##  1988F8 : 4   F:43    1988:32   Min.   :14.00   Min.   : 0.0380
##  1988P7 : 4   P:39    1989:26   1st Qu.:23.00   1st Qu.: 0.4248
##  1989F8 : 4           1990:24   Median :41.00   Median : 3.0025
##  1990F8 : 4                     Mean   :44.09   Mean   : 7.1576
##  1988F4 : 3                     3rd Qu.:69.00   3rd Qu.:15.0113
##  1988F2 : 3                     Max.   :84.00   Max.   :30.2717
##  (Other):60
```

```
# Get predictions from linear model
soybean_test$pred.lin <- predict(model.lin, newdata = soybean_test)

# Get predictions from gam model
soybean_test$pred.gam <- as.numeric(predict(model.gam, newdata = soybean_test))

# Gather the predictions into a "long" dataset
soybean_long <- soybean_test %>%
  gather(key = modeltype, value = pred, pred.lin, pred.gam)

# Calculate the rmse
soybean_long %>%
  mutate(residual = weight - pred) %>%     # residuals
  group_by(modeltype) %>%                  # group by modeltype
  summarize(rmse = sqrt(mean(residual^2))) # calculate the RMSE
```

```
## # A tibble: 2 x 2
##   modeltype     rmse
##       <chr>    <dbl>
## 1  pred.gam 2.286451
## 2  pred.lin 3.190995
```

```
# Compare the predictions against actual weights on the test data
soybean_long %>%
  ggplot(aes(x = Time)) +                        # the column for the x axis
  geom_point(aes(y = weight)) +                  # the y-column for the scatterplot
  geom_point(aes(y = pred, color = modeltype)) +   # the y-column for the point-and-line plot
  geom_line(aes(y = pred, color = modeltype, linetype = modeltype)) + # the y-column for the point-and-
  scale_color_brewer(palette = "Dark2")
```



Oberving the plot we can see that the GAM learns the non-linear growth function of the soybean plants, including the fact that weight is never negative, whereas the linear model intercepts below 0 i.e. a negative size.

3.5 Tree-Based Methods

Tree based models can be used for both regression and classification models. Decision Trees say 'if a AND b AND c THEN y'. We can therefore model non-linear models and multiplicative relationships - what is the affect of this AND that when combined together.

We can use RMSE as a measure of accuracy of the model. The challenge with tree models is that they are interested in the model space as a whole, splitting this in to regions. Trees can have difficulting fitting linear relationships, so linear models can be better for linear relationships. Trees also have difficulty with variables that change quickly and continously.

We can adjust the tree depth, but there is a risk of overfitting (too deep/complex) or underfitting (too shallow/coarse).

An ensemble model can be built combining different trees or indeed different models together, which will usually have the outcome of being better than a sinlge tree and less prone to overfitting, but at the loss of interpretability. Two such examples of ensemble models are random forests and gradient boosted trees.

3.5.1 Random Forests

One example of an ensemble approach is a random forest, building multiple trees from the training data. We build slightly different trees each time to add diversity to the model, by averaging the results of multiple models together to reduce the degree of overfitting. To build a random forest we perform the following

1. Draw bootstrapped sample from training data
2. For each sample grow a tree
   - At each node, pick best variable to split on (from a random subset of all variables)
   - Continue until tree is grown
3. To score a datum, evaluate it with all the trees and average the results.

We can use the ranger package to fit random forests. If the outcome is numeric, ranger will automatically do regression rather than classification. The default is for 500 trees, a reccommended minimum is 200. The value respect.unordered.factors will handle categorical values, set it to "order" if using cateogrical values, which will convert the values to numeric values.

The measures of accuracy are R squared and OOB (Out of Bag or out of sample performance). You should still evaluate the model further using test data.

In this exercise you will again build a model to predict the number of bikes rented in an hour as a function of the weather, the type of day (holiday, working day, or weekend), and the time of day. You will train the model on data from the month of July.

You will use the ranger package to fit the random forest model. For this exercise, the key arguments to the ranger() call are:

- formula
- data
- num.trees: the number of trees in the forest.
- respect.unordered.factors : Specifies how to treat unordered factor variables. We recommend setting this to "order" for regression.
- seed: because this is a random algorithm, you will set the seed to get reproducible results Since there are a lot of input variables, for convenience we will specify the outcome and the inputs in the variables outcome and vars, and use paste() to assemble a string representing the model formula.

```
# bikesJuly is in the workspace
str(bikesJuly)
```

```
## 'data.frame':    744 obs. of  12 variables:
##  $ hr        : Factor w/ 24 levels "0","1","2","3",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ holiday   : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ workingday: logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ weathersit: chr  "Clear to partly cloudy" "Clear to partly cloudy" "Clear to partly cloudy" "Clear
##  $ temp      : num  0.76 0.74 0.72 0.72 0.7 0.68 0.7 0.74 0.78 0.82 ...
##  $ atemp     : num  0.727 0.697 0.697 0.712 0.667 ...
##  $ hum       : num  0.66 0.7 0.74 0.84 0.79 0.79 0.79 0.7 0.62 0.56 ...
##  $ windspeed : num  0 0.1343 0.0896 0.1343 0.194 ...
##  $ cnt       : int  149 93 90 33 4 10 27 50 142 219 ...
##  $ instant   : int  13004 13005 13006 13007 13008 13009 13010 13011 13012 13013 ...
##  $ mnth      : int  7 7 7 7 7 7 7 7 7 7 ...
##  $ yr        : int  1 1 1 1 1 1 1 1 1 1 ...
```

```
# Random seed to reproduce results
seed <- 423563

# The outcome column
(outcome <- "cnt")
```

```
## [1] "cnt"
```

```
# The input variables
(vars <- c("hr", "holiday", "workingday", "weathersit", "temp", "atemp", "hum", "windspeed"))
```

```
## [1] "hr"         "holiday"    "workingday" "weathersit" "temp"
## [6] "atemp"      "hum"        "windspeed"
```

```
# Create the formula string for bikes rented as a function of the inputs
(fmla <- paste("cnt", "~", paste(vars, collapse = " + ")))
```

```
## [1] "cnt ~ hr + holiday + workingday + weathersit + temp + atemp + hum + windspeed"
```

```
# Load the package ranger
library(ranger)

# Fit and print the random forest model
(bike_model_rf <- ranger(fmla, # formula
                         bikesJuly, # data
                         num.trees = 500,
                         respect.unordered.factors = "order",
                         seed = seed))
```

```
## Ranger result
##
## Call:
##  ranger(fmla, bikesJuly, num.trees = 500, respect.unordered.factors = "order",      seed = seed)
##
## Type:                             Regression
## Number of trees:                  500
## Sample size:                      744
## Number of independent variables:  8
## Mtry:                             2
## Target node size:                 5
## Variable importance mode:         none
## OOB prediction error (MSE):       8230.568
## R squared (OOB):                  0.8205434
```

Here we see the R squared is very high - around 82%.

Next we will use the model that you fit in the previous exercise to predict bike rentals for the month of August.

The predict() function for a ranger model produces a list. One of the elements of this list is predictions, a vector of predicted values. You can access predictions with the $ notation for accessing named elements of a list:

- predict(model, data)$predictions

```
# bikesAugust is in the workspace
str(bikesAugust)
```

```
## 'data.frame':    744 obs. of  13 variables:
##  $ hr        : Factor w/ 24 levels "0","1","2","3",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ holiday   : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ workingday: logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
##  $ weathersit: chr  "Clear to partly cloudy" "Clear to partly cloudy" "Clear to partly cloudy" "Clear
##  $ temp      : num  0.68 0.66 0.64 0.64 0.64 0.64 0.64 0.64 0.66 0.68 ...
##  $ atemp     : num  0.636 0.606 0.576 0.576 0.591 ...
```

```
##  $ hum       : num  0.79 0.83 0.83 0.83 0.78 0.78 0.78 0.83 0.78 0.74 ...
##  $ windspeed : num  0.1642 0.0896 0.1045 0.1045 0.1343 ...
##  $ cnt       : int  47 33 13 7 4 49 185 487 681 350 ...
##  $ instant   : int  13748 13749 13750 13751 13752 13753 13754 13755 13756 13757 ...
##  $ mnth      : int  8 8 8 8 8 8 8 8 8 8 ...
##  $ yr        : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ pred      : num  94.96 51.74 37.98 17.58 9.36 ...
```

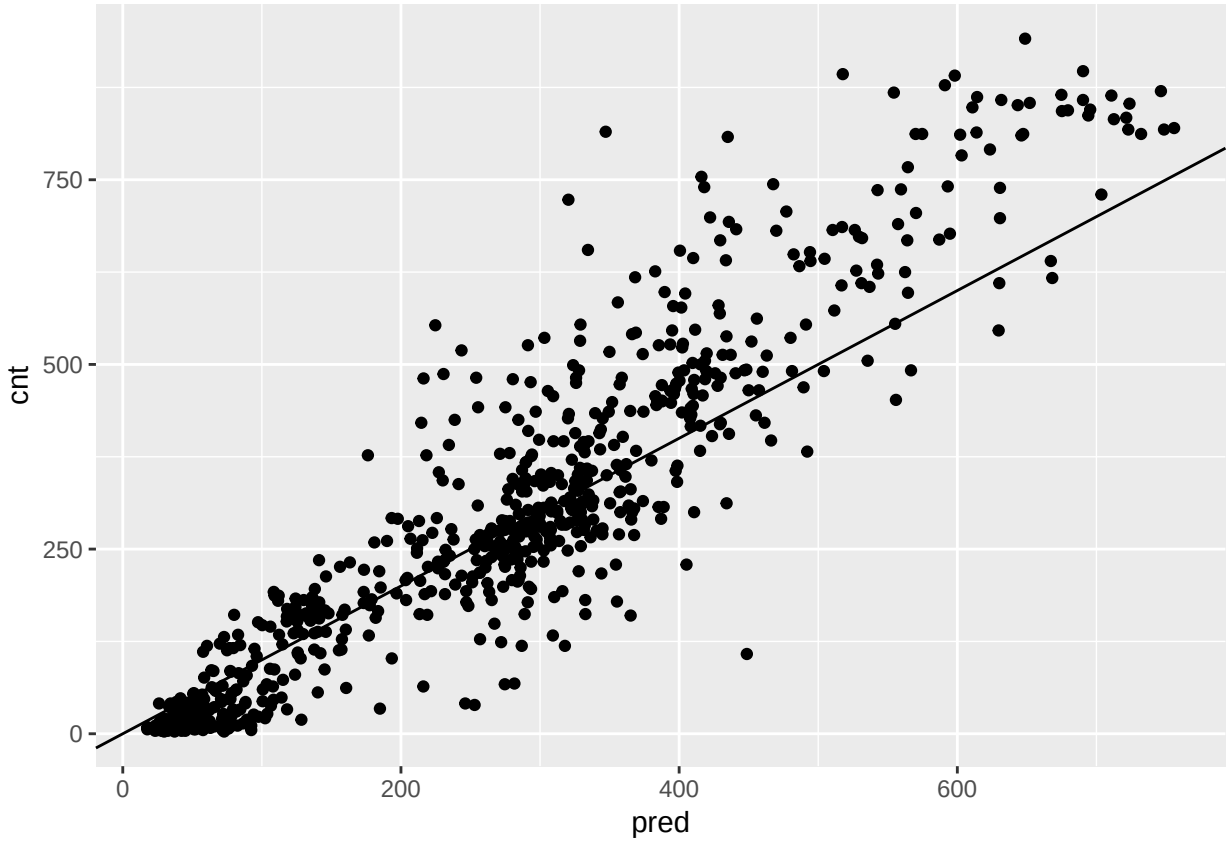
```
# bike_model_rf is in the workspace
bike_model_rf
```

```
## Ranger result
##
## Call:
##  ranger(fmla, bikesJuly, num.trees = 500, respect.unordered.factors = "order",      seed = seed)
##
## Type:                             Regression
## Number of trees:                  500
## Sample size:                      744
## Number of independent variables:  8
## Mtry:                             2
## Target node size:                 5
## Variable importance mode:         none
## OOB prediction error (MSE):       8230.568
## R squared (OOB):                  0.8205434
```

```
# Make predictions on the August data
bikesAugust$pred <- predict(bike_model_rf, bikesAugust)$predictions

# Calculate the RMSE of the predictions
bikesAugust %>%
  mutate(residual = cnt - pred)  %>% # calculate the residual
  summarize(rmse  = sqrt(mean(residual^2)))      # calculate rmse
```

```
##       rmse
## 1 97.18347
```

```
# Plot actual outcome vs predictions (predictions on x-axis)
ggplot(bikesAugust, aes(x = pred, y = cnt)) +
  geom_point() +
  geom_abline()
```

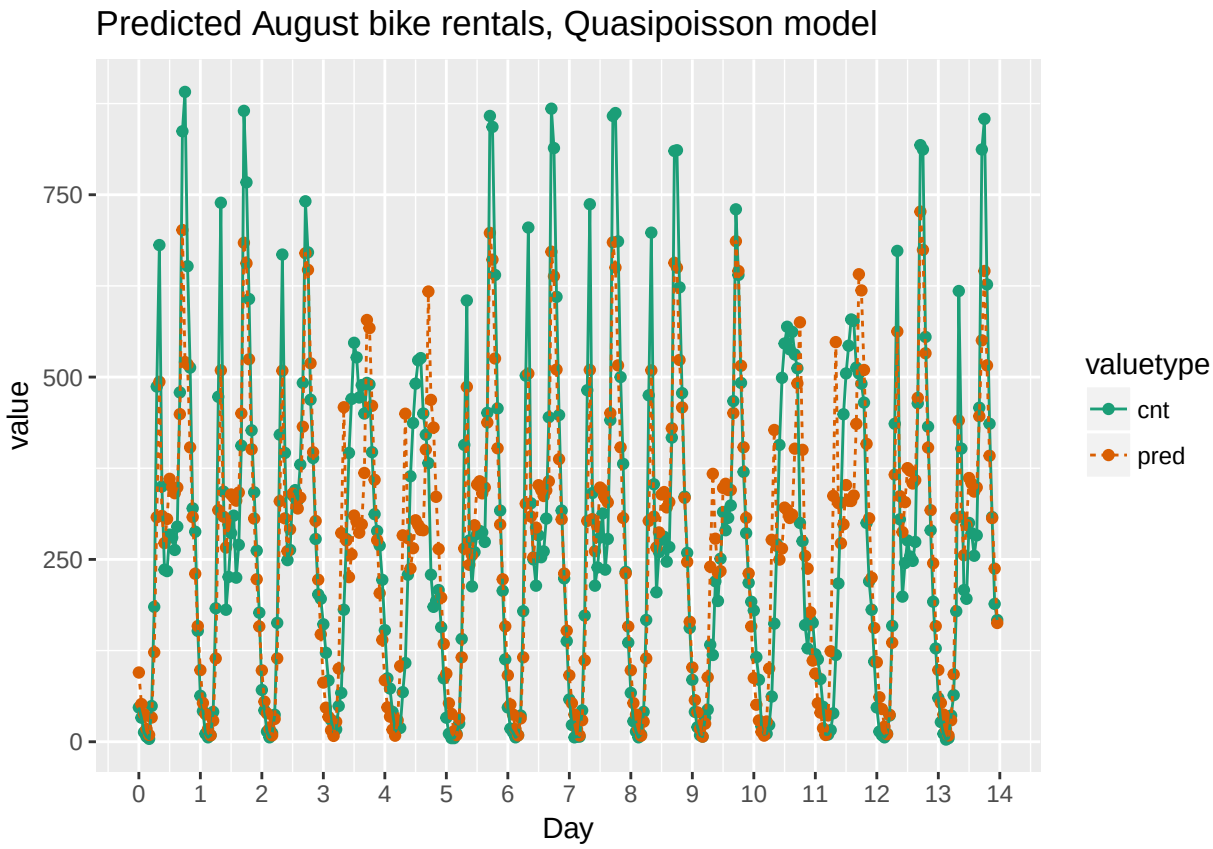


This random forest model outperforms the poisson count model on the same data; it is discovering more complex non-linear or non-additive relationships in the data.

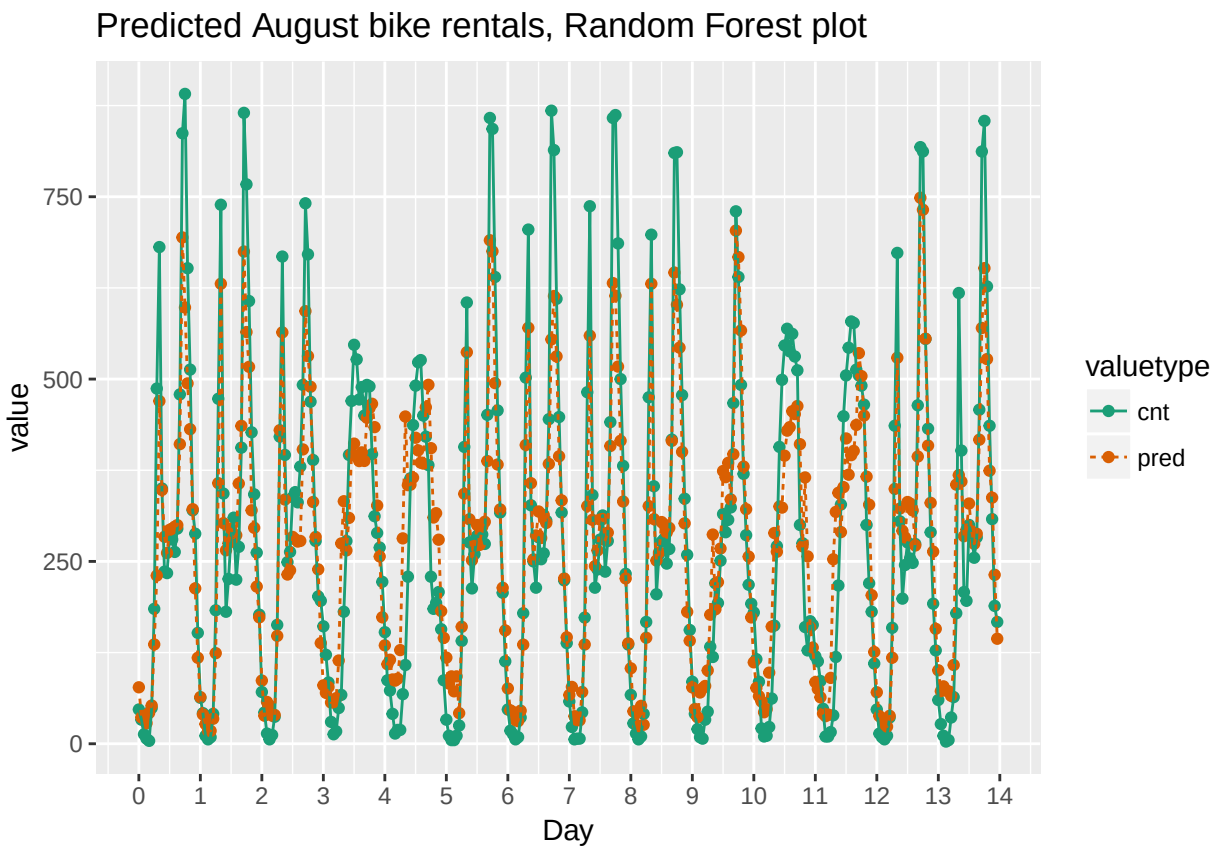
Next we will visualize the random forest model's August predictions as a function of time. We can compare the corresponding plot from the quasipoisson model that we built previously.

Recall that the quasipoisson model mostly identified the pattern of slow and busy hours in the day, but it somewhat underestimated peak demands. You would like to see how the random forest model compares.

