Wrangling F1 Data With R

A Data Junkie's Guide

Tony Hirst

Wrangling F1 Data With R

A Data Junkie's Guide

Tony Hirst

This book is for sale at http://leanpub.com/wranglingf1datawithr

This version was published on 2014-11-09

Tweet This Book!

Please help Tony Hirst by spreading the word about this book on Twitter!

The suggested hashtag for this book is #f1datajunkie.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#f1datajunkie

Thanks... Just because... (sic)

Acknowledgements

FORMULA 1, FORMULA ONE, F1, FIA FORMULA ONE WORLD CHAMPIONSHIP, GRAND PRIX, F1 GRAND PRIX, FORMULA 1 GRAND PRIX and related marks are trademarks of Formula One Licensing BV, a Formula One group company. All rights reserved.

The Ergast Developer API is an experimental web service which provides a historical record of motor racing data for non-commercial purposes. http://ergast.com/mrd/ .

R: A Language and Environment for Statistical Computing, is produced by the R Core Team/R Foundation for Statistical Computing, http://www.R-project.org .

RStudio™ is a trademark of RStudio, Inc. http://www.rstudio.com/ .

ggplot2: elegant graphics for data analysis, Hadley Wickham, Springer New York, 2009. http://ggplot2.org/

knitr: A general-purpose package for dynamic report generation in R, Yihui Xie. http://yihui.name/knitr/

Leanpub: flexible ebook posting, and host of this book at http://leanpub.com/wranglingf1datawithr/

Errata

An update of the RSQLite package to version 1.0.0 (25.10/14) requires the following changes:

```
### COMMENT OUT THE ORIGINAL SET UP
#require(RSQLite)
#ergastdb = dbConnect(drv='SQLite', dbname='./ergastdb13.sqlite')

### REPLACE WITH:
require(DBI)
ergastdb =dbConnect(RSQLite::SQLite(), './ergastdb13.sqlite')
```

Contents

Foreword

For several years I've spent Formula One race weekends dabbling with F1 data, posting the occasional review on the F1DataJunkie website (http://f1datajunkie.blogspot.com). If the teams can produce updates to their cars on a fortnightly or even weekly basis, I thought I should be able to push my own understanding of data analysis and visualisation at least a little way over the one hour TV prequel to each qualifying session and in the run up to each race.

This book represents a review of some of those race weekend experiments. Using a range of data sources, I hope to show how we can use powerful, and freely available, data analysis and visualisation tools to pull out some of the stories hidden in the data that may not always be reported.

Along the way, I hope to inspire you to try out some of the techniques for yourself, as well as developing new ones. And in much the same way that Formula One teams pride themselves in developing technologies and techniques that can be used outside of F1, you may find that some of the tools and techniques that I'll introduce in these pages may be useful to you in your own activities away from F1 fandom. If they are, let me know, and maybe we can pull the ideas together in an *#F1datajunkie spinoffs* book!

Indeed, the desire to learn plays a significant part on my own #f1datajunkie activities. Formula One provides a context, and authentic, real world data sets, for exploring new-to-me data analysis and visualisation techniques that I may be able to apply elsewhere. The pace of change of F1 drives me to try new things out each weekend, building on what I have learned already. But at the end of the day, if any of my dabblings don't work out, or I get an analysis wrong, it doesn't matter so much: after all, this is just recreational data play, a conversation with the data where I can pose questions and get straightforward answers back, and hopefully learn something along the way.

A Note on the Data Sources

Throughout this book, I'll be drawing on a range of data sources, some openly licensed (and therefore, free for reuse), such as the Ergast motor racing results API (http://ergast.com) maintained by Chris Newell; and some not, such as results data scraped from the official

Formula One website or telemetry data grabbed from the McLaren F1 team website's live dashboard over the course of several race weekends.

Where the data is openly licensed, I will point you to places where you can access it directly. For the copyrighted data, an element of subterfuge may be required: I can tell you how to grab the data, but I can't share a copy of it with you. On occasion, I may take exception to this rule and point you to a copy of the data I have made available for the purpose of reporting, personal research and and/or criticism.

Source code - and data sets (as *.sqlite* files) - will be available from: https://github.com/psychemedia/wrang
Note that the contents of this repository may lag the contents of this book.

Introduction

Preamble

This book is a hands-on guide to wrangling and visualising data, put together to encourage you to start working with Formula One data yourself using a range of free, open source tools and, wherever possible, openly licensed data. But this book isn't just a book for F1 fans. It's a book full of recreational data puzzles and examples that explore how to twist and bend data into shapes that tell stories. And it's crammed full of techniques that aren't just applicable to motorsport data. So if you find a technique or visualisation you think you may be able to use, or improve, with your own data, wheresoever it comes from, then go for it!

Formula One is a fast paced sport - and industry. Cars are originally developed over relatively short time-periods: when the season is on the race to update and improve car performance is as fast moving and ferocious in the pits and factories as it is on the track. F1 is also, increasingly, a data driven sport. Vast amounts of telemetry data are streamed in realtime from cars to pits and back to the home factories over the course of a race weekend, and throughout the race itself. Data plays a key role in car design: computational fluid dynamics (not something we shall cover here!) complements wind tunnel time for checking the aerodynamic performance of evolving car designs. And data plays a key role in developing race strategies: race simulations are run not just on the track but also in 'mission control' centres, not just in advance of the race but during the race itself, as strategies evolve, and 'events, dear boy, events' take their toll.

In many sports, "performance stats" and historical statistics provide an easy fill for commentators looking to add a little colour or context to a report, but where do the commentary teams find the data, or the stories in the data?

The focus in these pages will be primarily on the stats: statistics about previous races, previous championships, and the performance of current or previous drivers. We'll see where those stats come from, and how to create fun facts and figures of your own to rival the apparently boundless knowledge of the professional commentators, or professional motorsport statisticians such as @virtualstatman, Sean Kelly. If you fancy the occasional flutter on an F1 race or final championship standing, you may even be able to get some of the stats to work for you…

Where the data's available, we'll also try to have a glimpse behind the scenes at some of the number crunching the teams get up to, from analysing telemetry to plotting race strategy. There are several blogs that are worth following in this respect, such as intelligentF1[1], where rocket scientist James Beck analyses laptime data each race weekend to try to model, and explain, the tyre selection, fuel saving and race strategies the teams adopt each weekend.

I'm also hoping you may try to develop your own visualisations and analyses, joining a week-on-week race to develop, refine, improve and build on the work in these pages as well as your own. And I have another hope, too - that you may find some of the ideas about how to visualise data, how to work with data, how to have *conversations* with data of some use outside of F1 fandom, maybe in your workplace, maybe in your local community, or maybe in other areas of motorsport.

What are we trying to do with the data?

As well as using Formula One data to provide a context for learning how to wrangle and visaulise data in general, it's also the case that we want to use these techniques to learn something about the world of F1. So what's the data good for?

In the first case, we can look for stories in the data that tell us about *what has happened* - the drivers who have won the most races, or taken the most pole positions; which the most successful teams were, or are, for some definition of "successful"; how many laps each driver led a particular race for. And so on.

Secondly, we can use the data to try to *predict* what will happen in the future: who is most likely to win the championship this year, or a particular race given the performance in that weekend's practice and qualifying sessions.

Thirdly, we can use the data as a knowledgeable source we can have a conversation with about F1, if we ask the questions in the right way. *Conversations with data* is how I refer to this. You're probably already in the habit of having a conversation with Google about a particular topic: you put in a keyword, Google gives you some links back. You skim of some of them, refine your query, ask again. One link looks interesting so you follow it; the web page gives you another idea, back to Google, a new search query, and so on. In the sense that the Google search engine is just a big database, you've had some sort of conversation with it. After reading this book, you'll hopefully be similarly able to have a conversation with some raw datasets!

[1] http://intelligentf1.wordpress.com/

Choosing the tools

As far as the data analysis and visualisation tools go, I wanted to choose an approach that would allow you to work on any major platform (Windows, Mac, Linux) using the same, free tools (ideally open source) irrespective of platform. Needless to say, it was essential that you should be able to create a wide range of data visualisations across a range of Formula One related datasets. At the back of my mind was the idea that a browser based UI would present the ideal solution: in the furst case, browsers are nowadays ubiquitous; secondly, separating out a browser based user interface from an underlying server means that you can run the underlying server either on your own computer, in a virtual machine on your own computer, or on a remote server elsewhere on the web.