```
Quasipoissonmodel
```




```
# Plot predictions and cnt by date/time
randomforest_plot <- bikesAugust %>%
  mutate(instant = (instant - min(instant))/24) %>%  # set start to 0, convert unit to days
  gather(key = valuetype, value = value, cnt, pred) %>%
  filter(instant < 14) %>% # first two weeks
  ggplot(aes(x = instant, y = value, color = valuetype, linetype = valuetype)) +
  geom_point() +
  geom_line() +
  scale_x_continuous("Day", breaks = 0:14, labels = 0:14) +
  scale_color_brewer(palette = "Dark2") +
  ggtitle("Predicted August bike rentals, Random Forest plot")
randomforest_plot
```



The random forest model captured the day-to-day variations in peak demand better than the quasipoisson model, but it still underestmates peak demand, and also overestimates minimum demand. So there is still room for improvement.

3.5.2 One-hot encoding

For modelling purposes, we need to convert categorical variables to indicator variables. Some R packages do this automatically, but some non-native R packages, such as the xgboost package does not, as it is a non-R package (it's C with a R api). So, these categorical variables need to be converted to numeric ones. We can use the vtreat package.

- DesignTreatmentsZ() to design a treatment plan from the training data, then
- prepare() to created "clean" data
- all numerical
- no missing values
- use prepare() with treatment plan for all future data

In this exercise you will use vtreat to one-hot-encode a categorical variable on a small example. vtreat creates a treatment plan to transform categorical variables into indicator variables (coded "lev"), and to clean bad values out of numerical variables (coded "clean").

To design a treatment plan use the function designTreatmentsZ() e.g.

```
treatplan <- designTreatmentsZ(data, varlist)
```

data: the original training data frame varlist: a vector of input variables to be treated (as strings). designTreatmentsZ() returns a list with an element scoreFrame: a data frame that includes the names and types of the new variables:

scoreFrame <- treatplan %>% magrittr::use_series(scoreFrame) %>% select(varName, origName, code)

varName: the name of the new treated variable origName: the name of the original variable that the treated variable comes from code: the type of the new variable. "clean": a numerical variable with no NAs or NaNs "lev": an indicator variable for a specific level of the original categorical variable.

(magrittr::use_series() is an alias for $ that you can use in pipes.)

For these exercises, we want varName where code is either "clean" or "lev":

newvarlist <- scoreFrame %>% filter(code %in% c("clean", "lev") %>% magrittr::use_series(varName)

To transform the data set into all numerical and one-hot-encoded variables, use prepare():

data.treat <- prepare(treatplan, data, varRestrictions = newvarlist)

treatplan: the treatment plan data: the data frame to be treated varRestrictions: the variables desired in the treated data

```
# Create the dataframe for cleaning
color <- c("b", "r", "r", "r", "r", "b", "r", "g", "b", "b")
size <- c(13, 11, 15, 14, 13, 11, 9, 12, 7, 12)
popularity <- c(1.0785088, 1.3956245, 0.9217988, 1.2025453, 1.0838662, 0.8043527, 1.1035440, 0.8746332,
dframe <- cbind(color, size, popularity)
dframe <- as.data.frame((dframe))

# dframe is in the workspace
dframe
```

```
##    color size popularity
## 1      b   13  1.0785088
## 2      r   11  1.3956245
## 3      r   15  0.9217988
## 4      r   14  1.2025453
## 5      r   13  1.0838662
## 6      b   11  0.8043527
## 7      r    9   1.103544
## 8      g   12  0.8746332
## 9      b    7  0.6947058
## 10     b   12  0.8832502
```

```
# Create and print a vector of variable names
(vars <- c("color", "size"))
```

```
## [1] "color" "size"
```

```
# Load the package vtreat
library(vtreat)

# Create the treatment plan
treatplan <- designTreatmentsZ(dframe, vars)
```

```
## [1] "designing treatments Wed Apr 25 03:58:44 2018"
## [1] "designing treatments Wed Apr 25 03:58:44 2018"
## [1] " have level statistics Wed Apr 25 03:58:44 2018"
## [1] "design var color Wed Apr 25 03:58:44 2018"
## [1] "design var size Wed Apr 25 03:58:44 2018"
## [1] " scoring treatments Wed Apr 25 03:58:44 2018"
## [1] "have treatment plan Wed Apr 25 03:58:44 2018"
```

```
# Examine the scoreFrame
(scoreFrame <- treatplan %>%
    magrittr::use_series(scoreFrame) %>%
    select(varName, origName, code))
```

```
##          varName origName code
## 1  color_lev_x.b    color  lev
## 2  color_lev_x.g    color  lev
## 3  color_lev_x.r    color  lev
## 4     color_catP    color catP
## 5  size_lev_x.11     size  lev
## 6  size_lev_x.12     size  lev
## 7  size_lev_x.13     size  lev
## 8  size_lev_x.14     size  lev
## 9  size_lev_x.15     size  lev
## 10  size_lev_x.7     size  lev
## 11  size_lev_x.9     size  lev
## 12     size_catP     size catP
```

```
# We only want the rows with codes "clean" or "lev"
(newvars <- scoreFrame %>%
    filter(code %in% c("clean", "lev")) %>%
    magrittr::use_series(varName))
```

```
##  [1] "color_lev_x.b" "color_lev_x.g" "color_lev_x.r" "size_lev_x.11"
##  [5] "size_lev_x.12" "size_lev_x.13" "size_lev_x.14" "size_lev_x.15"
##  [9] "size_lev_x.7"  "size_lev_x.9"
```

```
# Create the treated training data
(dframe.treat <- prepare(treatplan, dframe, varRestriction = newvars))
```

```
##    color_lev_x.b color_lev_x.g color_lev_x.r size_lev_x.11 size_lev_x.12
## 1              1             0             0             0             0
## 2              0             0             1             1             0
## 3              0             0             1             0             0
## 4              0             0             1             0             0
## 5              0             0             1             0             0
## 6              1             0             0             1             0
## 7              0             0             1             0             0
## 8              0             1             0             0             1
## 9              1             0             0             0             0
## 10             1             0             0             0             1
##    size_lev_x.13 size_lev_x.14 size_lev_x.15 size_lev_x.7 size_lev_x.9
## 1              1             0             0            0            0
## 2              0             0             0            0            0
## 3              0             0             1            0            0
## 4              0             1             0            0            0
## 5              1             0             0            0            0
## 6              0             0             0            0            0
## 7              0             0             0            0            1
## 8              0             0             0            0            0
## 9              0             0             0            1            0
## 10             0             0             0            0            0
```

WE have have successfully one-hot-encoded categorical data. The new indicator variables have '*lev*' in their names, and the new cleaned continuous variables have '_clean' in their names. The treated data is all

numerical, with no missing values, and is suitable for use with xgboost and other R modeling functions.

When a level of a categorical variable is rare, sometimes it will fail to show up in training data. If that rare level then appears in future data, downstream models may not know what to do with it. When such novel levels appear, using model.matrix or caret::dummyVars to one-hot-encode will not work correctly.

vtreat is a "safer" alternative to model.matrix for one-hot-encoding, because it can manage novel levels safely. vtreat also manages missing values in the data (both categorical and continuous).

```
# Create the testframe for testing new vars
color <- c("g", "g", "y", "g", "g", "y", "b", "g", "g", "r")
size <- c(7, 8, 10, 12, 6, 8, 12, 12, 12, 8)
popularity <- c(0.9733920, 0.9122529, 1.4217153, 1.1905828, 0.9866464, 1.3697515, 1.0959387, 0.9161547,
testframe <- cbind(color, size, popularity)
testframe <- as.data.frame((dframe))

# treatplan is in the workspace
summary(treatplan)
```

```
##               Length Class           Mode
## treatments    4      -none-          list
## scoreFrame    8      data.frame      list
## outcomename   1      -none-          character
## vtreatVersion 1      package_version list
## outcomeType   1      -none-          character
## outcomeTarget 1      -none-          character
## meanY         1      -none-          logical
## splitmethod   1      -none-          character
```

```
# newvars is in the workspace
newvars
```

```
##  [1] "color_lev_x.b" "color_lev_x.g" "color_lev_x.r" "size_lev_x.11"
##  [5] "size_lev_x.12" "size_lev_x.13" "size_lev_x.14" "size_lev_x.15"
##  [9] "size_lev_x.7"  "size_lev_x.9"
```

```
# Print dframe and testframe
dframe
```

```
##    color size popularity
## 1      b   13  1.0785088
## 2      r   11  1.3956245
## 3      r   15  0.9217988
## 4      r   14  1.2025453
## 5      r   13  1.0838662
## 6      b   11  0.8043527
## 7      r    9   1.103544
## 8      g   12  0.8746332
## 9      b    7  0.6947058
## 10     b   12  0.8832502
```

```
testframe
```

```
##    color size popularity
## 1      b   13  1.0785088
## 2      r   11  1.3956245
## 3      r   15  0.9217988
## 4      r   14  1.2025453
## 5      r   13  1.0838662
```

```
## 6      b   11  0.8043527
## 7      r    9   1.103544
## 8      g   12  0.8746332
## 9      b    7  0.6947058
## 10     b   12  0.8832502
```

```
# Use prepare() to one-hot-encode testframe
(testframe.treat <- prepare(treatplan, testframe, varRestriction = newvars))
```

```
##    color_lev_x.b color_lev_x.g color_lev_x.r size_lev_x.11 size_lev_x.12
## 1              1             0             0             0             0
## 2              0             0             1             1             0
## 3              0             0             1             0             0
## 4              0             0             1             0             0
## 5              0             0             1             0             0
## 6              1             0             0             1             0
## 7              0             0             1             0             0
## 8              0             1             0             0             1
## 9              1             0             0             0             0
## 10             1             0             0             0             1
##    size_lev_x.13 size_lev_x.14 size_lev_x.15 size_lev_x.7 size_lev_x.9
## 1              1             0             0            0            0
## 2              0             0             0            0            0
## 3              0             0             1            0            0
## 4              0             1             0            0            0
## 5              1             0             0            0            0
## 6              0             0             0            0            0
## 7              0             0             0            0            1
## 8              0             0             0            0            0
## 9              0             0             0            1            0
## 10             0             0             0            0            0
```

vtreat encodes novel colors like yellow that were not present in the data as all zeros: 'none of the known colors'. This allows downstream models to accept these novel values without crashing.

Next we will create one-hot-encoded data frames of the July/August bike data, for use with xgboost later on. vars defines the variable vars with the list of variable columns for the model.

```
# The outcome column
(outcome <- "cnt")
```

```
## [1] "cnt"
```

```
# The input columns
(vars <- c("hr", "holiday", "workingday", "weathersit", "temp", "atemp", "hum", "windspeed"))
```

```
## [1] "hr"         "holiday"    "workingday" "weathersit" "temp"
## [6] "atemp"      "hum"        "windspeed"
```

```
# Load the package vtreat
library(vtreat)

# Create the treatment plan from bikesJuly (the training data)
treatplan <- designTreatmentsZ(bikesJuly, vars, verbose = FALSE)

# Get the "clean" and "lev" variables from the scoreFrame
(newvars <- treatplan %>%
  magrittr::use_series(scoreFrame) %>%
```

```
  filter(code %in% c("clean", "lev")) %>%  # get the rows you care about
  magrittr::use_series(varName))           # get the varName column
```

```
##  [1] "hr_lev_x.0"
##  [2] "hr_lev_x.1"
##  [3] "hr_lev_x.10"
##  [4] "hr_lev_x.11"
##  [5] "hr_lev_x.12"
##  [6] "hr_lev_x.13"
##  [7] "hr_lev_x.14"
##  [8] "hr_lev_x.15"
##  [9] "hr_lev_x.16"
## [10] "hr_lev_x.17"
## [11] "hr_lev_x.18"
## [12] "hr_lev_x.19"
## [13] "hr_lev_x.2"
## [14] "hr_lev_x.20"
## [15] "hr_lev_x.21"
## [16] "hr_lev_x.22"
## [17] "hr_lev_x.23"
## [18] "hr_lev_x.3"
## [19] "hr_lev_x.4"
## [20] "hr_lev_x.5"
## [21] "hr_lev_x.6"
## [22] "hr_lev_x.7"
## [23] "hr_lev_x.8"
## [24] "hr_lev_x.9"
## [25] "holiday_clean"
## [26] "workingday_clean"
## [27] "weathersit_lev_x.Clear.to.partly.cloudy"
## [28] "weathersit_lev_x.Light.Precipitation"
## [29] "weathersit_lev_x.Misty"
## [30] "temp_clean"
## [31] "atemp_clean"
## [32] "hum_clean"
## [33] "windspeed_clean"
```

```
# Prepare the training data
bikesJuly.treat <- prepare(treatplan, bikesJuly,  varRestriction = newvars)

# Prepare the test data
bikesAugust.treat <- prepare(treatplan, bikesAugust,  varRestriction = newvars)

# Call str() on the treated data
str(bikesAugust.treat)
```

```
## 'data.frame':    744 obs. of  33 variables:
##  $ hr_lev_x.0                           : num  1 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.1                           : num  0 1 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.10                          : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.11                          : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.12                          : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.13                          : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.14                          : num  0 0 0 0 0 0 0 0 0 0 ...
```

```
##  $ hr_lev_x.15                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.16                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.17                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.18                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.19                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.2                               : num  0 0 1 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.20                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.21                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.22                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.23                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.3                               : num  0 0 0 1 0 0 0 0 0 0 ...
##  $ hr_lev_x.4                               : num  0 0 0 0 1 0 0 0 0 0 ...
##  $ hr_lev_x.5                               : num  0 0 0 0 0 1 0 0 0 0 ...
##  $ hr_lev_x.6                               : num  0 0 0 0 0 0 1 0 0 0 ...
##  $ hr_lev_x.7                               : num  0 0 0 0 0 0 0 1 0 0 ...
##  $ hr_lev_x.8                               : num  0 0 0 0 0 0 0 0 1 0 ...
##  $ hr_lev_x.9                               : num  0 0 0 0 0 0 0 0 0 1 ...
##  $ holiday_clean                            : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ workingday_clean                         : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ weathersit_lev_x.Clear.to.partly.cloudy: num  1 1 1 1 0 0 1 0 0 0 ...
##  $ weathersit_lev_x.Light.Precipitation   : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ weathersit_lev_x.Misty                   : num  0 0 0 0 1 1 0 1 1 1 ...
##  $ temp_clean                               : num  0.68 0.66 0.64 0.64 0.64 0.64 0.64 0.64 0.66 0.68 .
##  $ atemp_clean                              : num  0.636 0.606 0.576 0.576 0.591 ...
##  $ hum_clean                                : num  0.79 0.83 0.83 0.83 0.78 0.78 0.78 0.83 0.78 0.74 .
##  $ windspeed_clean                          : num  0.1642 0.0896 0.1045 0.1045 0.1343 ...
```

```
str(bikesJuly.treat)
```

```
## 'data.frame':    744 obs. of  33 variables:
##  $ hr_lev_x.0                               : num  1 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.1                               : num  0 1 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.10                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.11                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.12                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.13                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.14                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.15                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.16                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.17                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.18                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.19                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.2                               : num  0 0 1 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.20                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.21                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.22                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.23                              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ hr_lev_x.3                               : num  0 0 0 1 0 0 0 0 0 0 ...
##  $ hr_lev_x.4                               : num  0 0 0 0 1 0 0 0 0 0 ...
##  $ hr_lev_x.5                               : num  0 0 0 0 0 1 0 0 0 0 ...
##  $ hr_lev_x.6                               : num  0 0 0 0 0 0 1 0 0 0 ...
##  $ hr_lev_x.7                               : num  0 0 0 0 0 0 0 1 0 0 ...
##  $ hr_lev_x.8                               : num  0 0 0 0 0 0 0 0 1 0 ...
##  $ hr_lev_x.9                               : num  0 0 0 0 0 0 0 0 0 1 ...
##  $ holiday_clean                            : num  0 0 0 0 0 0 0 0 0 0 ...
```

```
##  $ workingday_clean                        : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ weathersit_lev_x.Clear.to.partly.cloudy: num  1 1 1 1 1 1 1 1 1 1 ...
##  $ weathersit_lev_x.Light.Precipitation   : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ weathersit_lev_x.Misty                  : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ temp_clean                              : num  0.76 0.74 0.72 0.72 0.7 0.68 0.7 0.74 0.78 0.82 ...
##  $ atemp_clean                             : num  0.727 0.697 0.697 0.712 0.667 ...
##  $ hum_clean                               : num  0.66 0.7 0.74 0.84 0.79 0.79 0.79 0.7 0.62 0.56 ...
##  $ windspeed_clean                         : num  0 0.1343 0.0896 0.1343 0.194 ...
```

The bike data is now in completely numeric form, ready to use with xgboost. Note that the treated data does not include the outcome column.

3.5.3 Gradient Boosting Machines

Gradient boosting is an interative ensemble method, by improving the model each time. We start the model with a usually shallow tree. Next, we fit another model to the residuals of the model, then find the weighted sum of the second and first models that give the best fit. We can regualrise the learning by the factor eta, eta = 1 gives fast learning but with overfitting risk, smaller eta reduces speed of learning but reduces the risk of overfitting. We then repeat this process until the stopping condition is met.

Gradient boosting works on the training data, so it can be easy to overfit. The best approach then is to use OOB and cross validation (CV) for each model, then determine how many trees to use.

xgb.cv() is the function we use and has a number of diagnostic measures. One such measure is the

- xgb.cv()$evaluation_log: records estimated RMSE for each round - find the number that minimises the RMSE

Inputs to xgb.cv() and xgboost() are:

- data: input data as matrix ; label: outcome
- label: vector of outcomes (also numeric)
- objective: for regression - “reg:linear”
- nrounds: maximum number of trees to fit
- eta: learning rate
- max_depth: depth of trees
- early_stopping_rounds: after this many rounds without improvement, stop
- nfold (xgb.cv() only): number of folds for cross validation. 5 is a good number
- verbose: 0 to stay silent.

Then we use

elog <- as.data.frame(cv$evaluation_log)$nrounds < -which.min(elog$test_rmse_mean)

With the resulting number being the best number of trees. We then use xbgoost with this number (nrounds <- n) to get the final model.

Next we will get ready to build a gradient boosting model to predict the number of bikes rented in an hour as a function of the weather and the type and time of day. We will train the model on data from the month of July.

The July data is loaded into your workspace. Remember that bikesJuly.treat no longer has the outcome column, so you must get it from the untreated data: bikesJuly$cnt.

We will use the xgboost package to fit the random forest model. The function xgb.cv() uses cross-validation to estimate the out-of-sample learning error as each new tree is added to the model. The appropriate number of trees to use in the final model is the number that minimizes the holdout RMSE.

```
# The July data is in the workspace
ls()
```

```
##  [1] "bike_model"          "bike_model_rf"       "bikesAugust"
##  [4] "bikesAugust.treat"   "bikesJuly"           "bikesJuly.treat"
##  [7] "bloodpressure"       "bloodpressure_model" "color"
## [10] "dframe"              "dframe.treat"        "fdata"
## [13] "fdata2"              "fe_mean"             "flower_model"
## [16] "flowers"             "fmla"                "fmla.gam"
## [19] "fmla.lin"            "fmla.log"            "fmla_sqr"
## [22] "gp"                  "houseprice"          "houseprice_long"
## [25] "i"                   "incometest"          "incometrain"
## [28] "k"                   "mean_bikes"          "mmat"
## [31] "model"               "model.gam"           "model.lin"
## [34] "model.log"           "model_lin"           "model_sqr"
## [37] "mpg"                 "mpg_model"           "mpg_test"
## [40] "mpg_train"           "N"                   "newrates"
## [43] "newvars"             "nRows"               "outcome"
## [46] "perf"                "popularity"          "pred"
## [49] "pseudoR2"            "Quasipoissonmodel"   "randomforest_plot"
## [52] "res"                 "rho"                 "rho2"
## [55] "rmse"                "rmse_test"           "rmse_train"
## [58] "rsq"                 "rsq_glance"          "rsq_test"
## [61] "rsq_train"           "rss"                 "scoreFrame"
## [64] "sd_unemployment"     "seed"                "size"
## [67] "soybean_long"        "soybean_test"        "soybean_train"
## [70] "sparrow"             "sparrow_model"       "split"
## [73] "splitPlan"           "target"              "testframe"
## [76] "testframe.treat"     "treatplan"           "tss"
## [79] "unemployment"        "unemployment_model"  "var_bikes"
## [82] "vars"
```

```
# Load the package xgboost
library(xgboost)
```

```
##
## Attaching package: 'xgboost'
```

```
## The following object is masked from 'package:dplyr':
##
##     slice
```

```
# Run xgb.cv
cv <- xgb.cv(data = as.matrix(bikesJuly.treat),
             label = bikesJuly$cnt,
             nrounds = 100,
             nfold = 5,
             objective = "reg:linear",
             eta = 0.3,
             max_depth = 6,
             early_stopping_rounds = 10,
             verbose = 0    # silent
)

# Get the evaluation log
elog <- as.data.frame(cv$evaluation_log)
```

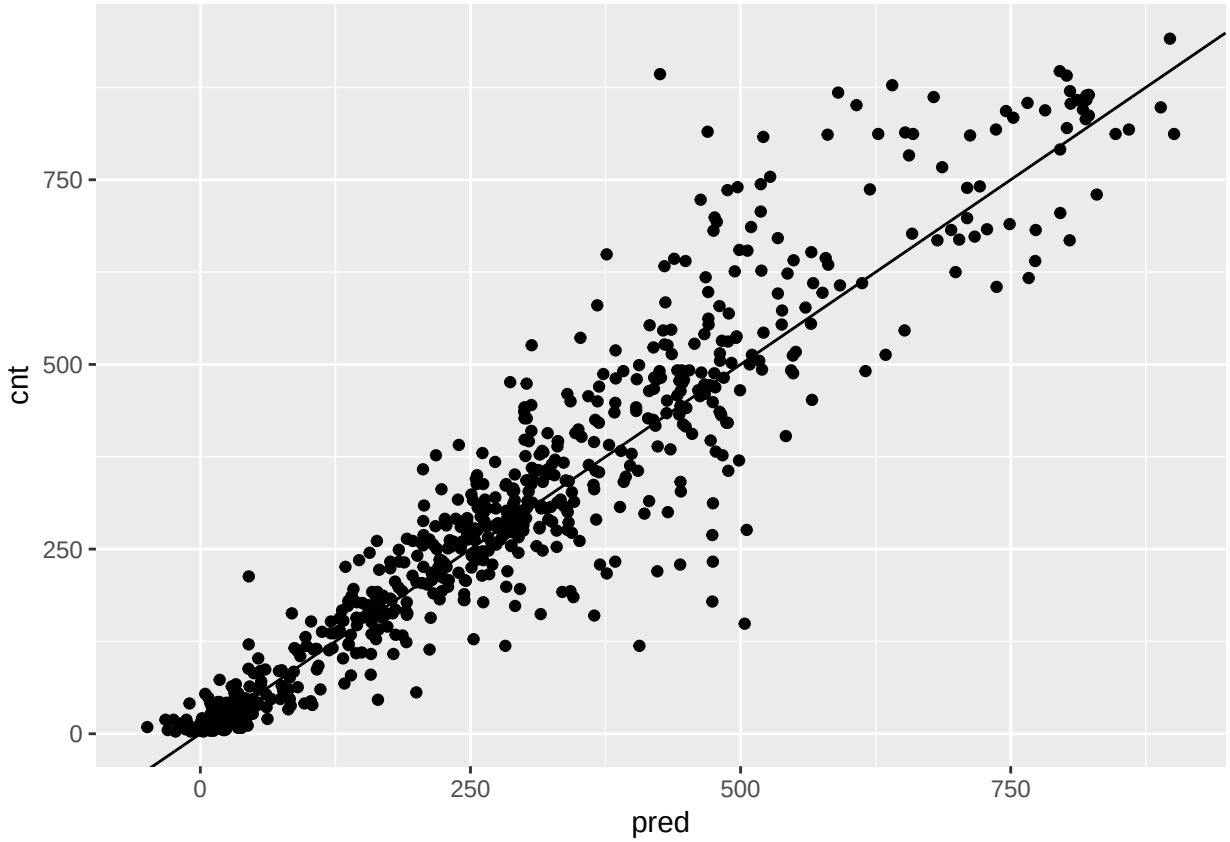
```
# Determine and print how many trees minimize training and test error
elog %>%
   summarize(ntrees.train = which.min(elog$train_rmse_mean),   # find the index of min(train_rmse_mean)
             ntrees.test  = which.min(elog$test_rmse_mean))   # find the index of min(test_rmse_mean)
```

```
##   ntrees.train ntrees.test
## 1           77          67
```

In most cases, ntrees.test is less than ntrees.train. The training error keeps decreasing even after the test error starts to increase. It's important to use cross-validation to find the right number of trees (as determined by ntrees.test) and avoid an overfit model.

Next we will fit a gradient boosting model using xgboost() to predict the number of bikes rented in an hour as a function of the weather and the type and time of day. We will train the model on data from the month of July and predict on data for the month of August.

```
# The number of trees to use, as determined by xgb.cv
ntrees <- 84

# Run xgboost
bike_model_xgb <- xgboost(data = as.matrix(bikesJuly.treat), # training data as matrix
                  label = bikesJuly$cnt,  # column of outcomes
                  nrounds = ntrees,       # number of trees to build
                  objective = "reg:linear", # objective
                  eta = 0.3,
                  depth = 6,
                  verbose = 0  # silent
)

# Make predictions
bikesAugust$pred <- predict(bike_model_xgb, as.matrix(bikesAugust.treat))

# Plot predictions (on x axis) vs actual bike rental count
ggplot(bikesAugust, aes(x = pred, y = cnt)) +
  geom_point() +
  geom_abline()
```



Overall, the scatterplot looked pretty good, but notice that the model made some negative predictions.

In the next exercise, you'll compare this model's RMSE to the previous bike models that you've built.

Finally we can calculate the RMSE.

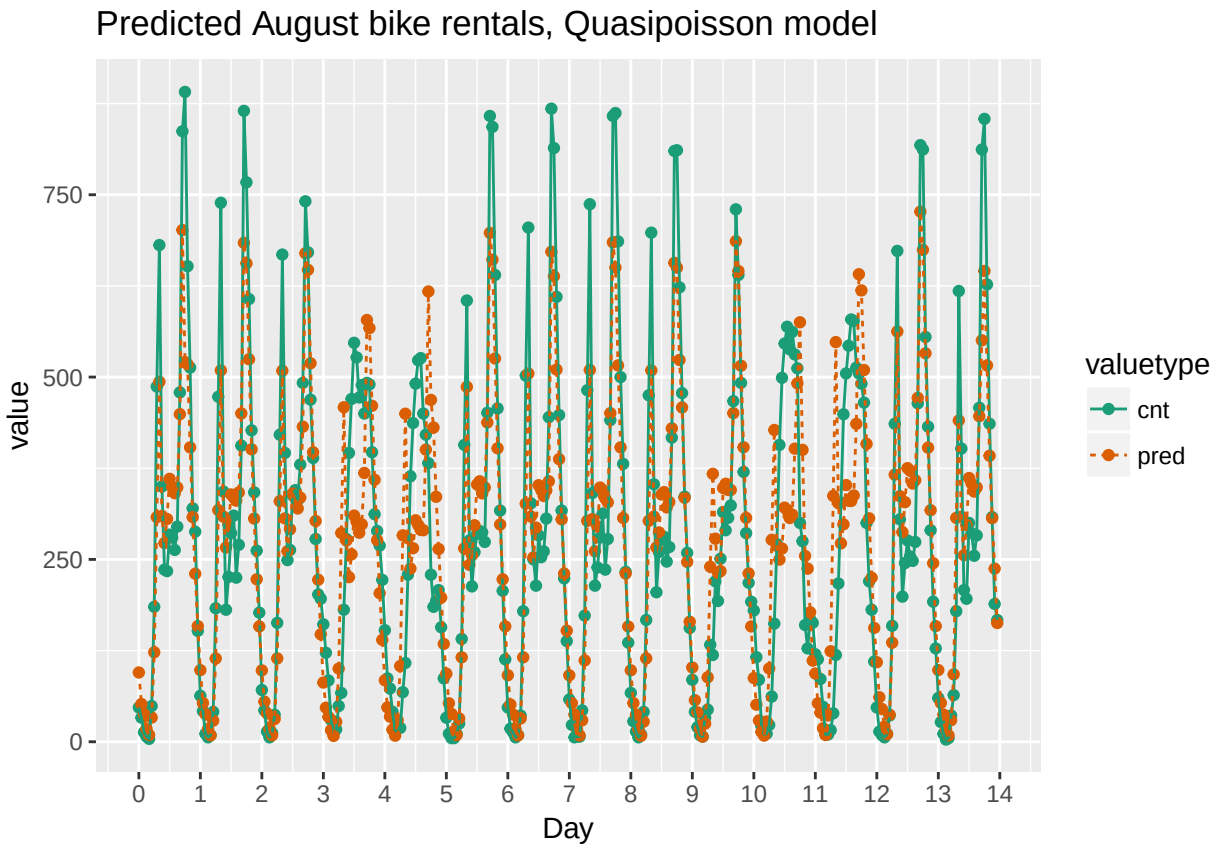
```
# Calculate RMSE
bikesAugust %>%
  mutate(residuals = cnt - pred) %>%
  summarize(rmse = sqrt(mean(residuals ^ 2)))
```

```
##       rmse
## 1 76.36407
```

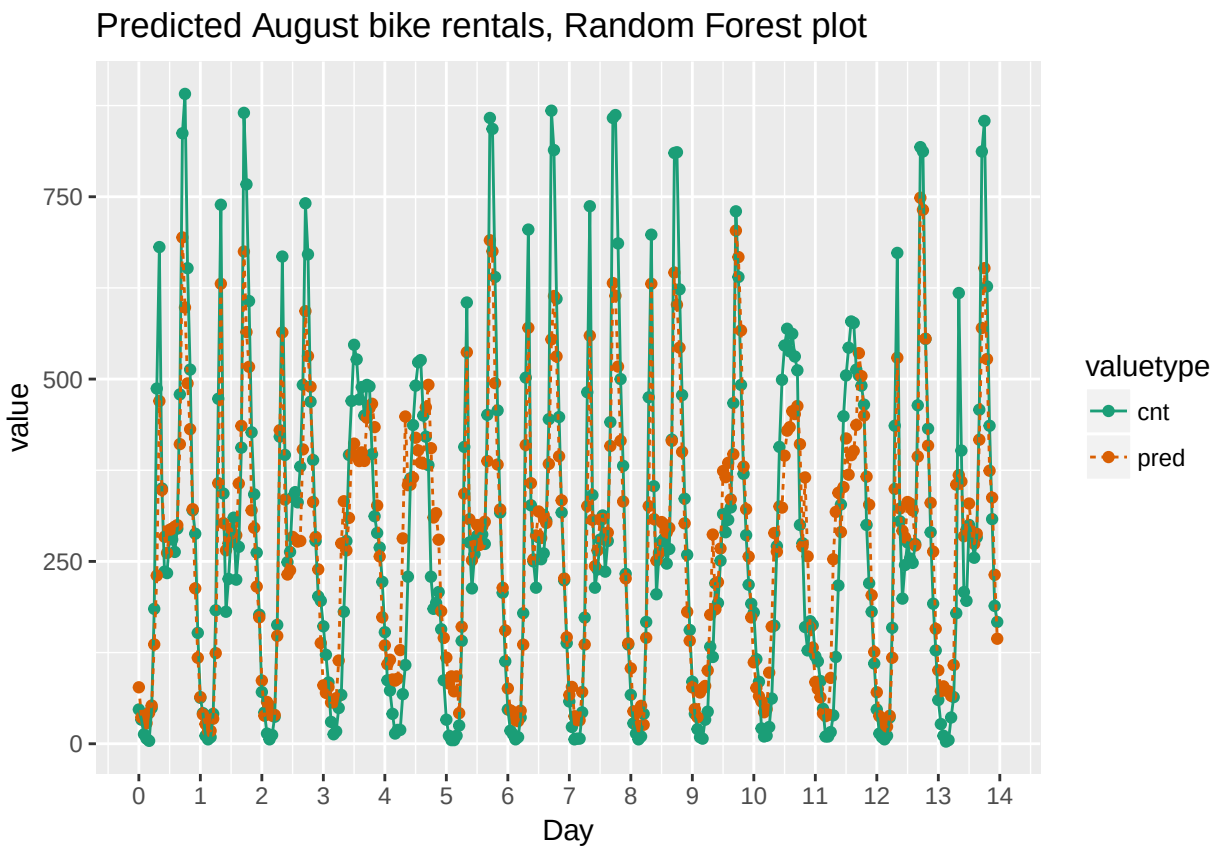
Even though this gradient boosting made some negative predictions, overall it makes smaller errors than the previous two models. Perhaps rounding negative predictions up to zero is a reasonable tradeoff.

Finally we print our previous plots and create a new one for the xgb and compare the results.

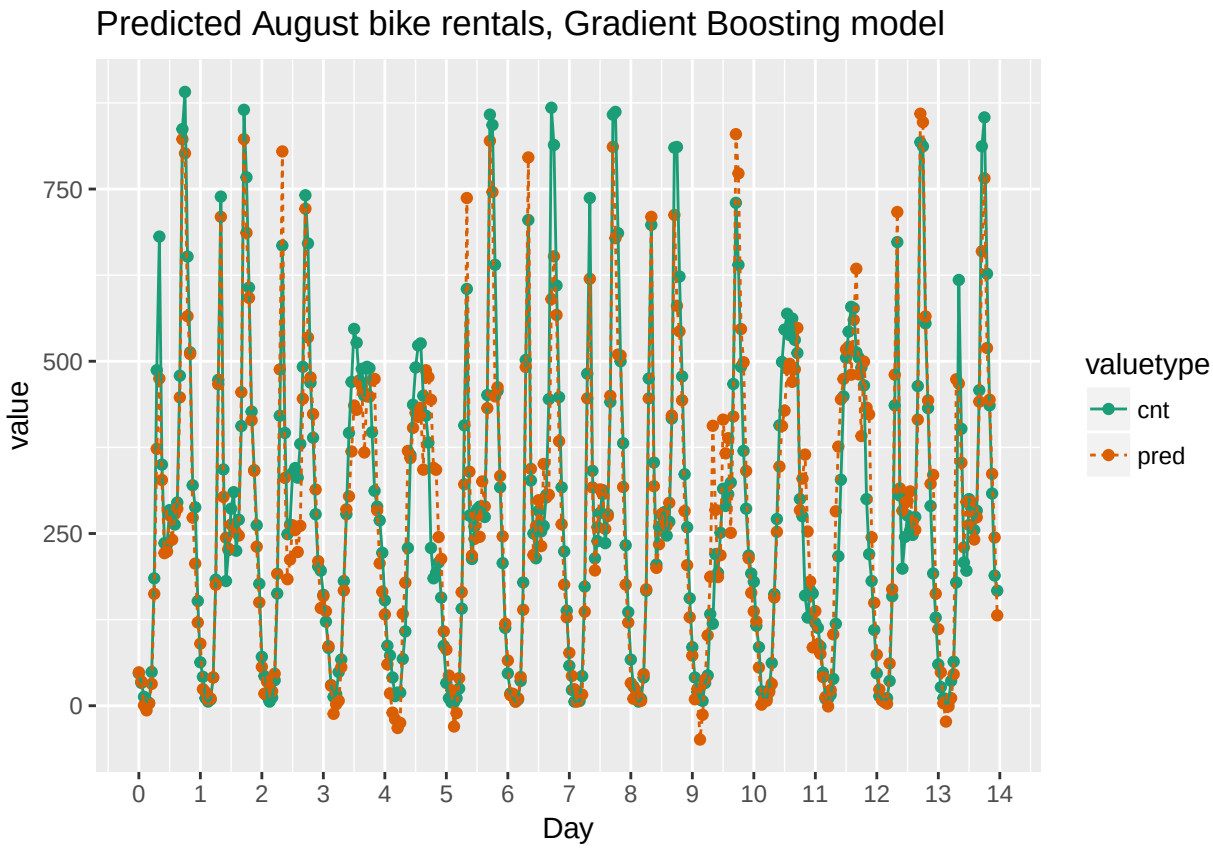
```
Quasipoissonmodel
```




```
randomforest_plot
```




```
# Plot predictions and actual bike rentals as a function of time (days)
bikesAugust %>%
  mutate(instant = (instant - min(instant))/24) %>%  # set start to 0, convert unit to days
  gather(key = valuetype, value = value, cnt, pred) %>%
  filter(instant < 14) %>% # first two weeks
  ggplot(aes(x = instant, y = value, color = valuetype, linetype = valuetype)) +
  geom_point() +
  geom_line() +
  scale_x_continuous("Day", breaks = 0:14, labels = 0:14) +
  scale_color_brewer(palette = "Dark2") +
  ggtitle("Predicted August bike rentals, Gradient Boosting model")
```


Predicted August bike rentals, Gradient Boosting model
value
750
500
250
0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Day
valuetype
cnt
pred

Chapter 4

Machine Learning Toolbox

Notes taken during/inspired by the DataCamp course 'Machine Learning Toolbox' by Zachary Deane-Mayer and Max Kuhn.

Course Handouts

- Part 1 - Regression models: fitting them and evaluating their performance
- [Part 2 - Classification models: fitting them and evaluating their performance] ()
- [Part 3 - Tuning model parameters to improve performance] ()
- [Part 4 - Preprocessing your data] ()
- [Part 5 - Selecting models: a case study in churn prediction] ()

Other useful links

- Max Kuhn: Applied Predictive Modeling NYC Talk
- Data Chat - Interview With Max Kuhn
- The caret package website
- Bagging graphical explanation
- Boosting graphical explanation
- Gradient Boosting in Practice: a deep dive into xgboost

4.1 Regression models: fitting them and evaluating their performance

CAret has been developed by Max for over 10 years. It automates supervised machine learning (aka predictive modelling) is ML when you have a target variable or something specific you want to predict like species and churn, these could be classification or regression. We then use metrics to evaluate how accuratley our models predict on new data.

For linear regression we will use RMSE as our evaluation metric. Typically this is done on our training (in-sample) data, however this can be too optomistic and lead to over-fitting. It is better to calculate this out-of-sample using caret.

Next we will calculate in-sample RMSE for the diamonds dataset from ggplot2.