Tied to the choice of development environment was the the choice of programming language. There were two major candidates - R and Python. What I was looking for was a programming/data analysis language that would:

- allow you to manipulate data relatively easily - ingesting it from whatever data source we might be using (a downloaded file, an online API, or a database management system);
- be supported by an integrated development environment that would let you develop your own analyses in an interactive fashion, allowing you to see graphical results alongside any code used to generate them, as well as a way of easily previewing the data you were working with.

There were two main options to the language/development environment/visualisation approach that I considered: R/RStudio/ggplot2 and python/IPyython notebook/matplotlib. Both these triumvirates are popular among those emerging communities of data scientists and data journalists. A third possibility, and one that I may explore for future versions of this book, is to run the R code from within an IPython notebook.

In the end, I opted for the R/RStudio/ggplot2 route, not least because I'd already played with a wide range of simple analyses and visualisations using that combination of tools on the f1datajunkie blog. The R milieu has also benefitted in recent months from Ramnath Vaidyanathan's pioneering work on the RCharts library that makes it easy to create a wide range of interactive browser based visualisations built on top of a variety of Javascript based data visualisation libraries, including several based on Mike Bostock's powerful d3.js library.

The RStudio development environment can run as a cross-platform standalone application, or run as a server accessed via a web browser, and presents a well designed environment within

which to explore data wrangling with R. Whilst you do not have to use RStudio to run any of the analysis or produce any of the visualisations produced herein, I would recommend it: it's a joy to use.

(There's also a possibility that once finished, I may try to produce a version of this book that follows the python/ipython notebook/matplotlib route, maybe again with a few extensions that support the use of Javascript charting libraries.;-)

The RStudio Environment

RStudio is an integrated development envirionment (IDE) for the R programming language. R is a free, open source (GPL licensed) programming language that was originally developed for statistical computing and analysis. R is supported by an active community of contributors who have developed a wide variety of packages for running different sorts of of statistical analysis. R also provides rich support for the production of high quality statistical charts and graphics and is increasingly used in the production of complex data visualisations.

The RStudio IDE is cross-platform application available in a free, open source edition as well as commercially supported versions. Whilst capable of running as a standalone desktop application, RStudio can also run as a server, making the IDE available via a web browser with R code executing on the underlying server. This makes packaging RStudio in a virtual machine, running it as a service, and accessing it through a browser on a host machine, a very tractable affair (for example, RStudio AMI shared by Louis Aslett[2] or Running RStudio via Docker in the Cloud[3]). Producing a virtual machine pre-populated with tools, scripts and datasets is very much on the roadmap for future revisions of this book.

The Data Sources

There are several sources of F1 data that I will be drawing on throughout this book, as well as some that will not necessarily be covered in the early editions at least.

Ergast Motor Racing Database - Overview

The ergast experimental Motor Racing Developer API[4] provides a historical record of Formula One results data dating back to 1950.

The data is organised into a set of 11 database tables:

[2] http://www.louisaslett.com/RStudio_AMI/

[3] http://www.magesblog.com/2014/09/running-rstudio-via-docker-in-cloud.html

[4] http://ergast.com/mrd/

- *Season List* - a list of the seasons for which data is available
- *Race Schedule* - the races that took place in each given season
- *Race Results* - the final classification for each race
- *Qualifying Results* - the results of each qualifying session from 2003 onwards
- *Standings* - driver and constructor chanpionship standings after each race
- *Driver Information* - information about each driver and their race career
- *Constructor Information* - details about the race history of each team
- *Circuit Information* - information about each circuit and its competition history
- *Finishing Status* - describes the finishing status for each competitor
- *Lap Times* - race lap times from the 2011 season onwards
- *Pit Stops* - pit stop data for each race from 2012 onwards

Chris Newell, maintainer of the ergast website, publishes the results data via both a machine readable online API and via a database dump. We will see how to work with both these sources to generate a wide range of charts as well as some simple interactive applications.

formula1.com Results Data

Although not published as open licensed data, or indeed as data in a data format, it is possible to scrape data from the F1.com website and put it into a database, such as a SQLite database.

The formula1.com website publishes current season and historical results data dating back to 1950 at http://www.formula1.com/results/[5]. From 1950 to 2002 only race results are provided. Since 2003, the data includes results from practice and qualifying sessions. From 2004, best sector times and speed trap data is also available for practice and qualifying sessions, and fastest laps and pit stop information for the race.

If you would like to run your own analyses over the formula1.com results data, I have posted details about the python screenscraper I use in one of the appendices.

FIA Event Information and Timing Data

Over the course of a race weekend, as well as live timing via the F1 website and the F1 official app, the FIA publish timing information for each session via a series of PDF documents. These documents are published on the FIA.com website over the course of the race weekend. Until 2012, the documents for each race would remain available until the next race, at which point

[5] http://www.formula1.com/results/

they would disappear from the public FIA website. From 2013, an archive site[6] has kept the documents available.

Downloading the PDF documents needs to be done one document at a time. To support the bulk downloading of documents for particular race weekend, I have described a short python program in one of the appendices that can download all the PDF documents associated with a particular race.

The documents published by the FIA for each race are as follows:

- *Stewards Biographies* - brief text based biography for each steward
- *Event Information* - biref introduction to the race, quick facts, summary of standings to date, circuit map
- *Circuit Information* - graphics of FIA circuit map, F1 circuit map [extracting images from circuit maps???]
- *Timing Information* - a range of timing information for each session of the race weekend
- *FIA Communications* - for example, notes to teams
- *Technical Reports* - for example, updates from the FIA Formula 1 Technical Delegate
- *Press Conference Transcripts*- transcripts from each of the daily press conferences (Thursday, Friday, Saturday, Sunday)
- *National Press Office* - Media Kit from the local press office. [For example, can we recreate the team data panels? Extract car images?]
- *Stewards Decisions* - notices about Steward's decisions for each day, with information broke down into seperate list items (No/Driver, Competitor (i.e. the team), Time, Session, Fact, Offence, Decision, Reason)
- *Championship Standings* - drivers and constructors championship standings once the race result is confirmed

The FIA Timing Data in Detail

The following list identifies the timing data is available for each of the sessions:

- **Practice**
  - Classification
  - Lap Times

[6] http://www.fia.com/championships/archives/formula-1-world-championship/2013

- **Qualifying**
  - Speed Trap
  - Best Sector Times
  - Maximum Speeds
  - Lap Times
  - Preliminary Classification
  - Provisional Classification
  - Official Classification
- **Race**
  - Starting Grid - Provisional
  - Starting Grid - Official
  - Pit Stop Summary
  - Maximum Speeds
  - Speed Trap
  - Lap Analysis
  - Best Sector Times
  - Lap Chart
  - History Chart
  - Fastest Laps
  - Preliminary Classification
  - Provisional Classification
  - Official Classification

Some of this data is also published as HTML data tables on the previously mentioned formula1.com *results* data area.

Using the FIA Event Information

Getting the data from the PDF documents into a usable form is a laborious procedure that requires scraping the data from the corresponding timing sheet and then either adding it to a database or making it otherwise available in a format that allows us to read it into an R data frame.

Note that we can recreate data sets corresponding to some of the sheets from other data sources, such as the Ergast API. However, other data sets must be grabbed by scraping the FIA sheets directly.

Descriptions of how to scrape from the FIA PDFs, or analyses of data only available from that source, will not be covered in the first few editions of this book.

Viva F1 - Race Penalties

For the 2012 and 2013 seaesons, the Viva F1[7] site publish a summary list of race penalties[8] awarded during the course of a race weekend, and then use this information to generate a visualisation of the penalties. Whilst not broken down as *data*, it is possible to make use of the common way in which the penalties are described to parse out certain "data elements" from the penalty descriptions.