```
# Load the data
diamonds <- ggplot2::diamonds

# Fit lm model: model
```

```
model <- lm(price ~ ., diamonds)

# Predict on full data: p
p <- predict(model, diamonds)

# Compute errors: error
error <- p - diamonds$price

# Calculate in-sample RMSE
sqrt(mean(error ^ 2))
```

```
## [1] 1129.843
```

The course focuses on predictive accuracy, that is to say does the model perform well When presented with new data. The best way to answer this is to test the model on new data using test data. This mimics the real world, where you do not actually know the outcome. We simulate this with a test/train split.

One way you can take a train/test split of a dataset is to order the dataset randomly, then divide it into the two sets. This ensures that the training set and test set are both random samples and that any biases in the ordering of the dataset (e.g. if it had originally been ordered by price or size) are not retained in the samples we take for training and testing your models. You can think of this like shuffling a brand new deck of playing cards before dealing hands.

```
# Set seed
set.seed(42)

# Shuffle row indices: rows
rows <- sample(nrow(diamonds))

# Randomly order data
diamonds <- diamonds[rows,]
```

Now that your dataset is randomly ordered, you can split the first 80% of it into a training set, and the last 20% into a test set. You can do this by choosing a split point approximately 80% of the way through your data.

```
# Determine row to split on: split
split <- round(nrow(diamonds) * .80)

# Create train
train <- diamonds[1:split, ]

# Create test
test <- diamonds[(split + 1):nrow(diamonds), ]
```

Now that you have a randomly split training set and test set, you can use the lm() function as you did in the first exercise to fit a model to your training set, rather than the entire dataset. Recall that you can use the formula interface to the linear regression function to fit a model with a specified target variable using all other variables in the dataset as predictors:

mod <- lm(y ~ ., training_data)

You can use the predict() function to make predictions from that model on new data. The new dataset must have all of the columns from the training data, but they can be in a different order with different values. Here, rather than re-predicting on the training set, you can predict on the test set, which you did not use for training the model. This will allow you to determine the out-of-sample error for the model in the next exercise:

p <- predict(model, new_data)

```
# Fit lm model on train: model
model <- lm(price ~ ., train)

# Predict on test: p
p <- predict(model,test)
```

Now that you have predictions on the test set, you can use these predictions to calculate an error metric (in this case RMSE) on the test set and see how the model performs out-of-sample, rather than in-sample as you did in the first exercise. You first do this by calculating the errors between the predicted diamond prices and the actual diamond prices by subtracting the predictions from the actual values.

```
# Compute errors: error
error <- (p - test$price)

# Calculate RMSE
sqrt(mean(error^2))
```

```
## [1] 1136.596
```

4.1.1 Cross-validation

Using a simple text:train split can be tricky, particularly if there are some outliers in one of the two datasets. A better approach is to use multiple test sets and average out of sample error. One way of achieving this is cross-validation where we use folds, 10 folds would mean we split our data in to ten sections with a single observation only occuring once. Cross validation is only used to calculate error metrics (out of sample), we then start again using on the full data. With 10 folds, we therefore have 11 models to fit - the 10 re-sampled models plus the final model. An alterntive which is available in caret is the bootstrap, but in practice this yields similar results to cross-validation.

caret package makes this very easy to do:

model <- train(y ~ ., my_data)

caret supports many types of cross-validation, and you can specify which type of cross-validation and the number of cross-validation folds with the trainControl() function, which you pass to the trControl argument in train():

model <- train(y ~ ., my_data, method = "lm", trControl = trainControl(method = "cv", number = 10, verboseIter = TRUE))

It's important to note that you pass the method for modeling to the main train() function and the method for cross-validation to the trainControl() function.

```
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     diamonds
```

```
# Fit lm model using 10-fold CV: model
model <- train(
```

```
  price ~ ., diamonds,
  method = "lm",
  trControl = trainControl(
    method = "cv", number = 10,
    verboseIter = TRUE
  )
)
```

```
## + Fold01: intercept=TRUE
## - Fold01: intercept=TRUE
## + Fold02: intercept=TRUE
## - Fold02: intercept=TRUE
## + Fold03: intercept=TRUE
## - Fold03: intercept=TRUE
## + Fold04: intercept=TRUE
## - Fold04: intercept=TRUE
## + Fold05: intercept=TRUE
## - Fold05: intercept=TRUE
## + Fold06: intercept=TRUE
## - Fold06: intercept=TRUE
## + Fold07: intercept=TRUE
## - Fold07: intercept=TRUE
## + Fold08: intercept=TRUE
## - Fold08: intercept=TRUE
## + Fold09: intercept=TRUE
## - Fold09: intercept=TRUE
## + Fold10: intercept=TRUE
## - Fold10: intercept=TRUE
## Aggregating results
## Fitting final model on full training set
```

```
# Print model to console
print(model)
```

```
## Linear Regression
##
## 53940 samples
##     9 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 48547, 48546, 48546, 48545, 48545, 48545, ...
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   1130.658  0.9197492  740.4646
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

Next we use a different dataset - Boston house prices - then use a 5 fold cross validation in the trainControl.

```
Boston <- MASS::Boston

# Fit lm model using 5-fold CV: model
model <- train(
  medv ~ ., Boston,
```

```
  method = "lm",
  trControl = trainControl(
    method = "cv", number = 5,
    verboseIter = TRUE
  )
)
```

```
## + Fold1: intercept=TRUE
## - Fold1: intercept=TRUE
## + Fold2: intercept=TRUE
## - Fold2: intercept=TRUE
## + Fold3: intercept=TRUE
## - Fold3: intercept=TRUE
## + Fold4: intercept=TRUE
## - Fold4: intercept=TRUE
## + Fold5: intercept=TRUE
## - Fold5: intercept=TRUE
## Aggregating results
## Fitting final model on full training set
```

```
# Print model to console
print(model)
```

```
## Linear Regression
##
## 506 samples
##  13 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 405, 403, 405, 406, 405
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   4.794707  0.7290369  3.372915
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

You can do more than just one iteration of cross-validation. Repeated cross-validation gives you a better estimate of the test-set error. You can also repeat the entire cross-validation procedure. This takes longer, but gives you many more out-of-sample datasets to look at and much more precise assessments of how well the model performs.

One of the awesome things about the train() function in caret is how easy it is to run very different models or methods of cross-validation just by tweaking a few simple arguments to the function call. For example, you could repeat your entire cross-validation procedure 5 times for greater confidence in your estimates of the model's out-of-sample accuracy

```
# Fit lm model using 5 x 5-fold CV: model
model <- train(
  medv ~ ., Boston,
  method = "lm",
  trControl = trainControl(
    method = "cv", number = 5,
    repeats = 5, verboseIter = TRUE
  )
```

```
)
```

```
## Warning: `repeats` has no meaning for this resampling method.
```

```
## + Fold1: intercept=TRUE
## - Fold1: intercept=TRUE
## + Fold2: intercept=TRUE
## - Fold2: intercept=TRUE
## + Fold3: intercept=TRUE
## - Fold3: intercept=TRUE
## + Fold4: intercept=TRUE
## - Fold4: intercept=TRUE
## + Fold5: intercept=TRUE
## - Fold5: intercept=TRUE
## Aggregating results
## Fitting final model on full training set
```

```
# Print model to console
print(model)
```

```
## Linear Regression
##
## 506 samples
##  13 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 405, 405, 404, 405, 405
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   4.875484  0.7316537  3.425015
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

Finally, the model you fit with the train() function has the exact same predict() interface as the linear regression models you fit earlier in this chapter.

After fitting a model with train(), you can simply call predict() with new data, e.g:

```
# Predict on full Boston dataset and display just the first 20 items
head (predict (model, Boston), n = 20)
```

```
##        1        2        3        4        5        6        7        8
## 30.00384 25.02556 30.56760 28.60704 27.94352 25.25628 23.00181 19.53599
##        9       10       11       12       13       14       15       16
## 11.52364 18.92026 18.99950 21.58680 20.90652 19.55290 19.28348 19.29748
##       17       18       19       20
## 20.52751 16.91140 16.17801 18.40614
```

4.2 Classification models: fitting them and evaluating their performance

In classification models you are trying to predict a categorical target e.g. whether a customer is satisfied or not. It it still a form of supervised learning and we can use a train/test split to evaluate performance. In

this sectgion we will use the Sonar dataset which contains charecteristics of a sonar signal for objects that are either rocks or mines.

The variables explain some part of the signal, then the class of either Rock or Mine - the data below shows the first 7 variables and the class, for a random sample of 10 rows. As the dataset is very small, we use a 40% test split.

```
library(mlbench)
data(Sonar)
Sonar[sample(1:nrow(Sonar), 10, replace=FALSE), c(1:7, 61)]
```

```
##         V1     V2     V3     V4     V5     V6     V7 Class
## 3   0.0262 0.0582 0.1099 0.1083 0.0974 0.2280 0.2431     R
## 31  0.0240 0.0218 0.0324 0.0569 0.0330 0.0513 0.0897     R
## 131 0.0443 0.0446 0.0235 0.1008 0.2252 0.2611 0.2061     M
## 77  0.0239 0.0189 0.0466 0.0440 0.0657 0.0742 0.1380     R
## 118 0.0228 0.0106 0.0130 0.0842 0.1117 0.1506 0.1776     M
## 46  0.0408 0.0653 0.0397 0.0604 0.0496 0.1817 0.1178     R
## 47  0.0308 0.0339 0.0202 0.0889 0.1570 0.1750 0.0920     R
## 130 0.1371 0.1226 0.1385 0.1484 0.1776 0.1428 0.1773     M
## 26  0.0201 0.0026 0.0138 0.0062 0.0133 0.0151 0.0541     R
## 86  0.0365 0.1632 0.1636 0.1421 0.1130 0.1306 0.2112     R
```

TO mkae our 60/40 split we do the following.

```
# Shuffle row indices: rows
rows <- sample(nrow(Sonar))

# Randomly order data: Sonar
Sonar <- Sonar[rows, ]

# Identify row to split on: split
split <- round(nrow(Sonar) * 0.6)

# Create train
train <- Sonar[1:split, ]

# Create test
test <- Sonar[(split + 1):nrow(Sonar), ]
```

Now you can fit a logistic regression model to your training set using the glm() function. glm() is a more advanced version of lm() that allows for more varied types of regression models, aside from plain vanilla ordinary least squares regression.

Don't worry about warnings like glm.fit: algorithm did not converge or glm.fit: fitted probabilities numerically 0 or 1 occurred. These are common on smaller datasets and usually don't cause any issues. They typically mean your dataset is perfectly seperable, which can cause problems for the math behind the model, but R's glm() function is almost always robust enough to handle this case with no problems.

```
# Fit glm model: model
model <- glm(formula = Class ~ ., family = "binomial", data = train)
```

```
## Warning: glm.fit: algorithm did not converge
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
# Predict on test: p
p <- predict(model, test, type = "response")
```

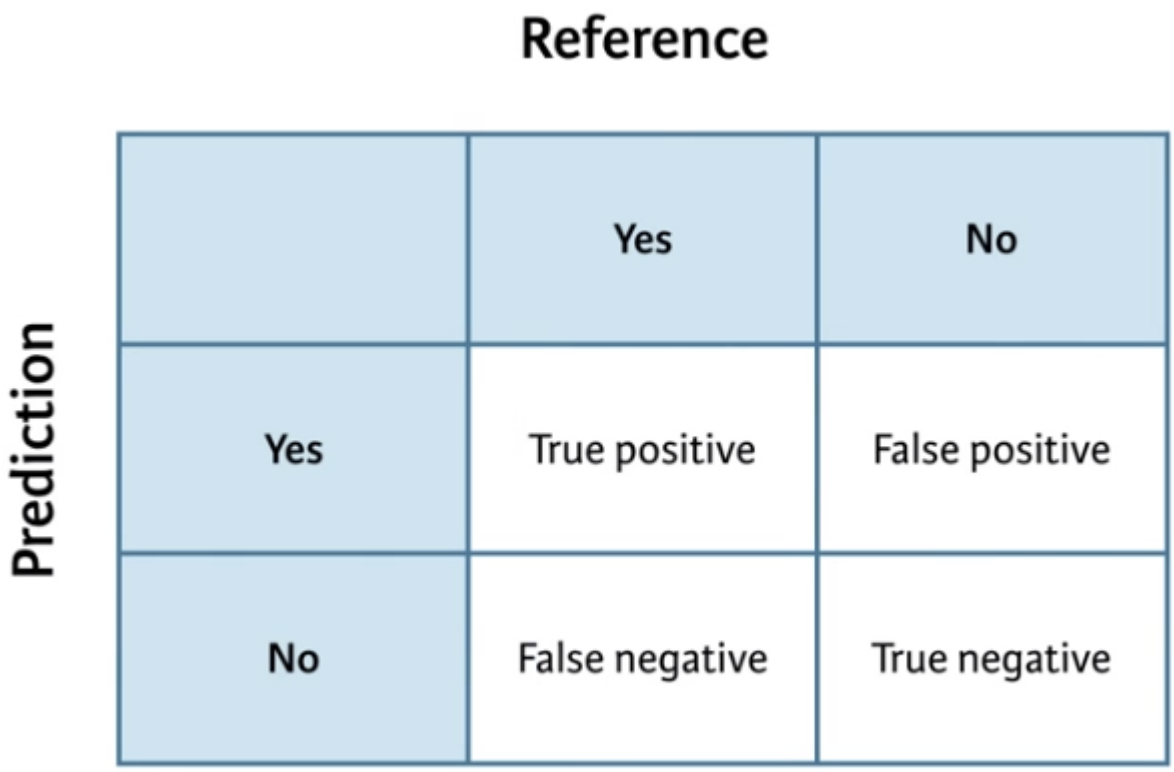


Figure 4.1: Confusion Matrix

(#fig:Confusion Matrix)

4.2.1 Confusion Matrix

A confusion Matrix is a table of the predicted results vs the actual results. It is called a Confusion Matrix as it explains how confused the model is between the two classes and highlights instances where one class is confused for the other. The items on the main diagonal are the cases where the model is correct.

You could make such a contingency table with the table() function in base R, but confusionMatrix() in caret yields a lot of useful ancillary statistics in addition to the base rates in the table.

```
# Calculate class probabilities: p_class
p_class <- ifelse(p > .5, "M", "R")

# Create confusion matrix
confusionMatrix(p_class, test$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  M  R
##          M 12 24
##          R 32 15
##
##                Accuracy : 0.3253
##                  95% CI : (0.2265, 0.437)
##     No Information Rate : 0.5301
```

```
##     P-Value [Acc > NIR] : 0.9999
##
##                   Kappa : -0.3387
##  Mcnemar's Test P-Value : 0.3496
##
##             Sensitivity : 0.2727
##             Specificity : 0.3846
##          Pos Pred Value : 0.3333
##          Neg Pred Value : 0.3191
##              Prevalence : 0.5301
##          Detection Rate : 0.1446
##    Detection Prevalence : 0.4337
##       Balanced Accuracy : 0.3287
##
##        'Positive' Class : M
##
```

We see that the No Information rate is around 52%, that is just predicting the dominant class all the time, mines. Our accuracy so far is below this which suggests that using a dummy model that always predicts mines is more accurate than the current model. The other values such as sensitivity and specificity give more elements of the confusion matrix:

- Specificity - true negative rate
- Sensitivity - true positive rate

Setting our threshold rate - currently 50% - is an exercise in deciding what is more important and depends on the cost-benefit analysis of the problem at hand. Do we want to flag more non-mines as mines, or do we want to be more accurate in our prediction but potentially miss some mines?

Now what happens if we apply a 90% threshold to our model?

```
# Apply threshold of 0.9: p_class
p_class <- ifelse(p > .9, "M", "R")

# Create confusion matrix
confusionMatrix(p_class, test$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  M  R
##          M 12 24
##          R 32 15
##
##                Accuracy : 0.3253
##                  95% CI : (0.2265, 0.437)
##     No Information Rate : 0.5301
##     P-Value [Acc > NIR] : 0.9999
##
##                   Kappa : -0.3387
##  Mcnemar's Test P-Value : 0.3496
##
##             Sensitivity : 0.2727
##             Specificity : 0.3846
##          Pos Pred Value : 0.3333
##          Neg Pred Value : 0.3191
```

```
##              Prevalence : 0.5301
##          Detection Rate : 0.1446
##    Detection Prevalence : 0.4337
##       Balanced Accuracy : 0.3287
##
##        'Positive' Class : M
##
```

Or at 10%

```
# Apply threshold of 0.10: p_class
p_class <- ifelse(p > .1, "M", "R")

# Create confusion matrix
confusionMatrix(p_class, test$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  M  R
##          M 13 24
##          R 31 15
##
##                Accuracy : 0.3373
##                  95% CI : (0.2372, 0.4495)
##     No Information Rate : 0.5301
##     P-Value [Acc > NIR] : 0.9999
##
##                   Kappa : -0.3167
##  Mcnemar's Test P-Value : 0.4185
##
##             Sensitivity : 0.2955
##             Specificity : 0.3846
##          Pos Pred Value : 0.3514
##          Neg Pred Value : 0.3261
##              Prevalence : 0.5301
##          Detection Rate : 0.1566
##    Detection Prevalence : 0.4458
##       Balanced Accuracy : 0.3400
##
##        'Positive' Class : M
##
```

4.2.2 The ROC Curve

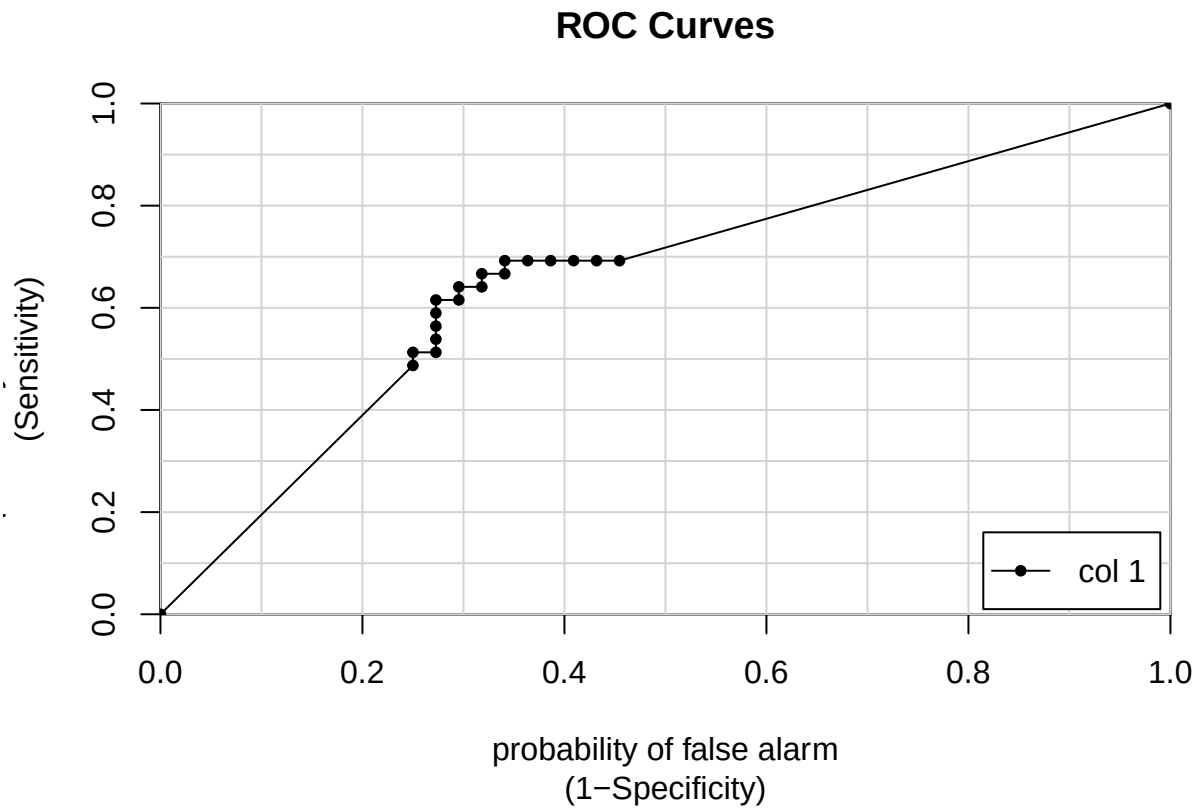
Rather than manually calculating the true positive and true negative rate for many hundreds of different models - at carying classification cut offs as we previously did - we can plot a ROC (Receiver Operator Charecteristic) curve automatically for every possible threshold. Originally the ROC curve was designed to look at the trade off with different thresholds when trying to differentiate between radar signals of flocks of birds vs planes.

The ROC curve plots the false positive rate (X Axis) vs the true positive rate (Y axis) with each point representing a different threshold.

There are a number of different ways of calculating ROC curves, one we will use here is the caTools package

and calculates the AUC or Area under the curve.

```
library(caTools)

# Predict on test: p
p <- predict(model, test, type = "response")

# Make ROC curve
colAUC(p, test$Class, plotROC = TRUE)
```




```
##              [,1]
## M vs. R 0.6564685
```

Whne looking at the ROC curve, a line close to the diagonal would be considered bad as it is following something not far off random predictions. Perfection seperation would produce a box, with a single point at 1, 0 or the top left hand corner which would be a 100% true positive rate and a 0% false positive rate.

The AUC for the 'perfect' model is 1. For the random model it is 0.5. The AUC statistics is a single number accuracy measure, it summarises the performance of the model across all possible classification thresholds. Most models are in the 0.5 to 1 range, some bad models can be in the 0.4 range. We can think of them as an exam grade e.g.

- A = 0.9
- B = 0.8
- C = 0.7

- ...
- F = 0.5

0.8 or above is good, some models in the 0.7 range are useful. The caret package will calculate the AUC for us.

You can use the trainControl() function in caret to use AUC (instead of acccuracy), to tune the parameters of your models. The twoClassSummary() convenience function allows you to do this easily.

When using twoClassSummary(), be sure to always include the argument classProbs = TRUE or your model will throw an error. You cannot calculate AUC with just class predictions. You need to have class probabilities as well.

```
# Create trainControl object: myControl
myControl <- trainControl(
  method = "cv",
  number = 10,
  summaryFunction = twoClassSummary,
  classProbs = TRUE, # IMPORTANT!
  verboseIter = TRUE
)
```

Now that you have a custom trainControl object, it's easy to fit caret models that use AUC rather than accuracy to tune and evaluate the model. You can just pass your custom trainControl object to the train() function via the trControl argument, e.g.:

train(, trControl = myControl)

This syntax gives you a convenient way to store a lot of custom modeling parameters and then use them across multiple different calls to train()

```
# Train glm with custom trainControl: model
model <- train(Class ~ ., data = Sonar,
                  method = "glm",
                  trControl = myControl)
```

```
## Warning in train.default(x, y, weights = w, ...): The metric "Accuracy" was
## not in the result set. ROC will be used instead.

## + Fold01: parameter=none

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## - Fold01: parameter=none
## + Fold02: parameter=none

## Warning: glm.fit: algorithm did not converge


## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## - Fold02: parameter=none
## + Fold03: parameter=none

## Warning: glm.fit: algorithm did not converge


## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## - Fold03: parameter=none
## + Fold04: parameter=none
```

```
## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## - Fold04: parameter=none
## + Fold05: parameter=none

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## - Fold05: parameter=none
## + Fold06: parameter=none

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## - Fold06: parameter=none
## + Fold07: parameter=none

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## - Fold07: parameter=none
## + Fold08: parameter=none

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## - Fold08: parameter=none
## + Fold09: parameter=none

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## - Fold09: parameter=none
## + Fold10: parameter=none

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## - Fold10: parameter=none
## Aggregating results
## Fitting final model on full training set

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
# Print model to console
model
```

```
## Generalized Linear Model
##
## 208 samples
##  60 predictor
##   2 classes: 'M', 'R'
```

```
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 187, 187, 187, 187, 187, 188, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.7539899  0.7295455  0.7522222
```

4.2.3 Tuning model parameters to improve performance

Next we will look at random forests as these are robust against over-fitting. They can yield accurate, non-linear models without much additional work. However, unlinke linear models they have so called hyper-parameters to tune and they have to be manually specified by the data scientist as inputs in to the model. These parameters can vary from dataset to dataset, the defaults can be ok, but often need adjusting.

With random forests we fit a different tree to each bootstrap sample of the data, called bagging - we can think of this as draw a random 'bag' of observations, at random with replacement, from the original TRAIN dataset many times - to which we think fit a tree on this subset of data. This effectivly creates multiple train/test splits on the TRAIN data and trains a different model, so we end up with an ensemble of different models.

Note - boosting is similar to bagging, in that we take a random 'bag' of data and fit a model. However we fit the models iteratively with boosting, with subsequent models fitted to those observations which were not previously fitted as well. When we draw another sample or bag, we weight the probability of observation selection based on how well the previous model fitted those observations, if they were poorly fitted, the new model is more likely to sample those and therefore more likely to accuratley predict them. We then re-inputt all the TRAIN data to our two models and see which observations were then, based on this two model ensemble, not predicted as well. When then try to build a new model in which these previously under-predicted observations are better predicted, then build the ensemble again and iterate as neccessary.

Random forests take bagging a step further, by randomly sampling the columns (variables) at each split and helps to yield more accurate models. It does this to try and de-correlate the errors in different bootstrap samples i.e. different models. You make errors everywhere, but they have different origins/models.

Bagging can therefore be run in parallel, since each bag or draw is independent of the others. In Boosting, this is not possible, since subsequent draws are dependent on previous draws/models.

ISR notes:

"While bagging can improve predictions for many regression methods, it is particularly useful for decision trees. To apply bagging to regression trees, we simply construct B regression trees using B bootstrapped training sets, and average the resulting predictions. These trees are grown deep, and are not pruned. Hence each individual tree has high variance, but low bias. Averaging these B trees reduces the variance. Bagging has been demonstrated to give impressive improvements in accuracy by combining together hundreds or even thousands of trees into a single procedure." (pg 317)

And

"Boosting does not involve bootstrap sampling; instead each tree is fit on a modified version of the original data set…Unlike fitting a single large decision tree to the data, which amounts to fitting the data hard and potentially overfitting, the boosting approach instead learns slowly. Given the current model, we fit a decision tree to the residuals from the model. That is, we fit a tree using the current residuals, rather than the outcome Y , as the response. We then add this new decision tree into the fitted function in order to update the residuals. Each of these trees can be rather small, with just a few terminal nodes, determined by the parameter d in the algorithm."

So bagging tends to produce large (deep) trees with high variance, where as boosting produces shallow trees with high bias.

Fitting a random forest model is exactly the same as fitting a generalized linear regression model, as you did in the previous chapter. You simply change the method argument in the train function to be "ranger". The ranger package is a rewrite of R's classic randomForest package and fits models much faster, but gives almost exactly the same results.

```
# Load the wine data
wine <- readRDS("./files/MLToolbox/wine_100.rds")

# Fit random forest: model
model <- train(
  quality ~.,
  tuneLength = 1,
  data = wine, method = "ranger",
  trControl = trainControl(method = "cv", number = 5, verboseIter = TRUE)
)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
## + Fold1: mtry=3, splitrule=variance
## - Fold1: mtry=3, splitrule=variance
## + Fold1: mtry=3, splitrule=extratrees
## - Fold1: mtry=3, splitrule=extratrees
## + Fold2: mtry=3, splitrule=variance
## - Fold2: mtry=3, splitrule=variance
## + Fold2: mtry=3, splitrule=extratrees
## - Fold2: mtry=3, splitrule=extratrees
## + Fold3: mtry=3, splitrule=variance
## - Fold3: mtry=3, splitrule=variance
## + Fold3: mtry=3, splitrule=extratrees
## - Fold3: mtry=3, splitrule=extratrees
## + Fold4: mtry=3, splitrule=variance
## - Fold4: mtry=3, splitrule=variance
## + Fold4: mtry=3, splitrule=extratrees
## - Fold4: mtry=3, splitrule=extratrees
## + Fold5: mtry=3, splitrule=variance
## - Fold5: mtry=3, splitrule=variance
## + Fold5: mtry=3, splitrule=extratrees
## - Fold5: mtry=3, splitrule=extratrees
## Aggregating results
## Selecting tuning parameters
## Fitting mtry = 3, splitrule = variance on full training set
```

```
# Print model to console
model
```

```
## Random Forest
```

```
##
## 100 samples
##  12 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 81, 80, 80, 79, 80
## Resampling results across tuning parameters:
##
##   splitrule   RMSE       Rsquared   MAE
##   variance    0.6579440  0.3157395  0.4978594
##   extratrees  0.6842383  0.2852057  0.5194518
##
## Tuning parameter 'mtry' was held constant at a value of 3
## RMSE was used to select the optimal model using  the smallest value.
## The final values used for the model were mtry = 3 and splitrule = variance.
```

A difference between random forests and linear models is that random forests require tuning, this is done using "hyperparameters" and is done before fitting the model. One such variable is *mtry* which is the number of randomly selected variables used in each split. It is not always possible to know which is better for your data in advance. But caret can help to automate the selection of these parameters using something called *grid search*. It will select the appropriate hyperparameters based on out-of-sample error.

Random forest models have a primary tuning parameter of mtry, which controls how many variables are exposed to the splitting search routine at each split. For example, suppose that a tree has a total of 10 splits and mtry = 2. This means that there are 10 samples of 2 predictors each time a split is evaluated.

We will use a larger tuning grid this time, but stick to the defaults provided by the train() function. We try a tuneLength of 3, rather than 1, to explore some more potential models, and plot the resulting model using the plot function.

```
# Fit random forest: model
model <- train(
  quality ~.,
  tuneLength = 3,
  data = wine, method = "ranger",
  trControl = trainControl(method = "cv", number = 5, verboseIter = TRUE)
)
```

```
## + Fold1: mtry= 2, splitrule=variance
## - Fold1: mtry= 2, splitrule=variance
## + Fold1: mtry= 7, splitrule=variance
## - Fold1: mtry= 7, splitrule=variance
## + Fold1: mtry=12, splitrule=variance
## - Fold1: mtry=12, splitrule=variance
## + Fold1: mtry= 2, splitrule=extratrees
## - Fold1: mtry= 2, splitrule=extratrees
## + Fold1: mtry= 7, splitrule=extratrees
## - Fold1: mtry= 7, splitrule=extratrees
## + Fold1: mtry=12, splitrule=extratrees
## - Fold1: mtry=12, splitrule=extratrees
## + Fold2: mtry= 2, splitrule=variance
## - Fold2: mtry= 2, splitrule=variance
## + Fold2: mtry= 7, splitrule=variance
## - Fold2: mtry= 7, splitrule=variance
## + Fold2: mtry=12, splitrule=variance
```

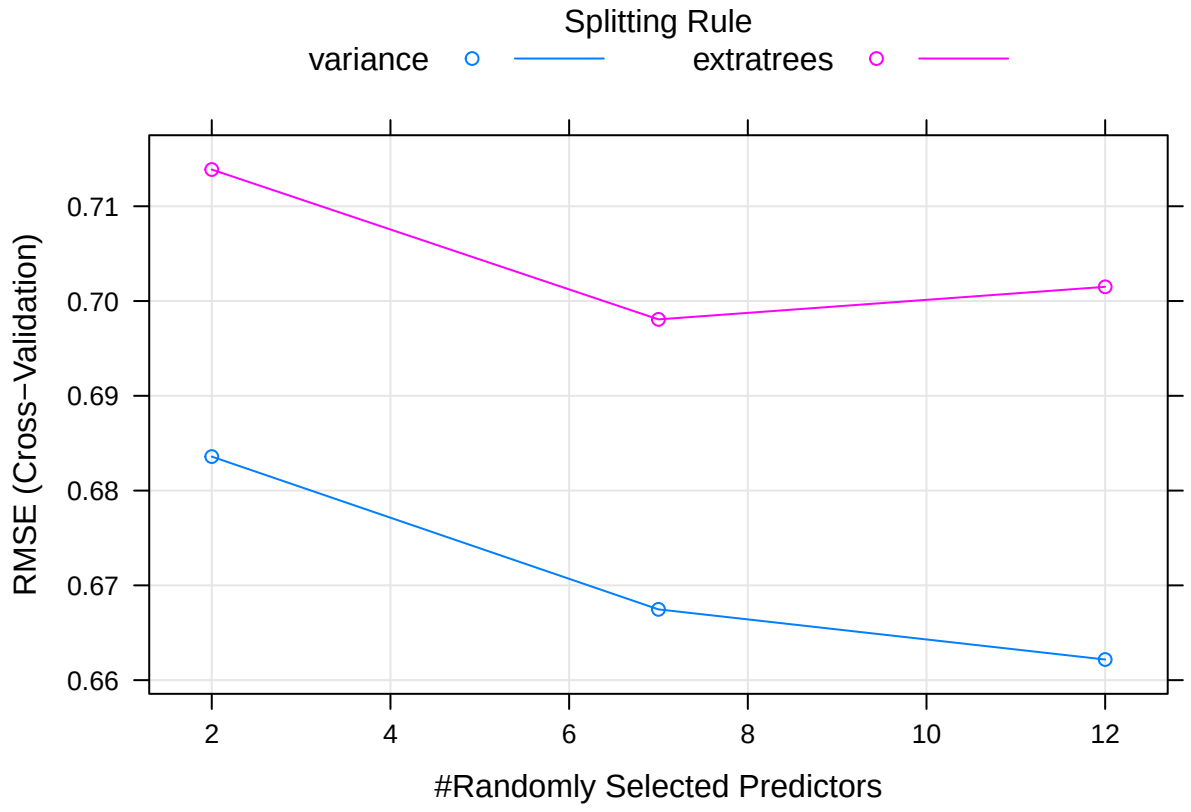
```
## - Fold2: mtry=12, splitrule=variance
## + Fold2: mtry= 2, splitrule=extratrees
## - Fold2: mtry= 2, splitrule=extratrees
## + Fold2: mtry= 7, splitrule=extratrees
## - Fold2: mtry= 7, splitrule=extratrees
## + Fold2: mtry=12, splitrule=extratrees
## - Fold2: mtry=12, splitrule=extratrees
## + Fold3: mtry= 2, splitrule=variance
## - Fold3: mtry= 2, splitrule=variance
## + Fold3: mtry= 7, splitrule=variance
## - Fold3: mtry= 7, splitrule=variance
## + Fold3: mtry=12, splitrule=variance
## - Fold3: mtry=12, splitrule=variance
## + Fold3: mtry= 2, splitrule=extratrees
## - Fold3: mtry= 2, splitrule=extratrees
## + Fold3: mtry= 7, splitrule=extratrees
## - Fold3: mtry= 7, splitrule=extratrees
## + Fold3: mtry=12, splitrule=extratrees
## - Fold3: mtry=12, splitrule=extratrees
## + Fold4: mtry= 2, splitrule=variance
## - Fold4: mtry= 2, splitrule=variance
## + Fold4: mtry= 7, splitrule=variance
## - Fold4: mtry= 7, splitrule=variance
## + Fold4: mtry=12, splitrule=variance
## - Fold4: mtry=12, splitrule=variance
## + Fold4: mtry= 2, splitrule=extratrees
## - Fold4: mtry= 2, splitrule=extratrees
## + Fold4: mtry= 7, splitrule=extratrees
## - Fold4: mtry= 7, splitrule=extratrees
## + Fold4: mtry=12, splitrule=extratrees
## - Fold4: mtry=12, splitrule=extratrees
## + Fold5: mtry= 2, splitrule=variance
## - Fold5: mtry= 2, splitrule=variance
## + Fold5: mtry= 7, splitrule=variance
## - Fold5: mtry= 7, splitrule=variance
## + Fold5: mtry=12, splitrule=variance
## - Fold5: mtry=12, splitrule=variance
## + Fold5: mtry= 2, splitrule=extratrees
## - Fold5: mtry= 2, splitrule=extratrees
## + Fold5: mtry= 7, splitrule=extratrees
## - Fold5: mtry= 7, splitrule=extratrees
## + Fold5: mtry=12, splitrule=extratrees
## - Fold5: mtry=12, splitrule=extratrees
## Aggregating results
## Selecting tuning parameters
## Fitting mtry = 12, splitrule = variance on full training set
```

```
# Print model to console
model
```

```
## Random Forest
##
## 100 samples
##  12 predictor
##
```

```
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 80, 80, 81, 80, 79
## Resampling results across tuning parameters:
##
##   mtry  splitrule   RMSE       Rsquared   MAE
##    2    variance    0.6835887  0.2508610  0.5170790
##    2    extratrees  0.7138749  0.2113001  0.5376460
##    7    variance    0.6674687  0.2907502  0.5100175
##    7    extratrees  0.6980645  0.2258272  0.5296820
##   12    variance    0.6621775  0.2977265  0.5070150
##   12    extratrees  0.7014995  0.2111631  0.5355658
##
## RMSE was used to select the optimal model using  the smallest value.
## The final values used for the model were mtry = 12 and splitrule
##  = variance.
```

```
# Plot model
plot(model)
```



We can pass our own custom tuning grids as a data frame to the train control argument. This is the most fleixble approach, however it can take a lot longer to train the model and can considerably increase run time.