Race Telemetry

Between 2010 and 2013, the McLaren race team published a live driver dashboard that relayed some of the telemetry data from their cars to an interactive, web based dashboard. (Mercedes also had adasboard that streamed live telemetry.) The data was pulled into the web page by polling a Mclaren data source once per second. At the time, it was possible to set up a small data logging script that would similarly call this source once a second and produce a data log containing telemetry data collected over a whole session. This data could then be used to analyse performance over the course of a session, or provide a statistical view over the data based on samples collected at similar locations around the track across one or more sessions.

The current F1 app includes live information about track position and tyre selection, but the data is not made openly available. The commercial licensing decisions surrounding this particular set of F1 data therefore makes fan driven innovation around it very difficult.

Getting the Data into RStudio

The Ergast API publishes data in two data formats - JSON (Javascript Object Notation) and XML. Calls are made to the API via a web URL, and the data is returned in the requested format. To call the API therefore requires a live web connection. To support this book, I have started to develop an R library, currently available as *ergastR-core.R* from the *wranglingf1datawithr* repository[9]. The routines in this library can be used to request data from the Ergast API in JSON form, and then cast it into an R data frame.

Historical data for all *complete* seasons to date is available as a MySQL database export file that is downloadable from the ergats website. Whilst R can connect to a MySQL database, using this data does require the that the data is uploaded to a MySQL database, and that

[7] http://www.vivaf1.com

[8] http://www.vivaf1.com/penalties.php

[9] https://github.com/psychemedia/wranglingf1datawithr/blob/master/src/ergastR-core.R

the database is configured with whatever permissions are required to allow R to access the data. To simplify the database route, I have converted to the MySQL export file to a SQLite database file. This simple database solution allows R to connect to the SQLite database directly. The appendix *Converting the ergast Database to SQLite* describes how to generate a sqlite3 version of the database from the original MySQL data export file.

We will see how to use both the ergast API and the explorted ergast database as the basis for F1 stats analyses and data visualisations.

> Sample datasets (in sqlite form) can be downloaded from github/psychemedia/wranglingf1datawithr[a] as: * ergast database - *ergastdb13.sqlite* * F1 results scrape - *scraperwiki.sqlite*
>
> [a] https://github.com/psychemedia/wranglingf1datawithr/tree/master/src

Example F1 Stats Sites

Several websites producing comprehensive stats reports around F1 that can provide useful inspiration for developing our own analyses and visualisations, or for trying to replicate those produced by other people.

I have already mentioned the intelligentF1[10] website, which analyses race history charts from actual races as well as second practice race simulations in an attempt to identify possible race strategies, particularly insofar as they relate to tyre wear and, form the 2014 season, fuel saving.

On the season and race stats side, F1fanatic[11] produces a wide range of browser based interactive season and race summary charts, some of which we'll have a go at replicating throughout this book.

Although not an F1 stats site *per se*, I always enjoy visiting sidepodcast.net[12], not least for its lively community and live race coverage.

[10] http://intelligentf1.wordpress.com/

[11] http://www.f1fanatic.co.uk/statistics/2014-f1-statistics/

[12] http://sidepodcast.net

How to Use This Book

This book is filled with bits and pieces of R code that have been used to directly generate all the analyses and visualisations shown in these pages. You should be able to copy the code and run it in your own version of RStudio assuming you have downloaded and installed the appropriate R packages, and that the necessary data files are available (whether by downloading them or accessing them via a live internet/web connection).

Explanations of how the code works is presented in both the text and as comments in the inline code. You are encouraged to read the program code to get a feel for how it works. Comments are used to introduce and explain various code elements where appropriate, so by not reading the code fragments you may miss out on learning some handy tips and tricks that are not introduced explictly in the main text.

Several of the chapter include an *Exercises* sections that includes a few recreational data puzzles and exercises for you to practice again some of the things covered in the chapter. Some of these exercise sections also include *TO DO* items. These reflect the work-in-progress nature of this live book and represent things I haven't yet got round to doing and that may be useful for future rolling editions of the text. *TO DO* items may go beyond simply rehearsing or stretching the ideas covered in the respective chapter and may require some new learning to be done, problems to be solved, or things to be figured out!

The Rest of This Book...

For the next few months, this book will be a living book, which means that it is subject to change. For the moment, the chapters wil be broadly grouped as follows:

- *getting started* sections - introducing the technical tools we'll be using, R and Rstudio, and the datasets we'll be playing with, in particular the ergast data.
- *race weekend analysis* - a look at data from over a race weekend, how to start analysing it and how we can visualise it;
- *season analysis* sections - looking at season reviews and toos and techniques for analysing and visualsising results across a championship and comparing performances year on year;
- *map views* - a look at how what geo and GPS data is available, and how we might be able to make use of it
- *the gambler's way* - a quick review of how to grab betting data, and some of the things we may be able to draw from it;

- *interactive web charts* using a variety of d3.js inspired HTML5 charting libraries via the rCharts library;
- *application development* - how to develop simple interactive applications with the shiny R library.

If you spot any problems with the code included in this book, please post an issue to Wrangling F1 Data with R - github[13].

If you would like to buy this book (including future updates to it), or make a donation to support its development, please visit Wrangling F1 Data with R - leanpub[14].

[13] https://github.com/psychemedia/wranglingf1datawithr/issues
[14] https://leanpub.com/wranglingf1datawithr

An Introduction to RStudio and R dataframes

In this chapter you will get to meet the RStudio interactive development environment and start to explore the R language. Whilst this book is not intended to take on the role of teaching you how to become a professional R programmer, you will hopefully learn enough to be able to read and write simple bits of R code yourself, as well as learning how to use some powerful R libraries such as the `ggplot2` graphics library to produce your own data visualisations.

In the majority of cases, wherever a data table, data visulisation or data analysis appears in this book, it will be preceded by the code required to generate it. This in part is a result of the workflow I have chosen to generate the book. Each chapter is written as a an "R markdown" (*Rmd*) formatted document that combines free text styled using the simple *markdown* language and blocks of R code. The *Rmd* document can then be processed so that each block of R code is executed and the results that it generates, such as a table or chart, included in the manuscript.

If you want to replicate any of the charts or analyses included in this book, you should be able to just copy the appropriate preceding bits of code and run them in your own version of RStudio.

Getting Started with RStudio

RStudio is a free, open source, cross-platform application that runs on Apple OS/X, Microsoft Windows and Linux operating systems. It can also be run as a service on a virtual or cloud hosted machine and accessed remotely via a web browser.

To get RStudio up and running, you need to:

- **download and install R** from the R Project for Statistical Computing[15]. The download area[16] requires you to select a "CRAN mirror". *CRAN* is the *Comprehensive R Archive Network* which hosts the R source code and a wide selection of community

[15] http://www.r-project.org/

[16] http://cran.r-project.org/mirrors.html

developed packages that cover a huge range of statistical analysis and data visualisation techniques. CRAN services are located all round the world, so choose a location that is convenient for you.
- **download and install the desktop edition of RStudio** from RStudio.org[17].

Configuring the server edition of RStudio so that you can run it as a service and access it via a web browser is a rather more involved process and will not be covered in this book. If you would like to run the server, the easiest way is probably to define a virtual machine (for example, as described in Running RStudio via Docker in the Cloud[18]) or make use of an Amazon machine image, such as the RStudio AMI shared by Louis Aslett[19].

Once you have installed RStudio and got it running, you should be presented with something like this:

Getting Started with R

As a programming language, R uses a syntax that has many resemblances to other programming languages (as well as a few quirks!). In this section, we will review some of the features of the R language that are particularly useful when it comes to working with data. Where appropriate, we will review particular techniques in more detail in later chapters where the technique plays a key role in wrangling the data so that we can visualise or analyse it. For now, the following whirlwind tour should give you an idea of some of the sorts of the thing that are possible, along with a glimpse of how we might achieve them.

Note that there is no trickery involved - the output is generated by running only and exactly the code shown in this document. Indeed, the output from running each code block is automatically inserted into the original manuscript of this book following execution of the code block.

One of the key ideas of many programming languages is the idea of a *function*. A function can be used to run one or more operations over a data set that is passed to it. We *call* a function by writing its name followed by a pair of brackets - for example *getwd()*. This function *returns* the name of, and path to, the directory we're currently working in.