You can provide any number of values for mtry, from 2 up to the number of columns in the dataset. In practice, there are diminishing returns for much larger values of mtry, so we will use a custom tuning grid that explores 2 simple models (mtry = 2 and mtry = 3) as well as one more complicated model (mtry = 7).

```
# Add the tune grid options - note ranger requires a . prior to tuning parameters, other models/package
tgrid <- expand.grid(
  .mtry = c(2, 3, 7),
  .splitrule = "extratrees"
)

# Fit random forest: model
model <- train(
  quality ~.,
  tuneGrid = tgrid,
  data = wine, method = "ranger",
  trControl = trainControl(method = "cv", number = 5, verboseIter = TRUE)
)
```

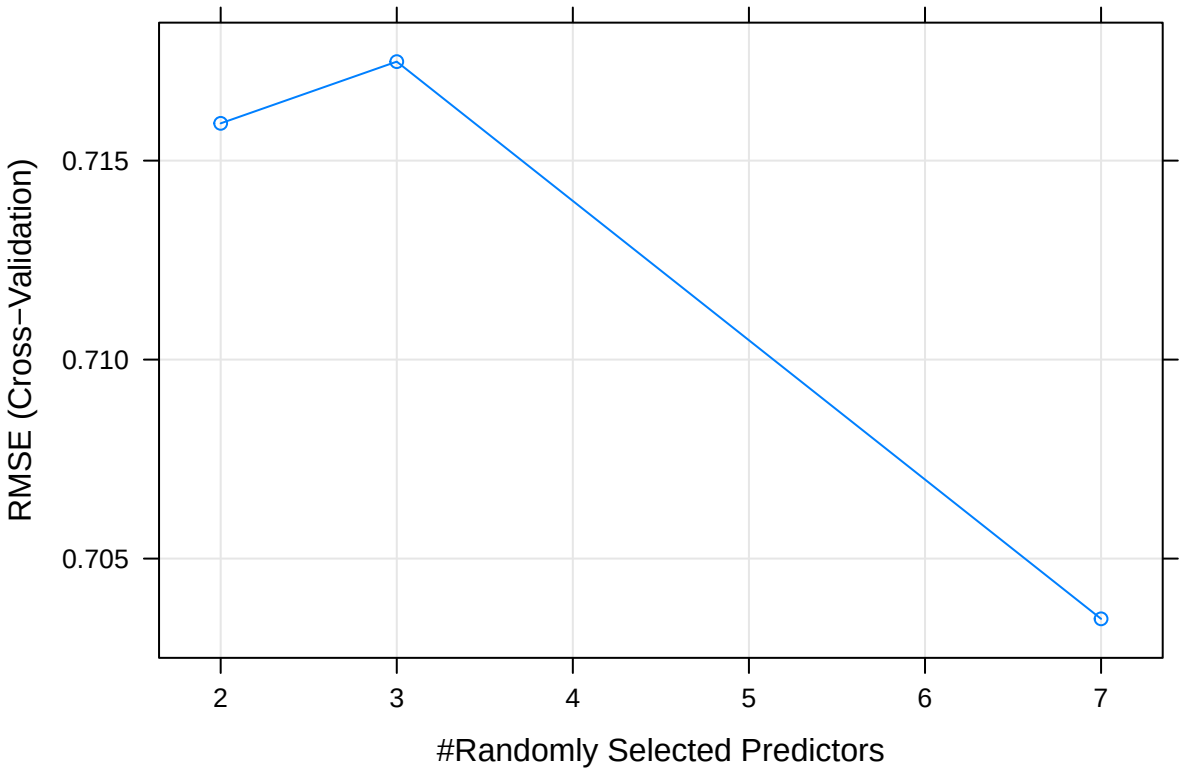
```
## + Fold1: mtry=2, splitrule=extratrees
## - Fold1: mtry=2, splitrule=extratrees
## + Fold1: mtry=3, splitrule=extratrees
## - Fold1: mtry=3, splitrule=extratrees
## + Fold1: mtry=7, splitrule=extratrees
## - Fold1: mtry=7, splitrule=extratrees
## + Fold2: mtry=2, splitrule=extratrees
## - Fold2: mtry=2, splitrule=extratrees
## + Fold2: mtry=3, splitrule=extratrees
## - Fold2: mtry=3, splitrule=extratrees
## + Fold2: mtry=7, splitrule=extratrees
## - Fold2: mtry=7, splitrule=extratrees
## + Fold3: mtry=2, splitrule=extratrees
## - Fold3: mtry=2, splitrule=extratrees
## + Fold3: mtry=3, splitrule=extratrees
## - Fold3: mtry=3, splitrule=extratrees
## + Fold3: mtry=7, splitrule=extratrees
## - Fold3: mtry=7, splitrule=extratrees
## + Fold4: mtry=2, splitrule=extratrees
## - Fold4: mtry=2, splitrule=extratrees
## + Fold4: mtry=3, splitrule=extratrees
## - Fold4: mtry=3, splitrule=extratrees
## + Fold4: mtry=7, splitrule=extratrees
## - Fold4: mtry=7, splitrule=extratrees
## + Fold5: mtry=2, splitrule=extratrees
## - Fold5: mtry=2, splitrule=extratrees
## + Fold5: mtry=3, splitrule=extratrees
## - Fold5: mtry=3, splitrule=extratrees
## + Fold5: mtry=7, splitrule=extratrees
## - Fold5: mtry=7, splitrule=extratrees
## Aggregating results
## Selecting tuning parameters
## Fitting mtry = 7, splitrule = extratrees on full training set
```

```
# Print model to console
model
```

```
## Random Forest
##
## 100 samples
##  12 predictor
```

```
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 80, 80, 80, 79, 81
## Resampling results across tuning parameters:
##
##   mtry  RMSE       Rsquared   MAE
##   2     0.7159368  0.2351616  0.5258831
##   3     0.7174871  0.2189432  0.5313928
##   7     0.7034861  0.2462823  0.5210960
##
## Tuning parameter 'splitrule' was held constant at a value of extratrees
## RMSE was used to select the optimal model using  the smallest value.
## The final values used for the model were mtry = 7 and splitrule
##  = extratrees.
```

```
# Plot model
plot(model)
```



4.2.4 Introducing glmnet

This is similar to GLM models but has built in model selection. It linear models better deal with collinearity, which is correlation amungst the predictors in a model and can help reduce overfitting for models with small data. There are two main forms - lasso and ridge regression. Both of these methods shrink the coefficents towards zero and can also be useful for very large datasets also.

- Lasso - penalises non-zero coefficients (l1 norm)
- Ridge - penalises absolute magnitude coefficients (l2 norm)

The amount that we penalise the coefficients depends on our tuning parameter. We can use cross-validation to help in the selection of the tuning parameter. At very high values of the tuning parameter (lambda), we penalise the indivdual variables so much they effectively have a value of zero. At very low levels, we penalise the variables so little that they approach their linear (least squared) equivalents. In both instances, we try to end up with parsimonous models. We can fit lasso and ridge together using glmnet using the alpha paramater - 1 is pure lasso, 0 is pure ridge.

The dataset used comes has nearly as many columns as rows. This is a simulated dataset based on the "don't overfit" competition on Kaggle a number of years ago. Classification problems are a little more complicated than regression problems because you have to provide a custom summaryFunction to the train() function to use the AUC metric to rank your models. Start by making a custom trainControl. Be sure to set classProbs = TRUE, otherwise the twoClassSummary for summaryFunction will break.

```
# Create custom trainControl: myControl
myControl <- trainControl(
  method = "cv", number = 10,
  summaryFunction = twoClassSummary,
  classProbs = TRUE, # IMPORTANT!
  verboseIter = TRUE
)
```

glmnet is capable of fitting two different kinds of penalized models, controlled by the alpha parameter:

Ridge regression (or alpha = 0) Lasso regression (or alpha = 1)

We now fit a glmnet model to the "don't overfit" dataset using the defaults provided by the caret package.

```
overfit <- read.csv("./files/MLToolbox/overfit.csv")

# Fit glmnet model: model
model <- train(
  y ~., overfit,
  method = "glmnet",
  trControl = myControl
)
```

```
## Warning in train.default(x, y, weights = w, ...): The metric "Accuracy" was
## not in the result set. ROC will be used instead.

## Warning: package 'glmnet' was built under R version 3.4.4

## Loading required package: Matrix

## Loading required package: foreach

## Loaded glmnet 2.0-13

## + Fold01: alpha=0.10, lambda=0.01013
## - Fold01: alpha=0.10, lambda=0.01013
## + Fold01: alpha=0.55, lambda=0.01013
## - Fold01: alpha=0.55, lambda=0.01013
## + Fold01: alpha=1.00, lambda=0.01013
## - Fold01: alpha=1.00, lambda=0.01013
## + Fold02: alpha=0.10, lambda=0.01013
## - Fold02: alpha=0.10, lambda=0.01013
## + Fold02: alpha=0.55, lambda=0.01013
## - Fold02: alpha=0.55, lambda=0.01013
```

```
## + Fold02: alpha=1.00, lambda=0.01013
## - Fold02: alpha=1.00, lambda=0.01013
## + Fold03: alpha=0.10, lambda=0.01013
## - Fold03: alpha=0.10, lambda=0.01013
## + Fold03: alpha=0.55, lambda=0.01013
## - Fold03: alpha=0.55, lambda=0.01013
## + Fold03: alpha=1.00, lambda=0.01013
## - Fold03: alpha=1.00, lambda=0.01013
## + Fold04: alpha=0.10, lambda=0.01013
## - Fold04: alpha=0.10, lambda=0.01013
## + Fold04: alpha=0.55, lambda=0.01013
## - Fold04: alpha=0.55, lambda=0.01013
## + Fold04: alpha=1.00, lambda=0.01013
## - Fold04: alpha=1.00, lambda=0.01013
## + Fold05: alpha=0.10, lambda=0.01013
## - Fold05: alpha=0.10, lambda=0.01013
## + Fold05: alpha=0.55, lambda=0.01013
## - Fold05: alpha=0.55, lambda=0.01013
## + Fold05: alpha=1.00, lambda=0.01013
## - Fold05: alpha=1.00, lambda=0.01013
## + Fold06: alpha=0.10, lambda=0.01013
## - Fold06: alpha=0.10, lambda=0.01013
## + Fold06: alpha=0.55, lambda=0.01013
## - Fold06: alpha=0.55, lambda=0.01013
## + Fold06: alpha=1.00, lambda=0.01013
## - Fold06: alpha=1.00, lambda=0.01013
## + Fold07: alpha=0.10, lambda=0.01013
## - Fold07: alpha=0.10, lambda=0.01013
## + Fold07: alpha=0.55, lambda=0.01013
## - Fold07: alpha=0.55, lambda=0.01013
## + Fold07: alpha=1.00, lambda=0.01013
## - Fold07: alpha=1.00, lambda=0.01013
## + Fold08: alpha=0.10, lambda=0.01013
## - Fold08: alpha=0.10, lambda=0.01013
## + Fold08: alpha=0.55, lambda=0.01013
## - Fold08: alpha=0.55, lambda=0.01013
## + Fold08: alpha=1.00, lambda=0.01013
## - Fold08: alpha=1.00, lambda=0.01013
## + Fold09: alpha=0.10, lambda=0.01013
## - Fold09: alpha=0.10, lambda=0.01013
## + Fold09: alpha=0.55, lambda=0.01013
## - Fold09: alpha=0.55, lambda=0.01013
## + Fold09: alpha=1.00, lambda=0.01013
## - Fold09: alpha=1.00, lambda=0.01013
## + Fold10: alpha=0.10, lambda=0.01013
## - Fold10: alpha=0.10, lambda=0.01013
## + Fold10: alpha=0.55, lambda=0.01013
## - Fold10: alpha=0.55, lambda=0.01013
## + Fold10: alpha=1.00, lambda=0.01013
## - Fold10: alpha=1.00, lambda=0.01013
## Aggregating results
## Selecting tuning parameters
## Fitting alpha = 1, lambda = 0.0101 on full training set
```

```
# Print model to console
model
```

```
## glmnet
##
## 250 samples
## 200 predictors
##   2 classes: 'class1', 'class2'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 225, 225, 225, 224, 224, 226, ...
## Resampling results across tuning parameters:
##
##   alpha  lambda        ROC        Sens  Spec
##   0.10   0.0001012745  0.4420290  0     0.9699275
##   0.10   0.0010127448  0.4462862  0     0.9784420
##   0.10   0.0101274483  0.4461051  0     0.9913043
##   0.55   0.0001012745  0.4247283  0     0.9572464
##   0.55   0.0010127448  0.4223732  0     0.9701087
##   0.55   0.0101274483  0.4367754  0     0.9869565
##   1.00   0.0001012745  0.3647645  0     0.9438406
##   1.00   0.0010127448  0.3645833  0     0.9655797
##   1.00   0.0101274483  0.4499094  0     0.9869565
##
## ROC was used to select the optimal model using  the largest value.
## The final values used for the model were alpha = 1 and lambda = 0.01012745.
```

```
# Print maximum ROC statistic
max(model[["results"]])
```

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

glmnet model have two tuning paramets, alpha (ridge to lasso) and lambda (size of the penalty). However for a single alpha value, glmnet will fit all values of lambda simultaneously. We are fitting a number of different models in effect, then see which fits the best. We could try for instance two values of alpha - 0 and 1 - then have a wide range of lambdas which we can fit using the seqeunce function.

We use a custom tuning grid as the default tuning grid is very small and there are many more potential glmnet models we may want to explore.

As previously mentioned, the glmnet model actually fits many models at once (one of the great things about the package). You can exploit this by passing a large number of lambda values, which control the amount of penalization in the model. train() is smart enough to only fit one model per alpha value and pass all of the lambda values at once for simultaneous fitting.

A good tuning grid for glmnet models is:

expand.grid(alpha = 0:1, lambda = seq(0.0001, 1, length = 100))

This grid explores a large number of lambda values (100, in fact), from a very small one to a very large one. (You could increase the maximum lambda to 10, but in this exercise 1 is a good upper bound.)

If you want to explore fewer models, you can use a shorter lambda sequence. For example, lambda = seq(0.0001, 1, length = 10) would fit 10 models per value of alpha.

You also look at the two forms of penalized models with this tuneGrid: ridge regression and lasso regression. alpha = 0 is pure ridge regression, and alpha = 1 is pure lasso regression. You can fit a mixture of the two

models (i.e. an elastic net) using an alpha between 0 and 1. For example, alpha = .05 would be 95% ridge regression and 5% lasso regression.

In this problem you'll just explore the 2 extremes - pure ridge and pure lasso regression - for the purpose of illustrating their differences.

```
# Train glmnet with custom trainControl and tuning: model
model <- train(
  y ~., overfit,
  tuneGrid = expand.grid(alpha = 0:1,
  lambda = seq(0.0001, 1, length = 20)),
  method = "glmnet",
  trControl = myControl
)
```

```
## Warning in train.default(x, y, weights = w, ...): The metric "Accuracy" was
## not in the result set. ROC will be used instead.
```

```
## + Fold01: alpha=0, lambda=1
## - Fold01: alpha=0, lambda=1
## + Fold01: alpha=1, lambda=1
## - Fold01: alpha=1, lambda=1
## + Fold02: alpha=0, lambda=1
## - Fold02: alpha=0, lambda=1
## + Fold02: alpha=1, lambda=1
## - Fold02: alpha=1, lambda=1
## + Fold03: alpha=0, lambda=1
## - Fold03: alpha=0, lambda=1
## + Fold03: alpha=1, lambda=1
## - Fold03: alpha=1, lambda=1
## + Fold04: alpha=0, lambda=1
## - Fold04: alpha=0, lambda=1
## + Fold04: alpha=1, lambda=1
## - Fold04: alpha=1, lambda=1
## + Fold05: alpha=0, lambda=1
## - Fold05: alpha=0, lambda=1
## + Fold05: alpha=1, lambda=1
## - Fold05: alpha=1, lambda=1
## + Fold06: alpha=0, lambda=1
## - Fold06: alpha=0, lambda=1
## + Fold06: alpha=1, lambda=1
## - Fold06: alpha=1, lambda=1
## + Fold07: alpha=0, lambda=1
## - Fold07: alpha=0, lambda=1
## + Fold07: alpha=1, lambda=1
## - Fold07: alpha=1, lambda=1
## + Fold08: alpha=0, lambda=1
## - Fold08: alpha=0, lambda=1
## + Fold08: alpha=1, lambda=1
## - Fold08: alpha=1, lambda=1
## + Fold09: alpha=0, lambda=1
## - Fold09: alpha=0, lambda=1
## + Fold09: alpha=1, lambda=1
## - Fold09: alpha=1, lambda=1
## + Fold10: alpha=0, lambda=1
## - Fold10: alpha=0, lambda=1
```

```
## + Fold10: alpha=1, lambda=1
## - Fold10: alpha=1, lambda=1
## Aggregating results
## Selecting tuning parameters
## Fitting alpha = 1, lambda = 0.0527 on full training set
```

```
# Print model to console
model
```

```
## glmnet
##
## 250 samples
## 200 predictors
##   2 classes: 'class1', 'class2'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 224, 226, 226, 225, 225, 225, ...
## Resampling results across tuning parameters:
##
##   alpha  lambda      ROC        Sens  Spec
##   0      0.00010000  0.3849638  0.00  0.9699275
##   0      0.05272632  0.4043478  0.00  1.0000000
##   0      0.10535263  0.4024457  0.00  1.0000000
##   0      0.15797895  0.4002717  0.00  1.0000000
##   0      0.21060526  0.4133152  0.00  1.0000000
##   0      0.26323158  0.4133152  0.00  1.0000000
##   0      0.31585789  0.4133152  0.00  1.0000000
##   0      0.36848421  0.4133152  0.00  1.0000000
##   0      0.42111053  0.4154891  0.00  1.0000000
##   0      0.47373684  0.4196558  0.00  1.0000000
##   0      0.52636316  0.4217391  0.00  1.0000000
##   0      0.57898947  0.4217391  0.00  1.0000000
##   0      0.63161579  0.4217391  0.00  1.0000000
##   0      0.68424211  0.4217391  0.00  1.0000000
##   0      0.73686842  0.4217391  0.00  1.0000000
##   0      0.78949474  0.4196558  0.00  1.0000000
##   0      0.84212105  0.4174819  0.00  1.0000000
##   0      0.89474737  0.4153986  0.00  1.0000000
##   0      0.94737368  0.4153986  0.00  1.0000000
##   0      1.00000000  0.4153986  0.00  1.0000000
##   1      0.00010000  0.3384058  0.05  0.9018116
##   1      0.05272632  0.5086957  0.00  1.0000000
##   1      0.10535263  0.5000000  0.00  1.0000000
##   1      0.15797895  0.5000000  0.00  1.0000000
##   1      0.21060526  0.5000000  0.00  1.0000000
##   1      0.26323158  0.5000000  0.00  1.0000000
##   1      0.31585789  0.5000000  0.00  1.0000000
##   1      0.36848421  0.5000000  0.00  1.0000000
##   1      0.42111053  0.5000000  0.00  1.0000000
##   1      0.47373684  0.5000000  0.00  1.0000000
##   1      0.52636316  0.5000000  0.00  1.0000000
##   1      0.57898947  0.5000000  0.00  1.0000000
##   1      0.63161579  0.5000000  0.00  1.0000000
##   1      0.68424211  0.5000000  0.00  1.0000000
```

```
##   1      0.73686842  0.5000000  0.00  1.0000000
##   1      0.78949474  0.5000000  0.00  1.0000000
##   1      0.84212105  0.5000000  0.00  1.0000000
##   1      0.89474737  0.5000000  0.00  1.0000000
##   1      0.94737368  0.5000000  0.00  1.0000000
##   1      1.00000000  0.5000000  0.00  1.0000000
##
## ROC was used to select the optimal model using  the largest value.
## The final values used for the model were alpha = 1 and lambda = 0.05272632.
```

```
# Print maximum ROC statistic
max(model[["results"]][["ROC"]])
```

```
## [1] 0.5086957
```

4.3 Preprocessing your data

Real world data has missing values and often models don't know what to missing data. Models can exclude cases where values are missing and it can lead to bias. A better approach is to use median imputation.

In this section we use a version of the Wisconsin Breast Cancer dataset. This dataset presents a classic binary classification problem: 50% of the samples are benign, 50% are malignant, and the challenge is to identify which are which.

This dataset is interesting because many of the predictors contain missing values and most rows of the dataset have at least one missing value. This presents a modeling challenge, because most machine learning algorithms cannot handle missing values out of the box. For example, your first instinct might be to fit a logistic regression model to this data, but prior to doing this you need a strategy for handling the NAs.

Fortunately, the train() function in caret contains an argument called preProcess, which allows you to specify that median imputation should be used to fill in the missing values. In previous chapters, you created models with the train() function using formulas such as y ~ .. An alternative way is to specify the x and y arguments to train(), where x is an object with samples in rows and features in columns and y is a numeric or factor vector containing the outcomes. Said differently, x is a matrix or data frame that contains the whole dataset you'd use for the data argument to the lm() call, for example, but excludes the response variable column; y is a vector that contains just the response variable column.