[17] https://www.rstudio.com/ide/download/
[18] http://www.magesblog.com/2014/09/running-rstudio-via-docker-in-cloud.html
[19] http://www.louisaslett.com/RStudio_AMI/

```
getwd()
```

```
## [1] "/Users/ajh59/Dropbox/wranglingf1datawithr/src"
```

To get some help information about a function, in the Rstudio console enter a ? immediately followed by the function name. For example, we can look up what the complemtary `setwd()` function does by entering `?setwd`:

That is, we can use `setwd()` to set the current working directory.

When working with data, we often want to be able to create a list of values. The function `c()` accepts a list of comma separated values that it will then use to construct a vector or a list:

```
c("Hamilton","Rosberg","Vettel","Ricciardo","Alonso","Raikkonen","Button","Magnussen")
```

```
## [1] "Hamilton"  "Rosberg"   "Vettel"    "Ricciardo" "Alonso"    "Raikkonen"
## [7] "Button"    "Magnussen"
```

The margin numbers in the output identify the index number, or list count number, of the first item in each row of the display. In this case, I can see (wor quickly work out) that there are 8 elements in the list.

As most of the datasets we're going to be working with come in the form of two dimensional data tables (the sort of layout you may be familiar with from looking at a simple spreadsheet), then the *data frame* is the data structure you're likely to spend most of your time working with.

In many respects, dataframes can be thought of in much the same way as a set of tabular data contained in a worksheet in a spreadsheet application. The data is arranged in rows and columns, each of which can be individually identified. In addition, each cell in the dataframe/worksheet can be uniquely addressed by giving its row and column "co-ordinates".

To explore some of the features of a dataframe, I have produced a sample dataset that contains data from the first three races if the 2014 season, identifying who was on the podium for each race along along with their finishing position.

The data is currently stored in a CSV - comma separated variable/value - file, a simple text based file format for sharing tabular data that you can find here: winners2014start-csv.csv[20]. You can view the raw data file by clicking on the <> icon in the file title bar.

If you download the raw data to your computer, and take note of the location where you actually saved the file too, you can load the data into RStudio:

You can also use the `read.csv()` function to load the data in yourself:

```
winners=read.csv('~/Dropbox/wranglingf1datawithr/src/winners2014start.csv')
winners
```

```
##   carNum pos        driverId constructorId grid fastlaptime fastlaprank
## 1      6   1         rosberg      mercedes    3      92.478           1
## 2     20   2 kevin_magnussen       mclaren    4      93.066           6
## 3     22   3          button       mclaren   10      92.917           5
## 4     44   1        hamilton      mercedes    1     103.066           1
## 5      6   2         rosberg      mercedes    3     103.960           2
## 6      1   3          vettel      red_bull    2     104.289           4
## 7     44   1        hamilton      mercedes    2      97.108           2
## 8      6   2         rosberg      mercedes    1      97.020           1
## 9     11   3           perez   force_india    4      99.320           7
##       race
## 1 Australia
## 2 Australia
## 3 Australia
## 4  Malaysia
## 5  Malaysia
## 6  Malaysia
## 7   Bahrain
## 8   Bahrain
## 9   Bahrain
```

Note that in R the operator used to assign a value or an object (such as the output of the `read.csv()` function), to a *variable*, such as `winners` is traditionally written as <-, as in:

[20] https://gist.github.com/psychemedia/11187809#file-winners2014start-csv

```
winners <- read.csv('~/Dropbox/wranglingf1datawithr/src/winners2014start.csv')
winners
```

```
##   carNum pos        driverId constructorId grid fastlaptime fastlaprank
## 1      6   1         rosberg      mercedes    3      92.478           1
## 2     20   2 kevin_magnussen       mclaren    4      93.066           6
## 3     22   3          button       mclaren   10      92.917           5
## 4     44   1        hamilton      mercedes    1     103.066           1
## 5      6   2         rosberg      mercedes    3     103.960           2
## 6      1   3          vettel      red_bull    2     104.289           4
## 7     44   1        hamilton      mercedes    2      97.108           2
## 8      6   2         rosberg      mercedes    1      97.020           1
## 9     11   3           perez   force_india    4      99.320           7
##       race
## 1 Australia
## 2 Australia
## 3 Australia
## 4  Malaysia
## 5  Malaysia
## 6  Malaysia
## 7   Bahrain
## 8   Bahrain
## 9   Bahrain
```

However, I prefer to use the equally valid assignment operator = which is conventionally used in many other programming languages.

If we just enter the name of a variable, such as `winners`, the R interpreter will print out the contents of the variable.

If we want to reference just the values in a single column as a vector of values, we can reference the column as follows:

- `DATAFRAME['COLNAME']`: for example, *winners['driverId']*

```
winners['driverId']
```

```
##          driverId
## 1         rosberg
## 2 kevin_magnussen
## 3          button
## 4        hamilton
## 5         rosberg
## 6          vettel
## 7        hamilton
## 8         rosberg
## 9           perez
```

We can also get a list of the values contained within a column in one of two ways:

- `DATAFRAME[['COLNAME']]`: for example, *winner[['driverId']]*
- `DATAFRAME$COLNAME`: for example, *winners$driverId*

```
winners[['driverId']]
```

```
## [1] rosberg         kevin_magnussen button          hamilton
## [5] rosberg         vettel          hamilton        rosberg
## [9] perez
## Levels: button hamilton kevin_magnussen perez rosberg vettel
```

If we wish to inspect the contents of a dataframe in particular in a more convenient way, we can using the viewing area of RStudio. The easiest way to do this is to identify which data object you wish to inspect via the *Environment* tab, then click on it to launch it into a tab in the viewing area:

Alternatively, via the RStudio console, you can execute the `View()` function applied to the required dataframe object directly (for example, `View(winners)`).

We can get different sorts of summary about a data from by using the `str()` and `summary()` commands.

To review the structure of a dataframe, use `str()`:

```
str(winners)
```

```
## 'data.frame':        9 obs. of  8 variables:
##  $ carNum       : int  6 20 22 44 6 1 44 6 11
##  $ pos          : int  1 2 3 1 2 3 1 2 3
##  $ driverId     : Factor w/ 6 levels "button","hamilton",..: 5 3 1 2 5 6 2 5 4
##  $ constructorId: Factor w/ 4 levels "force_india",..: 3 2 2 3 3 4 3 3 1
##  $ grid         : int  3 4 10 1 3 2 2 1 4
##  $ fastlaptime  : num  92.5 93.1 92.9 103.1 104 ...
##  $ fastlaprank  : int  1 6 5 1 2 4 2 1 7
##  $ race         : Factor w/ 3 levels "Australia","Bahrain",..: 1 1 1 3 3 3 2 2 2
```

In this case, we see that several columns have been *typed* as having **int**eger value contents, (for example, the *carNum* and *pos* columns), *fastlaptime* has been identified as a **num**eric, and the *driverId*, *constructorId* and *race* columns as *Factors* (that is, categorical variables). Since the adoption of personalised, permanent driver numbers for cars at the start of the 2014 season, the *carNum* is less an integer than a categorical variable. We can covert it to a factor as follows:

```
winners$carNum=factor(winners$carNum)
```

If you rerun the `str()` command you will see that the *carNum* is now a factor with 6 levels:

```
str(winners$carNum)
```

```
##  Factor w/ 6 levels "1","6","11","20",..: 2 4 5 6 2 1 6 2 3
```

If appropriate, we can also cast columns to integer values (with *as.integer()*), numerical values (floats/reals, *as.numeric()*) and to characters/strings (*as.character()*). Note that when casting something detected as a factor to an integer or numeric, we first need to cast it to a character string. So for example, to cast the *winners$carNum* from a factor back to an integer, we would use the construction *as.integer(as.character(winners$carNum))*.

We can inspect the different values take by a categorical variable using the `levels()` function as applied to a column associated with a dataframe (*dataframe$column*):

```
levels(winners$race)
```

```
## [1] "Australia" "Bahrain"   "Malaysia"
```

To get summary statistics back about the contents of a dataframe, which may or may not be *meaningful* summary statistics, use `summary()`:

```
summary(winners)
```

```
##  carNum      pos              driverId     constructorId      grid
##  1 :1   Min.   :1   button         :1   force_india:1     Min.   : 1.000
##  6 :3   1st Qu.:1   hamilton       :2   mclaren    :2     1st Qu.: 2.000
##  11:1   Median :2   kevin_magnussen:1   mercedes   :5     Median : 3.000
##  20:1   Mean   :2   perez          :1   red_bull   :1     Mean   : 3.333
##  22:1   3rd Qu.:3   rosberg        :3                     3rd Qu.: 4.000
##  44:2   Max.   :3   vettel         :1                     Max.   :10.000
##   fastlaptime      fastlaprank          race
##  Min.   : 92.48   Min.   :1.000   Australia:3
##  1st Qu.: 93.07   1st Qu.:1.000   Bahrain  :3
##  Median : 97.11   Median :2.000   Malaysia :3
##  Mean   : 98.14   Mean   :3.222
##  3rd Qu.:103.07   3rd Qu.:5.000
##  Max.   :104.29   Max.   :7.000
```

The `read.csv()` function can also load data in from a URL, rather than a path that specifies local file on your own computer. However, the data file is being served from a secure *https* link, rather than a simpler *http* web address, so we need to do a workaround by loading the data in using a function that is contained in the *RCurl* R package. If your R installation does not have the RCurl package installed, you will need to install it yourself. In RStudio, you can install a package from the *Packages* tab:

We can also define a simple function that will check to see whether a package is installed before we try to load it in; if it's missing, the package installer will be run first:

```
#Lines preceded by a # are comments that are not executed as R code

# This simple recipe takes in a list of packages that need to be loaded,
# checks to see whether they are installed, installs any that are missing,
# including their dependencies, and then loads them all in

#The list of packages to be loaded
list.of.packages <- c("RCurl","ggplot2")

#You should be able to simply reuse the following lines of code as is
new.packages <- list.of.packages[!(list.of.packages %in% installed.packages()[,"Packa\
ge"])]
if(length(new.packages)) install.packages(new.packages)
lapply(list.of.packages,function(x){library(x,character.only=TRUE)})


## [[1]]
##  [1] "RCurl"     "bitops"    "knitr"     "stats"     "graphics"
##  [6] "grDevices" "utils"     "datasets"  "methods"   "base"
##
## [[2]]
##  [1] "ggplot2"   "RCurl"     "bitops"    "knitr"     "stats"
##  [6] "graphics"  "grDevices" "utils"     "datasets"  "methods"
## [11] "base"
```

If you grab the link for the *raw* file (the <> icon in the file title bar), we can now use the following recipe to load the data in directly from the URL.

```
urlstub='https://gist.githubusercontent.com/psychemedia'

#The actual path may be different - check the actual link from
## https://gist.github.com/psychemedia/11187809#file-winners2014start-csv
urlpath='11187809/raw/38295d24d5e2e35e0b92cb2cf4082a500a691ffd/winners2014start.csv'

#The paste() command concatenates a comma separated list,
#joining values with a separator specified by the
#sep='' parameter; use sep='' for a seamless join
url=paste(urlstub,urlpath,sep='/')

library(RCurl)
data = getURL(url)
winners = read.csv(text = data)
winners
```

```
##   carNum pos        driverId constructorId grid fastlaptime fastlaprank
## 1      6   1         rosberg      mercedes    3      92.478           1
## 2     20   2 kevin_magnussen       mclaren    4      93.066           6
## 3     22   3          button       mclaren   10      92.917           5
## 4     44   1        hamilton      mercedes    1     103.066           1
## 5      6   2         rosberg      mercedes    3     103.960           2
## 6      1   3          vettel      red_bull    2     104.289           4
## 7     44   1        hamilton      mercedes    2      97.108           2
## 8      6   2         rosberg      mercedes    1      97.020           1
## 9     11   3           perez   force_india    4      99.320           7
##       race
## 1 Australia
## 2 Australia
## 3 Australia
## 4  Malaysia
## 5  Malaysia
## 6  Malaysia
## 7   Bahrain
## 8   Bahrain
## 9   Bahrain
```

If we want to pass something into the function, we put it inside the brackets. For example, the *head()* function can take in a dataframe and then display the first few lines of the dataframe (for example, `head(winners)`). The *head()* function can also accept further arguments that are placed within the brackets, and separated by commas. For example, by default, *head()* previews the first 10 rows of a data frame. We can change this number by means of the *n* parameter. To display just the first three (3) lines of the `winners` dataframe, we would write *head(winners, n=3)*.

```
head(winners, n=3)
```

```
##   carNum pos        driverId constructorId grid fastlaptime fastlaprank
## 1      6   1         rosberg      mercedes    3      92.478           1
## 2     20   2 kevin_magnussen       mclaren    4      93.066           6
## 3     22   3          button       mclaren   10      92.917           5
##       race
## 1 Australia
## 2 Australia
## 3 Australia
```

We can also look at the final few lines of a data frame using a complementary function, *tail()*; once again, the *n* parameter can be used to specify how many lines to display.

For the purposes of this book, I am using a further function - `kable()` - that tidies up the display of data tables. *(`kable()` can be found in the *knitr* library that is preinstalled in RStudio when working with R markdown.Rmd files.)** If you are trying out the code yourself, omit the `kable()` function from it unless you particularly want to the markdown out format.

```
kable( head(winners) ,row.names=F,format="markdown")
```

carNum	pos	driverId	constructorId	grid	fastlaptime	fastlaprank	race
6	1	rosberg	mercedes	3	92.478	1	Australia
20	2	kevin_-magnussen	mclaren	4	93.066	6	Australia
22	3	button	mclaren	10	92.917	5	Australia
44	1	hamilton	mercedes	1	103.066	1	Malaysia
6	2	rosberg	mercedes	3	103.960	2	Malaysia
1	3	vettel	red_bull	2	104.289	4	Malaysia

If you have done any programming before, you will probably be familiar with the notion of variable *types*. For example, a variable may be of an *integer* type (that is, it only and must take on whole number values), a *character* type (often referred to as a *string*) or a ral or floating point number. In the same waym the columns within a data frame can take on a particular type.

We can inspect the structure of the table using the command *str()* which gives us some summary information about the type of each column in a dataframe:

```
str(winners)
```

```
## 'data.frame':        9 obs. of  8 variables:
##  $ carNum       : int  6 20 22 44 6 1 44 6 11
##  $ pos          : int  1 2 3 1 2 3 1 2 3
##  $ driverId     : Factor w/ 6 levels "button","hamilton",..: 5 3 1 2 5 6 2 5 4
##  $ constructorId: Factor w/ 4 levels "force_india",..: 3 2 2 3 3 4 3 3 1
##  $ grid         : int  3 4 10 1 3 2 2 1 4
##  $ fastlaptime  : num  92.5 93.1 92.9 103.1 104 ...
##  $ fastlaprank  : int  1 6 5 1 2 4 2 1 7
##  $ race         : Factor w/ 3 levels "Australia","Bahrain",..: 1 1 1 3 3 3 2 2 2
```

You will notice that in this dataframe, the columns are made up of *integer* types and *factors*. Factors can be thought of as *categorical variables*, where the values come from an enumerated list of legitimate possible values.

Sometimes, it can be more convenient to preview a dataframe by just looking at the column names. The *colnames()* function is a handy way of doing that.

```
colnames(winners)
```

```
## [1] "carNum"        "pos"           "driverId"      "constructorId"
## [5] "grid"          "fastlaptime"   "fastlaprank"   "race"
```

Vectorised Computation

As in a spreadsheet, R dataframes also support *vectorised computation*, in that a particular operation can be applied to each cell in a particular column by applying that function *as if* we were applying it to just the column.

Adding New Columns

As an example of vectorised computation, let's see how to add new columns to a dataframe.