For this exercise, the argument x to train() is loaded in your workspace as breast_cancer_x and y as breast_cancer_y.

```
# Load the breast cancer data - loads two datasets x to train as breast_cancer_x and y as breast_cancer
load("./files/MLToolbox/BreastCancer.RData")

# Apply median imputation: model
model <- train(
  x = breast_cancer_x, y = breast_cancer_y,
  method = "glm",
  trControl = myControl,
  preProcess = "medianImpute"
)
```

```
## Warning in train.default(x = breast_cancer_x, y = breast_cancer_y, method =
## "glm", : The metric "Accuracy" was not in the result set. ROC will be used
## instead.
```

```
## + Fold01: parameter=none
## - Fold01: parameter=none
```

```
## + Fold02: parameter=none
## - Fold02: parameter=none
## + Fold03: parameter=none
## - Fold03: parameter=none
## + Fold04: parameter=none
## - Fold04: parameter=none
## + Fold05: parameter=none
## - Fold05: parameter=none
## + Fold06: parameter=none
## - Fold06: parameter=none
## + Fold07: parameter=none
## - Fold07: parameter=none
## + Fold08: parameter=none
## - Fold08: parameter=none
## + Fold09: parameter=none
## - Fold09: parameter=none
## + Fold10: parameter=none
## - Fold10: parameter=none
## Aggregating results
## Fitting final model on full training set
```

```
# Print model to console
model
```

```
## Generalized Linear Model
##
## 699 samples
##   9 predictor
##   2 classes: 'benign', 'malignant'
##
## Pre-processing: median imputation (9)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 630, 630, 629, 629, 629, 629, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.9915576  0.9694203  0.9376667
```

There are some problems with median imputation, particularly if there is systematic bias and the data is missing not at random. In effect this means if there is a pattern in missing values, median imputation can miss this. For instance, we often find that higher income individuals are less likely to report their household income in survey data. Note that tree based models tend to be more robust to missing not at random case. An alternative to median imputation is knn imputation, it imputes based on "similar" non missing rows based on some defined variables which we define. KNN is not always better than median imputation, so it is worthwhile trying both knn and median imputation and keep the one which gives the most accurate model.

While this is a lot more complicated to implement in practice than simple median imputation, it is very easy to explore in caret using the preProcess argument to train(). You can simply use preProcess = "knnImpute" to change the method of imputation used prior to model fitting.

```
library("RANN") # installed from dev verison on github

# Apply KNN imputation: model2
model2 <- train(
  x = breast_cancer_x, y = breast_cancer_y,
  method = "glm",
```

```
  trControl = myControl,
  preProcess = "knnImpute"
)
```

```
## Warning in train.default(x = breast_cancer_x, y = breast_cancer_y, method =
## "glm", : The metric "Accuracy" was not in the result set. ROC will be used
## instead.
```

```
## + Fold01: parameter=none
## - Fold01: parameter=none
## + Fold02: parameter=none
## - Fold02: parameter=none
## + Fold03: parameter=none
## - Fold03: parameter=none
## + Fold04: parameter=none
## - Fold04: parameter=none
## + Fold05: parameter=none
## - Fold05: parameter=none
## + Fold06: parameter=none
## - Fold06: parameter=none
## + Fold07: parameter=none
## - Fold07: parameter=none
## + Fold08: parameter=none
## - Fold08: parameter=none
## + Fold09: parameter=none
## - Fold09: parameter=none
## + Fold10: parameter=none
## - Fold10: parameter=none
## Aggregating results
## Fitting final model on full training set
```

```
# Print model to console
model2
```

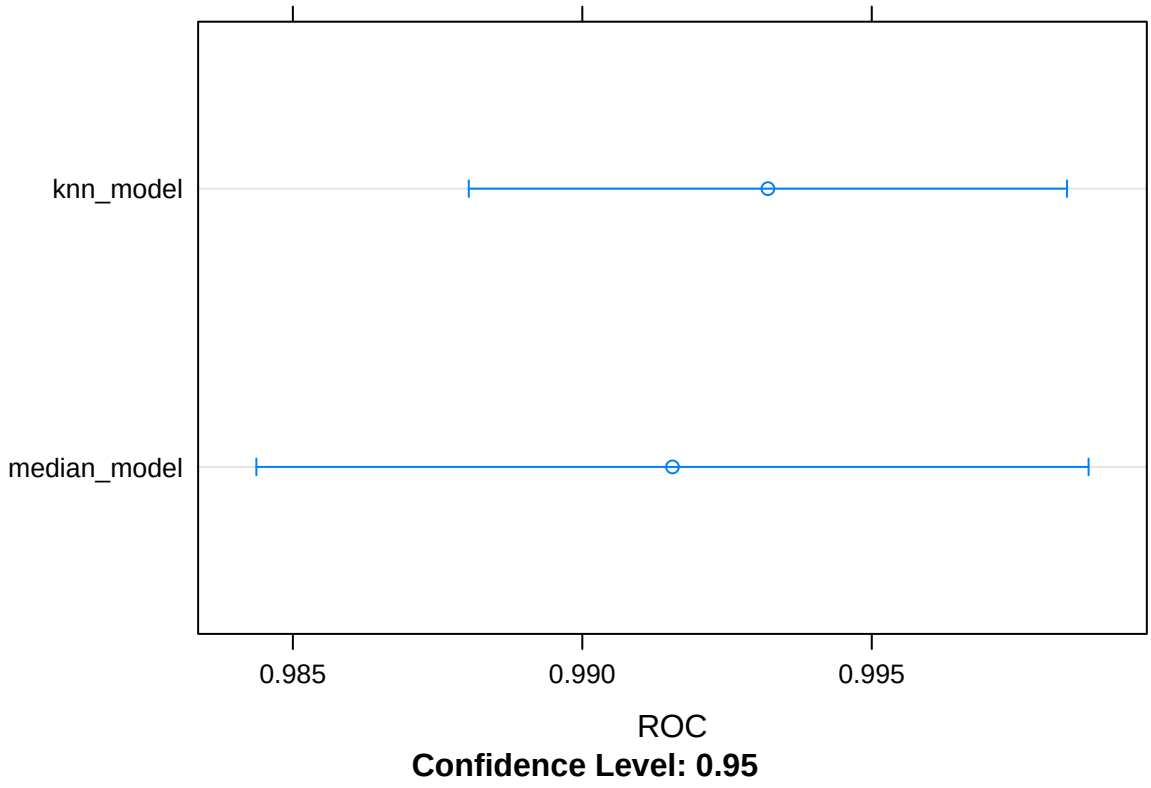
```
## Generalized Linear Model
##
## 699 samples
##   9 predictor
##   2 classes: 'benign', 'malignant'
##
## Pre-processing: nearest neighbor imputation (9), centered (9), scaled (9)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 629, 628, 629, 629, 629, 630, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.9932061  0.9715459  0.9253333
```

All of the preprocessing steps in the train() function happen in the training set of each cross-validation fold, so the error metrics reported include the effects of the preprocessing.

This includes the imputation method used (e.g. knnImputate or medianImpute). This is useful because it allows you to compare different methods of imputation and choose the one that performs the best out-of-sample.

We can use resamples to get resampled results of the models, then use dotplot to look at the results.

```
# Create a set of resampling results from the dataset for each model
resamples <- resamples(x = list(median_model = model, knn_model = model2))

# Plot the results
dotplot(resamples, metric = "ROC")
```



4.3.1 Mutiple pre-processing steps

The preProcess command has mutiple options, and different options can be chained together e.g. medin imputation, centre and scale your data, then fit a GLM model. One set of preprocessing functions that is particularly useful for fitting regression models is standardization: centering and scaling. You first center by subtracting the mean of each column from each value in that column, then you scale by dividing by the standard deviation.

Standardization transforms your data such that for each column, the mean is 0 and the standard deviation is 1. This makes it easier for regression models to find a good solution.

Note that the order of operations matters e.g. centre and scaling should happen after median imputation, PCA should happen after this.

```
# Fit glm with median imputation: model1
model1 <- train(
  x = breast_cancer_x, y = breast_cancer_y,
  method = "glm",
  trControl = myControl,
  preProcess = "medianImpute"
```

```
)
```

```
## Warning in train.default(x = breast_cancer_x, y = breast_cancer_y, method =
## "glm", : The metric "Accuracy" was not in the result set. ROC will be used
## instead.
```

```
## + Fold01: parameter=none
## - Fold01: parameter=none
## + Fold02: parameter=none
## - Fold02: parameter=none
## + Fold03: parameter=none
## - Fold03: parameter=none
## + Fold04: parameter=none
## - Fold04: parameter=none
## + Fold05: parameter=none
## - Fold05: parameter=none
## + Fold06: parameter=none
## - Fold06: parameter=none
## + Fold07: parameter=none
## - Fold07: parameter=none
## + Fold08: parameter=none
## - Fold08: parameter=none
## + Fold09: parameter=none
## - Fold09: parameter=none
## + Fold10: parameter=none
## - Fold10: parameter=none
## Aggregating results
## Fitting final model on full training set
```

```
# Print model1
model1
```

```
## Generalized Linear Model
##
## 699 samples
##   9 predictor
##   2 classes: 'benign', 'malignant'
##
## Pre-processing: median imputation (9)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 630, 629, 629, 630, 629, 629, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.9904964  0.9695652  0.9418333
```

```
# Fit glm with median imputation and standardization: model2
model2 <- train(
  x = breast_cancer_x, y = breast_cancer_y,
  method = "glm",
  trControl = myControl,
  preProcess = c("medianImpute", "center", "scale")
)
```

```
## Warning in train.default(x = breast_cancer_x, y = breast_cancer_y, method =
## "glm", : The metric "Accuracy" was not in the result set. ROC will be used
```

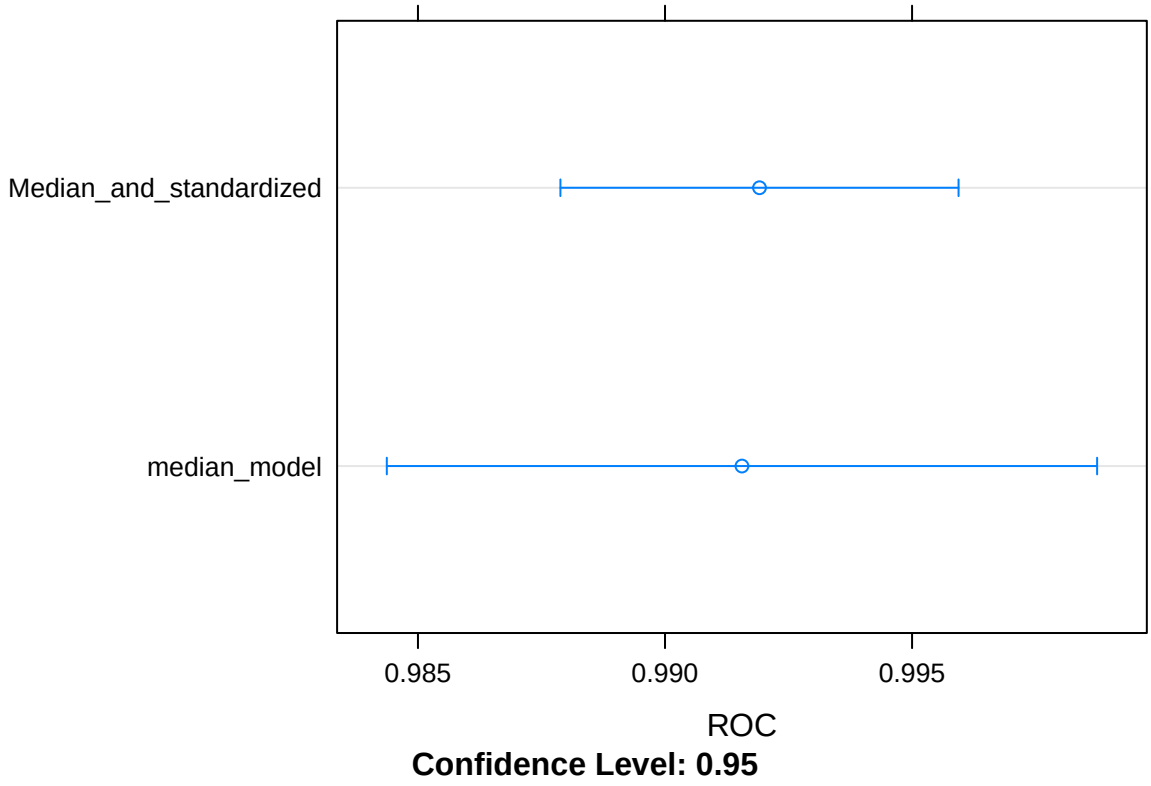
```
## instead.
```

```
## + Fold01: parameter=none
## - Fold01: parameter=none
## + Fold02: parameter=none
## - Fold02: parameter=none
## + Fold03: parameter=none
## - Fold03: parameter=none
## + Fold04: parameter=none
## - Fold04: parameter=none
## + Fold05: parameter=none
## - Fold05: parameter=none
## + Fold06: parameter=none
## - Fold06: parameter=none
## + Fold07: parameter=none
## - Fold07: parameter=none
## + Fold08: parameter=none
## - Fold08: parameter=none
## + Fold09: parameter=none
## - Fold09: parameter=none
## + Fold10: parameter=none
## - Fold10: parameter=none
## Aggregating results
## Fitting final model on full training set
```

```
# Print model2
model2
```

```
## Generalized Linear Model
##
## 699 samples
##   9 predictor
##   2 classes: 'benign', 'malignant'
##
## Pre-processing: median imputation (9), centered (9), scaled (9)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 629, 629, 630, 629, 629, 629, ...
## Resampling results:
##
##   ROC        Sens      Spec
##   0.9919122  0.969372  0.9375
```

```
# Create a set of resampling results from the dataset for each model
resamples <- resamples(x = list(median_model = model, Median_and_standardized = model2))

# Plot the results
dotplot(resamples, metric = "ROC")
```



4.3.2 Handling low-Information Predictors

Some variables in our dataset may contain very little information e.g. variables that are constant or close to constant. Constant models can cause problems, as a CV fold may end up with one when performing a split. It can be hard to work out what has gone wrong with such a model, as the results are often meaningless, as the metrics can be missing.

To remove constant value columns, we can use the "zv" or zero variance option or "nzv" to remove the nearly constant columns in the preProcess step. Removing such columns/variables reduces model-fitting time without reducing model accuracy.

caret contains a utility function called nearZeroVar() for removing such variables to save time during modeling.

nearZeroVar() takes in data x, then looks at the ratio of the most common value to the second most common value, freqCut, and the percentage of distinct values out of the number of total samples, uniqueCut. By default, caret uses freqCut = 19 and uniqueCut = 10, which is fairly conservative. I like to be a little more aggressive and use freqCut = 2 and uniqueCut = 20 when calling nearZeroVar().

We will use the blood-brain dataset. This is a biochemical dataset in which the task is to predict the following value for a set of biochemical compounds:

log((concentration of compound in brain) / (concentration of compound in blood))

This gives a quantitative metric of the compound's ability to cross the blood-brain barrier, and is useful for understanding the biological properties of that barrier.

One interesting aspect of this dataset is that it contains many variables and many of these variables have

extemely low variances. This means that there is very little information in these variables because they mostly consist of a single value (e.g. zero).

```
# Load the blood brain dataset
load("./files/MLToolbox/BloodBrain.RData")

# Identify near zero variance predictors: remove_cols
remove_cols <- nearZeroVar(bloodbrain_x, names = TRUE,
                           freqCut = 2, uniqueCut = 20)

# Get all column names from bloodbrain_x: all_cols
all_cols <- names(bloodbrain_x)

# Remove from data: bloodbrain_x_small
bloodbrain_x_small <- bloodbrain_x[ , setdiff(all_cols, remove_cols)]
```

Note you can either do this by hand, prior to modeling, using the nearZeroVar() function as shown above. Or this can also be done using the preProcess argument equal to "nzv". Doing it by hand using nerZeroVar() gives more fine grained control however.

Next we can can fit a glm model to it using the train() function. This model will run faster than using the full dataset and will yield very similar predictive accuracy.

```
# Fit model on reduced data: model
model <- train(x = bloodbrain_x_small, y = bloodbrain_y, method = "glm")
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading

## Warning in predict.lm(object, newdata, se.fit, scale = 1, type =
## ifelse(type == : prediction from a rank-deficient fit may be misleading
```

```r
# Print model to console
model
```

```
## Generalized Linear Model
##
## 208 samples
## 112 predictors
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
```

```
## Summary of sample sizes: 208, 208, 208, 208, 208, 208, ...
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   1.699781  0.1032369  1.142914
```

4.3.3 Principle Components Analysis (PCA)

This can be used for pre-processing and combines low variance, correlated variables together. They result in perpendicular predictors.

This is an alternative to removing low-variance predictors is to run PCA on your dataset. This is sometimes preferable because it does not throw out all of your data: many different low variance predictors may end up combined into one high variance PCA variable, which might have a positive impact on your model's accuracy.