In the first case, consider adding a column that contains the same constant value in each cell. Using a notion known as *broadcast*, we can set the contents of a new column to a single value (a vector of the required length is generated using the value in each location); or we can create vectors of the correct length to add to the dataframe by other means:

```
tmp.df=data.frame(origcol=c('a','b','c'))
tmp.df['newcol']=1
#The nrow() function gives the length (number of rows) of a dataframe
#The seq() function generates a sequence of values
#of a specified length over a required range
tmp.df$newcol2 = seq(nrow(tmp.df))

tmp.df
```

```
##   origcol newcol newcol2
## 1       a      1       1
## 2       b      1       2
## 3       c      1       3
```

Alternatively, we can create a new column in the dataframe from the values contained in one or more of the other columns. Let's create a new column, *posdelta*, that contains an integer value that reports on the the number of positions gained, or lost, by each driver going from their starting position on the grid to their final classified position (I'll create a copy of the `winners` dataframe to work with...):

```
winners2=winners

winners2['posdelta']=winners['grid']-winners['pos']
#The following is also valid
winners2['posdelta']=winners['grid']-winners$pos

kable(head( winners2, n=3))
```

carNum	pos	driverId	constructorId	grid	fastlaptime	fastlaprank	race	posdelta
6	1	rosberg	mercedes	3	92.478	1	Australia	2
20	2	kevin_magnussen	mclaren	4	93.066	6	Australia	2
22	3	button	mclaren	10	92.917	5	Australia	7

The `transform` function in the `plyr` package provides an alternative way of achieving the same thing:

```
#Load in the plyr package
library(plyr)

winners3 = transform(winners, posdelta = grid - pos )
kable(head( winners3, n=3))
```

carNum	pos	driverId	constructorId	grid	fastlaptime	fastlaprank	race	posdelta
6	1	rosberg	mercedes	3	92.478	1	Australia	2
20	2	kevin_-mag-nussen	mclaren	4	93.066	6	Australia	2
22	3	button	mclaren	10	92.917	5	Australia	7

The `transform` function actually allows us to specify multiple columns in the same call:

```
winners4 = transform(winners, posdelta = grid - pos, poslapdelta = fastlaprank - pos )
kable(head( winners4, n=3))
```

carNum	pos	driverId	constructorId	grid	fastlaptime	fastlaprank	race	posdelta	poslapdelta
6	1	rosberg	mercedes	3	92.478	1	Australia	2	0
20	2	kevin_-mag-nussen	mclaren	4	93.066	6	Australia	2	4
22	3	button	mclaren	10	92.917	5	Australia	7	2

If you want to create a column that is used to define a further column, use the `mutate()` function rather than `transform`.

Another useful plyr function is `count()` that will count the number of occurrences of unique values of within a dataframe column or set of columns.

Filtering data in a dataframe

We can filter a dataframe to show just a subset of the data it contains in a variety of ways:

- we can limit the number of rows we want to display
- we can limit the number of columns we want to display
- we can limit the number of rows and the number of columns we want to display

Let's see how to do each of these in turn using two different, but equivalent, techniqes: a "cryptic"" way, and a more verbose way.

We'll start with the verbose way, using the `subset()` function.

```
kable( subset(winners, subset=(pos==2 | pos==3)) )
```

	carNum	pos	driverId	constructorId	grid	fastlaptime	fastlaprank	race
2	20	2	kevin_-magnussen	mclaren	4	93.066	6	Australia
3	22	3	button	mclaren	10	92.917	5	Australia
5	6	2	rosberg	mercedes	3	103.960	2	Malaysia
6	1	3	vettel	red_bull	2	104.289	4	Malaysia
8	6	2	rosberg	mercedes	1	97.020	1	Bahrain
9	11	3	perez	force_india	4	99.320	7	Bahrain

The `subset()` function accepts a dataframe, with a *subset* parameter that applies a rule that identifies which rows to include in the returned subset. In this case, I select rows where the *pos* column value equals (==) the number 2 *or* the number 3; (the | symbol is a logical (Boolean) operator that represents the operation OR). We can test for inequlity using the != operator. We can also test for Boolean *and*ed conditions using the operator *&*. Numerical operators can also be applied (for example, *subset=(pos>=2)* would give the same result as shown in the example above).

Cryptically, we can achieve the same thing by passing in a "truth vector" that says whether or not a row is "TRUE" and should be included in the output dataframe:

```
winners[winners['pos']!=1,]
```

```
##   carNum pos        driverId constructorId grid fastlaptime fastlaprank
## 2     20   2 kevin_magnussen       mclaren    4      93.066           6
## 3     22   3          button       mclaren   10      92.917           5
## 5      6   2         rosberg      mercedes    3     103.960           2
## 6      1   3          vettel      red_bull    2     104.289           4
## 8      6   2         rosberg      mercedes    1      97.020           1
## 9     11   3           perez   force_india    4      99.320           7
##       race
## 2 Australia
## 3 Australia
## 5  Malaysia
## 6  Malaysia
## 8   Bahrain
## 9   Bahrain
```

Let's unpick that a little. Firstly, we define the "truthiness" of a row condition:

```
winners['pos']!=1
```

```
##         pos
##  [1,] FALSE
##  [2,]  TRUE
##  [3,]  TRUE
##  [4,] FALSE
##  [5,]  TRUE
##  [6,]  TRUE
##  [7,] FALSE
##  [8,]  TRUE
##  [9,]  TRUE
```

Passing this True/False vector into the column slot of the dataframe selects which rows to include in the result (we return the TRUE rows and drop the FALSE ones from the result): *dataframe[rowSelectionVector, columnSelectorList]*. Passing a blank *columnSelectorList* in means return *all* the rows.

To select just a subset of the output columns, pass in a list of the names of the columns we want to select:

```
winners[,c('driverId','race','pos')]
```

```
##          driverId      race pos
## 1         rosberg Australia   1
## 2 kevin_magnussen Australia   2
## 3          button Australia   3
## 4        hamilton  Malaysia   1
## 5         rosberg  Malaysia   2
## 6          vettel  Malaysia   3
## 7        hamilton   Bahrain   1
## 8         rosberg   Bahrain   2
## 9           perez   Bahrain   3
```

Using the subset function, we can identify the columns we want to select using the *select=* parameter.

```
kable( subset(winners,select=c(driverId,race,pos)) )
```

driverId	race	pos
rosberg	Australia	1
kevin_magnussen	Australia	2
button	Australia	3
hamilton	Malaysia	1
rosberg	Malaysia	2
vettel	Malaysia	3
hamilton	Bahrain	1
rosberg	Bahrain	2
perez	Bahrain	3

We can also select the columns that we *do not* want to include in the output:

```
kable( subset(winners,select=-c(driverId,race,pos)) )
```

carNum	constructorId	grid	fastlaptime	fastlaprank
6	mercedes	3	92.478	1
20	mclaren	4	93.066	6
22	mclaren	10	92.917	5
44	mercedes	1	103.066	1
6	mercedes	3	103.960	2
1	red_bull	2	104.289	4
44	mercedes	2	97.108	2
6	mercedes	1	97.020	1
11	force_india	4	99.320	7

To both *select* a set of columns and *subset* the rows, we can pass in both arguments:

```
kable( subset(winners,subset=(pos==1),select=c(driverId,race)) )
```

	driverId	race
1	rosberg	Australia
4	hamilton	Malaysia
7	hamilton	Bahrain

Or more cryptically:

```
kable( winners[winners['pos']==1,c('driverId','race')] )
```

	driverId	race
1	rosberg	Australia
4	hamilton	Malaysia
7	hamilton	Bahrain

By carefully choosing the criteria that we use to filter dataframes by row and/or column we can start to have a conversation with a dataframe based on the values contained within it.

Sorting dataframes

We can also use a variety of sorting operations to sort the data in a dataframe. The `arrange()` function from the `plyr` package provides what is perhaps the easiest way of sorting the rows in a dataframe.

```
#As we have previously loaded the plyr package we should not need to load it again
#library(plyr)

#The arrange function accepts a dataframe and then
#a list of columns to sort on.
arrange(winners[,c('carNum','driverId','race')],carNum)
```

```
##   carNum        driverId      race
## 1      1          vettel  Malaysia
## 2      6         rosberg Australia
## 3      6         rosberg  Malaysia
## 4      6         rosberg   Bahrain
## 5     11           perez   Bahrain
## 6     20 kevin_magnussen Australia
## 7     22          button Australia
## 8     44        hamilton  Malaysia
## 9     44        hamilton   Bahrain
```

We can sort on multiple columns, in either ascending (*asc*) or descending (*desc*) directions:

```
arrange(winners[,c('carNum','driverId','race')],desc(carNum),race)
```

```
##   carNum        driverId      race
## 1     44        hamilton   Bahrain
## 2     44        hamilton  Malaysia
## 3     22          button Australia
## 4     20 kevin_magnussen Australia
## 5     11           perez   Bahrain
## 6      6         rosberg Australia
## 7      6         rosberg   Bahrain
## 8      6         rosberg  Malaysia
## 9      1          vettel  Malaysia
```

We can also use the - (minus) operator to specify the reverse, or descending, order direction.

Merging Dataframes

If we need to merge dataframes, we can do so about a common column. For example, if we split the winners dataframe into two separate, smaller dataframes, we can then merge them back together with the `merge()` function.

```
winners.sub1=winners[,c('driverId','pos')]
winners.sub2=winners[,c('driverId','grid')]
winners.sub.merge=merge(winners.sub1,winners.sub2,by='driverId')
kable(head(winners.sub.merge, n=5))
```

driverId	pos	grid
button	3	10
hamilton	1	2
hamilton	1	1
hamilton	1	2
hamilton	1	1

The merge will work even if the columns are ordered differently, and can cope with unmatched entries. If the "merge column" has different names in the two dataframes to be merged, specify the names explicitly using *by.x=* for the name of the merge column in the first dataframe, and *by.y=* for the name of the merge column in the second.

Another form of sorting is to sort the *levels* associated with a factor. The `reorder()` function allows us to reorder the levels of one categorical variable/factor based on the values of a second variable.

Reshaping Data

Sometimes we may just have a single, long list of data values in an ordered fashion that we want to cast into a tabular format.

In such cases, the following rather handy function will take in the single ordered list of values, along with a list of desired column names, and shape a dataframe for us.

```
colify=function(datalist,cols){
  df=data.frame(matrix(datalist,
                       nrow=length(datalist)/length(cols),
                       byrow=T))
  names(df)=cols
  df
}

#Example:
#An ordered list of drivers by name, code and manufacturer
ll=c('hamilton','HAM','Mercedes','rosberg','ROS','Mercedes')
```

```
#Now let's make a table, specifying the column names we want to apply
colify(ll,c('Name','TLID','Manufacturer'))
```

```
##       Name TLID Manufacturer
## 1 hamilton  HAM     Mercedes
## 2  rosberg  ROS     Mercedes
```

Several libraries have been developed to help with the reshaping of R dataframes. If you've never really worked with datasets before, it may surprise you to think that a single data set can take on many different shapes. But it can...

For example, consider this dataframe, which contains the ergast driverId and positions of the podium finishers:

```
winners.lite=winners[,c('driverId','race','pos')]
head(winners.lite,n=3)
```

```
##          driverId      race pos
## 1         rosberg Australia   1
## 2 kevin_magnussen Australia   2
## 3          button Australia   3
```

This data is in a so-alled *long* form; but there is another equally valid shape that this data can take, often referred to as a *wide* format, in which we have separate columns corresponding to each race, with the values in those columns representing the race winners. To reshape the data, we will use a function from the appropriately named `reshape2` library, which we will need to load in first.

```
# You may need to download and install the reshape2 package before
# trying to load it in for the first time.

#Once the library is installed, we can load it in
library(reshape2)

wide.df = dcast(winners.lite, driverId ~ race)


## Using pos as value column: use value.var to override.


head( wide.df, n=3 )


##          driverId Australia Bahrain Malaysia
## 1          button         3      NA       NA
## 2        hamilton        NA       1        1
## 3 kevin_magnussen         2      NA       NA
```

In reshaping the data, the `dcast` function identfies the unique values in the the *race* column, and uses these as new column names. It then inspects each row in the original dataframe, looking for unique *driverId* values to act as unique row identifiers in the wide dataframe, then uses (by default) the contents of the rightmost column as the values of the new data cells (in this case, the `pos` column; use `value.var=` to specify another column). If there is no combination of the the row name and new column name in the original dataset, a *not available* NA designator is used.

We can also transform data from *wide* to *long* format, this time using the `melt()` function, which is also contained in the `reshape2` package.

```
#The melt function is also contained within the reshape package.
#If we've already loaded the package in, we don't need to load it again
head(melt( wide.df ), n=3)


## Using driverId as id variables
```

```
##          driverId  variable value
## 1          button Australia     3
## 2        hamilton Australia    NA
## 3 kevin_magnussen Australia     2
```

We can also pass in some additional parameters to the `melt()` function that tidy up the resulting data frame

```
melt( wide.df, id.vars=c("driverId"),
      variable.name = "race",
      value.name = "pos",
      na.rm=T )
```

```
##           driverId      race pos
## 1           button Australia   3
## 3  kevin_magnussen Australia   2
## 5           rosberg Australia   1
## 8          hamilton   Bahrain   1
## 10            perez   Bahrain   3
## 11          rosberg   Bahrain   2
## 14         hamilton  Malaysia   1
## 17          rosberg  Malaysia   2
## 18           vettel  Malaysia   3
```

(The *na.rm=T* (or equivalently, *na.rm=TRUE*) parameter setting can be clumsily read as *NA remove is true*, which is to say *remove rows for which it is true that the value is NA*.)

Split-Apply-Combine

The "split-apply-combine" recipe identified by Hadley Wickham as a widely used technique for extracting groups of a data from a dataset, operating on them, and then combining the results back together, is well supported using his `plyr` R package.

It can be quite a puzzle getting your head round some of the transformations, but there is a logic to it, honestly!

Let's use the `winner` dataframe again, and find the average (mean) position recorded by each team. To do that, we need to *split* the data frame into groups of rows by team, find the average (mean) position for each team (that is, for each separate group of rows) by *applying* the `mean()` function and then create a new dataframe by *combining* rows that contain the names of the teams and their average position. That is, we need to *split* - *apply* - *combine*. To do that, we use the `ddply()` function to *summarise* the data:

```
ddply(winners,.(constructorId),summarise, meanpos=mean(pos))
```

```
##   constructorId meanpos
## 1   force_india     3.0
## 2       mclaren     2.5
## 3      mercedes     1.4
## 4      red_bull     3.0
```

The length of the data frame is the same as the number of groups we are summarising.

Another way of using `ddply` is to *mutate* a dataframe. In this case, we can apply a function to each group and use the result in the definition of a new column on the original dataframe (note: the original dataframe is not changed; we need to assign the output of `ddply` to a new variable, or back to the original dataframe):

```
winners.new=ddply(winners[,c('carNum','constructorId','pos')],
                  .(constructorId),
                  mutate,
                  meanTeamPos=mean(pos))

winners.new=ddply(winners.new,
                  .(constructorId),
                  mutate,
                  meanTeamPosDelta=pos-mean(pos))

kable(winners.new)
```

carNum	constructorId	pos	meanTeamPos	meanTeamPosDelta
11	force_india	3	3.0	0.0
20	mclaren	2	2.5	-0.5
22	mclaren	3	2.5	0.5
6	mercedes	1	1.4	-0.4
44	mercedes	1	1.4	-0.4
6	mercedes	2	1.4	0.6
44	mercedes	1	1.4	-0.4
6	mercedes	2	1.4	0.6
1	red_bull	3	3.0	0.0

`ddply` is a very powerful function for executing *split-apply-combine* style data transformations. If you have ever used spreadsheet pivot tables, you may notice some resemblance to the *summarise* mode of operation. The *mutate* function additionally allows us apply the results of group based summaries to row based operations within each group of rows.

Summary

This chapter has provided a very quick introduction to the RStudio environment and the R prohramming language.

RStudio is a very powerful tool and can be used to author a wide range of document types that incorporate executable R code and the results of its execution. For example, you can use RStudio to produce slideshows that incorporate R code and the results of running it, or HTML or PDF documents generated from reusable templates. RStudio can also be used to support the development of R packages, which is something I need to learn how to do! To learn more about RStudio, check out the RStudio documentation[21], or one of the increasing number of books on RStudio[22].

As to the R language itself, we have introduced many of the key tecnhiques and functions that will be used throughout the rest of this book: how to cosntuct dataframes and read data in from CSV files, how to filter dataframes, how to sort (or *arrange*) them, *transform* them, reshape them from wide to long form and back again (*melt* and *dcast*), and process them by group using the *split-apply-combine* method via the `ddply` function.