This is an especially good trick for linear models: the pca option in the preProcess argument will center and scale your data, combine low variance variables, and ensure that all of your predictors are orthogonal. This creates an ideal dataset for linear regression modeling, and can often improve the accuracy of your models.

```
# Fit glm model using PCA: model
model <- train(
  x = bloodbrain_x, y = bloodbrain_y,
  method = "glm", preProcess = "pca"
)

# Print model to console
model
```

```
## Generalized Linear Model
##
## 208 samples
## 132 predictors
##
## Pre-processing: principal component signal extraction (132),
##  centered (132), scaled (132)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 208, 208, 208, 208, 208, 208, ...
## Resampling results:
##
##   RMSE       Rsquared   MAE
##   0.6021496  0.4360237  0.4553297
```

4.4 Selecting models: a case study in churn prediction

In this section we will work with a more realistic dataset - customer churn in a telecoms company. So we can compare different models, we are going to use the same settup for each model - same same number of train and test splits for instance. To do this we can define a trainControl object which will specify which rows are used for modelling and which rows are used for hold outs.

```
# Load the churn data
load("./files/MLToolbox/Churn.RData")

# Create custom indices: myFolds
```

```
myFolds <- createFolds(churn_y, k = 5)

# Create reusable trainControl object: myControl
myControl <- trainControl(
  summaryFunction = twoClassSummary,
  classProbs = TRUE, # IMPORTANT!
  verboseIter = TRUE,
  savePredictions = TRUE,
  index = myFolds
)
```

We will first use the glmnet model. Like linear models, this can be easily interpretable and can make a good starting point for modelling. We can look at the outputs to understand key drivers of churn. IT fits quickly and can provide very accurate models. glmnet models are therefore simple, fast and interpretable.

```
# Fit glmnet model: model_glmnet
model_glmnet <- train(
  x = churn_x, y = churn_y,
  metric = "ROC",
  method = "glmnet",
  trControl = myControl
)
```

```
## + Fold1: alpha=0.10, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold1: alpha=0.10, lambda=0.01821
## + Fold1: alpha=0.55, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold1: alpha=0.55, lambda=0.01821
## + Fold1: alpha=1.00, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold1: alpha=1.00, lambda=0.01821
## + Fold2: alpha=0.10, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold2: alpha=0.10, lambda=0.01821
## + Fold2: alpha=0.55, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold2: alpha=0.55, lambda=0.01821
## + Fold2: alpha=1.00, lambda=0.01821
```

```
## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold2: alpha=1.00, lambda=0.01821
## + Fold3: alpha=0.10, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold3: alpha=0.10, lambda=0.01821
## + Fold3: alpha=0.55, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold3: alpha=0.55, lambda=0.01821
## + Fold3: alpha=1.00, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold3: alpha=1.00, lambda=0.01821
## + Fold4: alpha=0.10, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold4: alpha=0.10, lambda=0.01821
## + Fold4: alpha=0.55, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold4: alpha=0.55, lambda=0.01821
## + Fold4: alpha=1.00, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold4: alpha=1.00, lambda=0.01821
## + Fold5: alpha=0.10, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold5: alpha=0.10, lambda=0.01821
## + Fold5: alpha=0.55, lambda=0.01821

## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground

## - Fold5: alpha=0.55, lambda=0.01821
## + Fold5: alpha=1.00, lambda=0.01821
```

```
## Warning in lognet(x, is.sparse, ix, jx, y, weights, offset, alpha, nobs, :
## one multinomial or binomial class has fewer than 8 observations; dangerous
## ground
```

```
## - Fold5: alpha=1.00, lambda=0.01821
## Aggregating results
## Selecting tuning parameters
## Fitting alpha = 1, lambda = 0.0182 on full training set
```

Next we are going try a random forest model, which can often be a useful second step after glmnet. Random forests combines an ensemble of non-linear decision trees into a highly flexible (and usually quite accurate) model. They are slower than glmnet and a bit more 'black box' compared to glmnet, but in some situations they can yield much more accurate parameters without much tuning. They also require very little preprocesing. They also capture threshold effects and interactions by default, both of which occur often in real world situations.

```
# Fit random forest: model_rf
model_rf <- train(
  x = churn_x, y = churn_y,
  metric = "ROC",
  method = "ranger",
  trControl = myControl
)
```

```
## + Fold1: mtry= 2, splitrule=gini
## - Fold1: mtry= 2, splitrule=gini
## + Fold1: mtry=36, splitrule=gini
## - Fold1: mtry=36, splitrule=gini
## + Fold1: mtry=70, splitrule=gini
## - Fold1: mtry=70, splitrule=gini
## + Fold1: mtry= 2, splitrule=extratrees
## - Fold1: mtry= 2, splitrule=extratrees
## + Fold1: mtry=36, splitrule=extratrees
## - Fold1: mtry=36, splitrule=extratrees
## + Fold1: mtry=70, splitrule=extratrees
## - Fold1: mtry=70, splitrule=extratrees
## + Fold2: mtry= 2, splitrule=gini
## - Fold2: mtry= 2, splitrule=gini
## + Fold2: mtry=36, splitrule=gini
## - Fold2: mtry=36, splitrule=gini
## + Fold2: mtry=70, splitrule=gini
## - Fold2: mtry=70, splitrule=gini
## + Fold2: mtry= 2, splitrule=extratrees
## - Fold2: mtry= 2, splitrule=extratrees
## + Fold2: mtry=36, splitrule=extratrees
## - Fold2: mtry=36, splitrule=extratrees
## + Fold2: mtry=70, splitrule=extratrees
## - Fold2: mtry=70, splitrule=extratrees
## + Fold3: mtry= 2, splitrule=gini
## - Fold3: mtry= 2, splitrule=gini
## + Fold3: mtry=36, splitrule=gini
## - Fold3: mtry=36, splitrule=gini
## + Fold3: mtry=70, splitrule=gini
## - Fold3: mtry=70, splitrule=gini
## + Fold3: mtry= 2, splitrule=extratrees
## - Fold3: mtry= 2, splitrule=extratrees
```

```
## + Fold3: mtry=36, splitrule=extratrees
## - Fold3: mtry=36, splitrule=extratrees
## + Fold3: mtry=70, splitrule=extratrees
## - Fold3: mtry=70, splitrule=extratrees
## + Fold4: mtry= 2, splitrule=gini
## - Fold4: mtry= 2, splitrule=gini
## + Fold4: mtry=36, splitrule=gini
## - Fold4: mtry=36, splitrule=gini
## + Fold4: mtry=70, splitrule=gini
## - Fold4: mtry=70, splitrule=gini
## + Fold4: mtry= 2, splitrule=extratrees
## - Fold4: mtry= 2, splitrule=extratrees
## + Fold4: mtry=36, splitrule=extratrees
## - Fold4: mtry=36, splitrule=extratrees
## + Fold4: mtry=70, splitrule=extratrees
## - Fold4: mtry=70, splitrule=extratrees
## + Fold5: mtry= 2, splitrule=gini
## - Fold5: mtry= 2, splitrule=gini
## + Fold5: mtry=36, splitrule=gini
## - Fold5: mtry=36, splitrule=gini
## + Fold5: mtry=70, splitrule=gini
## - Fold5: mtry=70, splitrule=gini
## + Fold5: mtry= 2, splitrule=extratrees
## - Fold5: mtry= 2, splitrule=extratrees
## + Fold5: mtry=36, splitrule=extratrees
## - Fold5: mtry=36, splitrule=extratrees
## + Fold5: mtry=70, splitrule=extratrees
## - Fold5: mtry=70, splitrule=extratrees
## Aggregating results
## Selecting tuning parameters
## Fitting mtry = 36, splitrule = extratrees on full training set
```

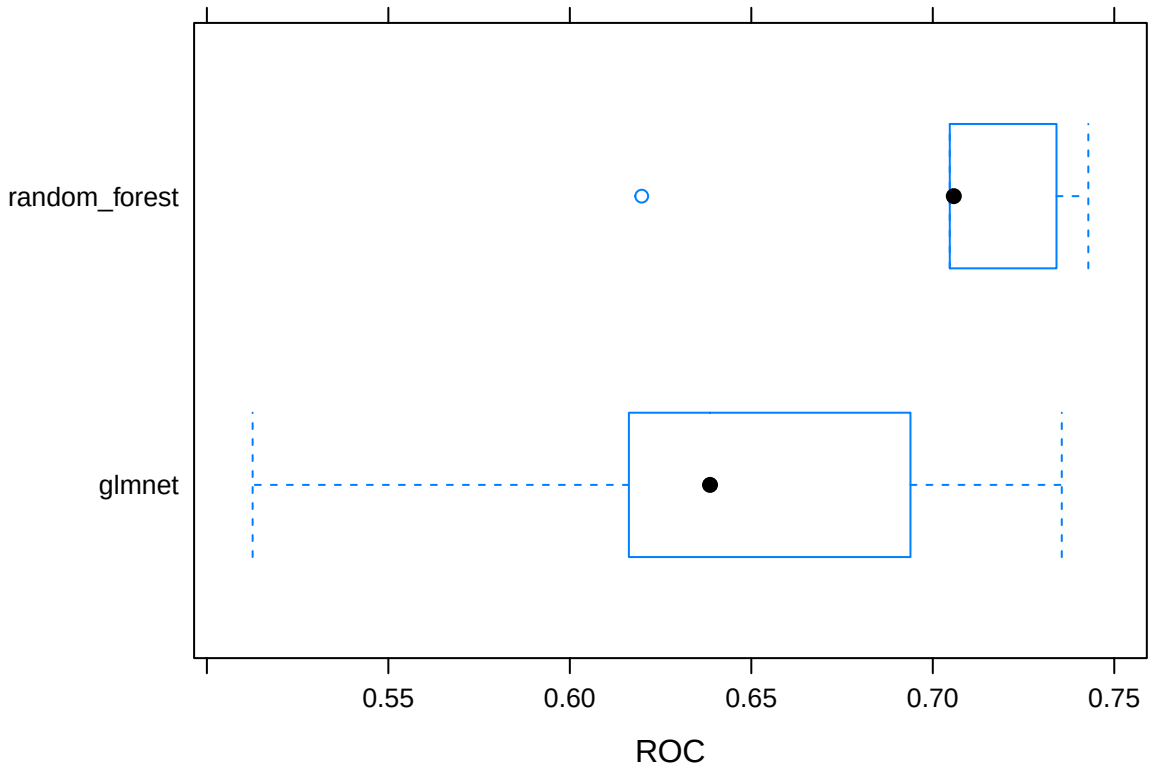
After fitting our models we have to determine which fits the data the best, ensuring they have been trained and tested on the same data. We want to select the model with the highest AUC, but with as little variation (standard deviation) in AUC. To do this, we use the resamples command from caret, provided they have the same training data and use the same trainControl object with preset cross-validation folds. resamples() takes as input a list of models and can be used to compare dozens of models at once.

```
# Create model_list
model_list <- list(glmnet = model_glmnet, random_forest = model_rf)

# Pass model_list to resamples(): resamples
resamples <- resamples(model_list)

# Summarize the results
summary(resamples)
```

```
##
## Call:
## summary.resamples(object = resamples)
##
## Models: glmnet, random_forest
## Number of resamples: 5
##
## ROC
```

```
##                   Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## glmnet        0.5125995 0.6162688 0.6386286 0.6393789 0.6938550 0.7355429
## random_forest 0.6198055 0.7046857 0.7057913 0.7014449 0.7340849 0.7428571
##               NA's
## glmnet           0
## random_forest    0
##
## Sens
##                   Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## glmnet        0.8742857 0.9137931 0.9367816 0.9392447 0.9828571 0.9885057
## random_forest 0.9828571 0.9885057 0.9885057 0.9919737 1.0000000 1.0000000
##               NA's
## glmnet           0
## random_forest    0
##
## Spec
##               Min. 1st Qu. Median       Mean    3rd Qu. Max. NA's
## glmnet           0       0   0.08 0.12707692 0.11538462 0.44    0
## random_forest    0       0   0.00 0.02369231 0.03846154 0.08    0
```

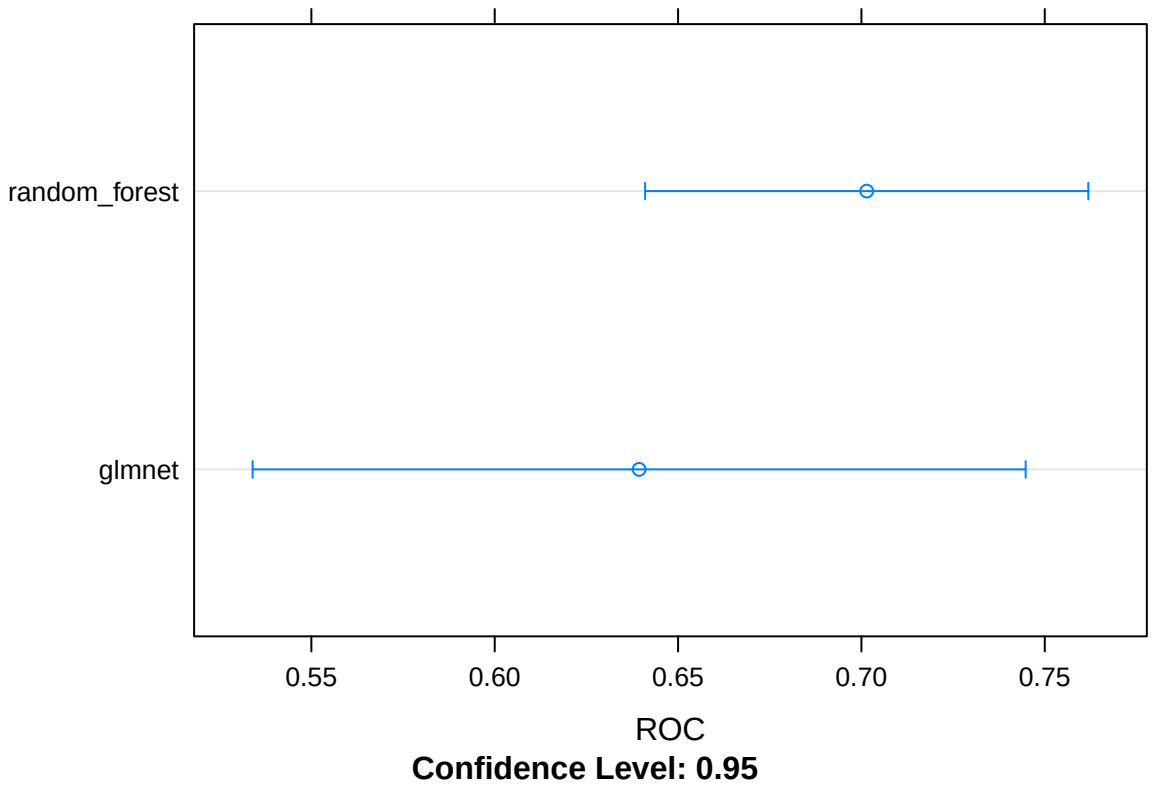
```
# Boxplot of ROC for the models
bwplot(resamples, metric = "ROC")
```



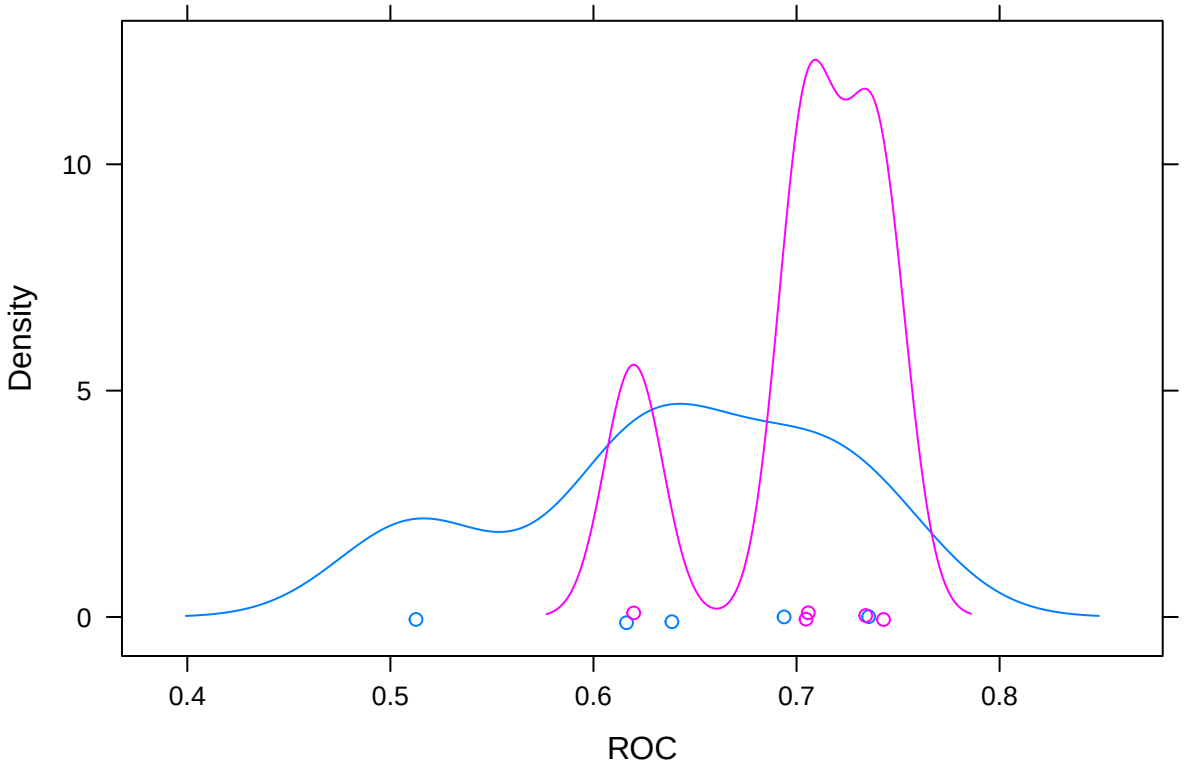
Resamples provides a lot of functions for comparing models, one of them is the boxplot (bwplot) shown above. Or a dot plot as shown below, which is similar to the box plot but in a simpler and can be used whcn comparing many models. We can also get a full density plot of the results using kernel density scores, which can be useful for looking at outliers in particular folds. We can also use a scatter plot for each resample

(fold) which highlights which model had the most accurate prediction for each fold (random forest had them for every fold in this plot, shown below).

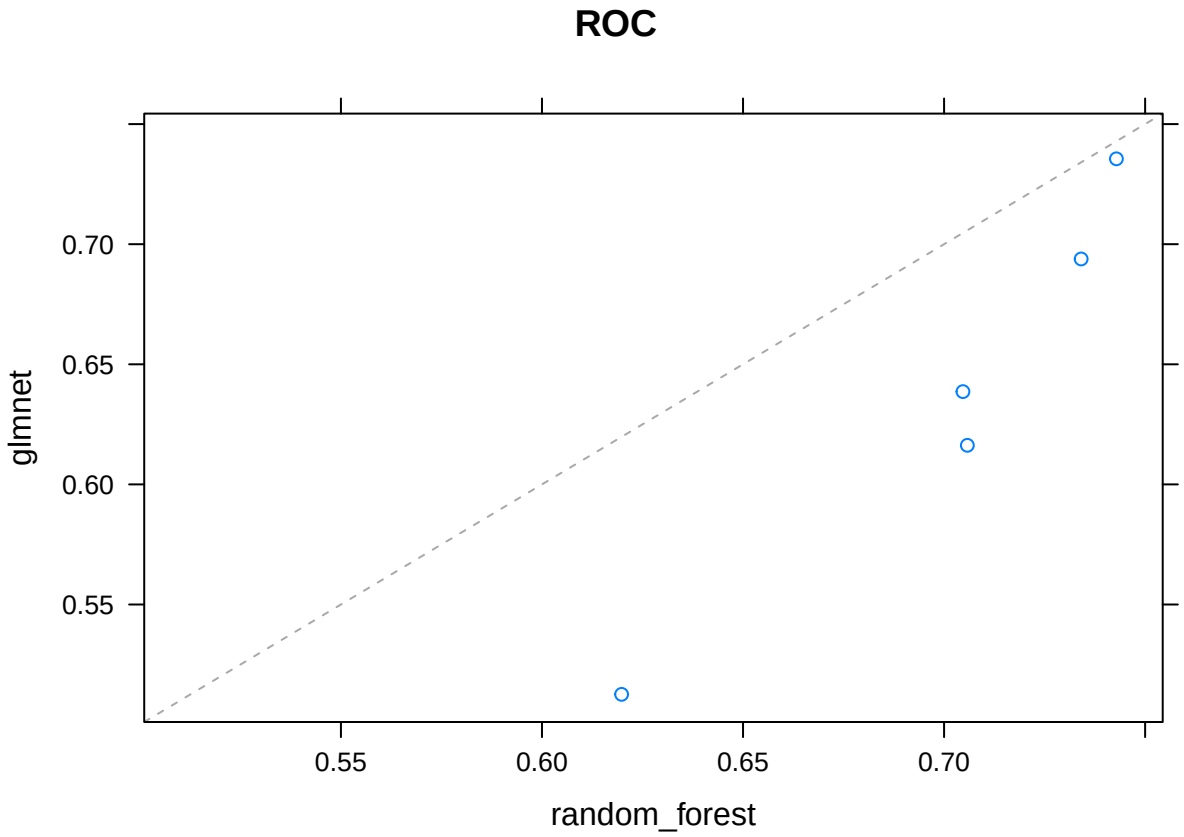
```
# Dotplot of ROC for the models
dotplot(resamples, metric = "ROC")
```




```
# Density plot of ROC for the models
densityplot(resamples, metric = "ROC")
```




```
# Scatterplot of ROC for the models
xyplot(resamples, metric = "ROC")
```


ROC
glmnet
0.70
0.65
0.60
0.55
0.55
0.60
0.65
0.70
random_forest

Bibliography