Over the coming chapters, you will see more of the R language, in particular how it can be used to reshape data to get it into a format where we can easily visualise it. You will also see a lot more of ggplot2, as well as other approaches towards generating data visualisations from within R.

Exercises

In the `winners` dataframe, what levels are associated with the *constructorId* and *driverId* factors?

Filter the `winners` data frame to generate another dataframe that contains the rows for just the Force India and Mercedes teams. Further limit the derived dataframe to only show the `driverId` and `fastlaptime` columns.

[21] https://support.rstudio.com/hc/en-us/categories/200035113-Documentation

[22] http://www.amazon.co.uk/s/ref=nb_sb_noss_1?url=search-alias%3Dstripbooks&field-keywords=rstudio&tag=ouseful-21

Getting the data from the Ergast Motor Racing Database API

We can access the contents of the ergast motor racing database in three distinct ways:

- online, as tabular results in an HTML web page, for seasonal data up to and including the current season and results up to and including the last race;
- online, via the ergast API, for seasonal data up to and including the current season and results up to and including the last race;
- via a downloaded image of the database for results to the end of the last completed season.

There are also several third party applications that have been built on top of the ergast data. For further details, see the ergast Application Gallery[23].

Although it can be instructive to review the information available on the ergast website directly, as well as the applications that other people have built, we are more interested in accessing the actual data, whether by the API or the database. Whilst it doesn't really matter where we get the data from for the point of view of analysis, the API and the database offer slighlty different *affordances* when it comes to actually getting data out in a particular form. For example, the API requires a network connection for live data requests or to populate a cache (a local stored copy of data returned from an API request), whereas the database can be run offline but requires a database management system to serve the data in response to database requests. The API also provides data results that combines data from several separate database tables right from the start, whereas with the database we need to work out ourselves how to combine data from several separate data tables.

For convenience, I will tend to refer to *accessing the ergast API* when I mean calling the online API, and *accessing the ergast database* when it comes to querying a local database. However, you should not need to have to install the database for the majority of examples covered in this book - the API will work fine (and is essential when it comes to making queries about the current season). On the other hand, if you are looking for an opportunity to learn a little bit about databases and how to query them, now might be a good time to start!

[23] http://ergast.com/mrd/gallery

Accessing Data from the ergast API

If you have a web connection, one of the most convenient ways of accessing the ergast data is via the ergast API. An API is an *application programming interface* that allows applications to pull data directly from a remote service, such as a database management system, via a programmable interface. The provision of an API means that we can write a short programme to pull data directly from the ergast database that lives at *ergast.com* via the ergast API.

The ergast API publishes data as a JSON or XML data feed. Handling the data directly is a little fiddly, so I have started to put together a small library to make it easier to access this data, as well as enriching it. This type of library is often referred to as a *wrapper* in that it "wraps" the original HTTP/JSON API with a set of native R functions. *For more details, see the appendix.* The library can be found at ergastR-core.R[24] and currently contains the following functions:

- *driversData.df(YEAR)*: information about the drivers competing in a given year;
- *racesData.df(YEAR)*: details of the races that took place or are scheduled to take place in a given year;
- *resultsData.df(YEAR,RACENUMBER)*: results of races by year and racenumber;
- *raceWinner(YEAR,RACENUMBER)*: the winner of a race specified by year and race number;
- *lapsData.df(YEAR,RACENUMBER)*: information about laptimes during a particular race;
- *driverCareerStandings.df(DRIVERID)*: information about the career standing in terms of end of season classifications for a particular driver;
- *seasonStandings(YEAR,RACE?)*: championship standings at the end of the year, or (optionally), the end of a particular race.

On my to do list is learn how to put together a proper R package...

Introducing the simple ergastR functions

To load the core *ergastR* functions in, download the *raw* file[25] to the current working directory, and use the `source('ergastR-core.R')` command to load in the file. ALternatively, load the `devtools` package and use the `source_url()` function.

[24] https://gist.github.com/psychemedia/11187809#file-ergastr-core-r
[25] https://gist.github.com/psychemedia/11187809#file-ergastr-core-r

```
#If the R file is in the current working directory
source('ergastR-core.R')
#If you need to specify the path to the file
#source('~/Dropbox/wranglingf1datawithr/src/ergastR-core.R')

#You can also load the file in from the online gist
#Use the source_url() function from the devtools package
#Note that you may need to install the devtools package first
#library(devtools)
#source_url('https://gist.githubusercontent.com/psychemedia/11187809/raw/ergastR-core\
.R')
```

Let's look at a preview of each table in turn. We can do this using the R function *head()*, which displays just the first few rows (10 by default) of a dataframe. For example, *head(df)* previews the first 10 rows for the dataframe *df*. To alter the number of rows displayed, for example to 5, use the construction *head(df,**n=5**)*. To view the rows at the end of the table, you can use the *tail()* command in a similar way.

```
#USAGE: driversData.df(YEAR)
drivers.df = driversData.df(2014)

#The knitr library contains a handy function - kable - for generating tabular markdow\
n.
#We can use it in an Rmd script by setting an Rmd chunk with the option {r results='a\
sis'}
#Note that /format='markdown'/ is actually the default output for kable.
kable(head(drivers.df),row.names=F,format="markdown")
```

driverId	name	code	permNumber
alonso	Alonso	ALO	14
jules_bianchi	Bianchi	BIA	17
bottas	Bottas	BOT	77
button	Button	BUT	22
chilton	Chilton	CHI	4
ericsson	Ericsson	ERI	9

In the ergast database, the `driverId` is used to distinguish each driver. The `driversData.df()` function can thus be used to provide additional information about drivers from their `driverId`, such as their new permanent number and their three letter driver code.

When it comes to identifying races, we need two pieces of information. The `year` and the `round`. We can look up races by year by calling `racesData.df()` with the year of interest:

```
#USAGE: racesData.df(YEAR)
races.df = racesData.df(2014)
```

round	racename	circuitId
1	Australian Grand Prix	albert_park
2	Malaysian Grand Prix	sepang
3	Bahrain Grand Prix	bahrain
4	Chinese Grand Prix	shanghai
5	Spanish Grand Prix	catalunya
6	Monaco Grand Prix	monaco

Knowing the round number we are interested in then allows us to look up data about a particular race. For example, let's look at the first few lines of the results data for the 2014 Malyasian Grand Prix, which happened to be round 2 of that year:

```
#USAGE: resultsData.df(YEAR,RACENUMBER)
results.df = resultsData.df(2014,2)

kable(head(results.df))
```

carNum pos driverId constructorId grid laps status fastlapnum fastlaptime fastlaprank —— - —- ———— ———— —— —— ———— ———— ———— ———— 44 1 hamilton mercedes 1 56 Finished 53 103.066 1 6 2 rosberg mercedes 3 56 Finished 55 103.960 2 1 3 vettel red_bull 2 56 Finished 51 104.289 4 14 4 alonso ferrari 4 56 Finished 47 104.165 3 27 5 hulkenberg force_india 7 56 Finished 38 105.982 10 22 6 button mclaren 10 56 Finished 47 106.039 11

Having access to laptime data is essential for many race reports. The `lapsData.df()` function returns laptime data for each driver during a particular race.

```
#USAGE: lapsData.df(YEAR,RACENUMBER)
laps.df = lapsData.df(2014,2)
head(laps.df)
```

```
##   lap driverId position  strtime rawtime    cuml   diff
## 1   1 hamilton        1 1:51.824 111.824 111.824     NA
## 2   2 hamilton        1 1:47.501 107.501 219.325 -4.323
## 3   3 hamilton        1 1:47.763 107.763 327.088  0.262
## 4   4 hamilton        1 1:48.375 108.375 435.463  0.612
## 5   5 hamilton        1 1:47.428 107.428 542.891 -0.947
## 6   6 hamilton        1 1:47.532 107.532 650.423  0.104
```

Note that the `cuml` and `diff` columns are not returned by the ergast API - I have generated them by ordering the laps for each driver by increasing lap number and then calculating the cumulative live time and the difference between consecutive lap times for each driver separately. *We will see how to do this in a later section.*

We can look up the winner of that race using the `raceWinner()` function:

```
#USAGE: raceWinner(YEAR,RACENUMBER)
winner = raceWinner(2014,2)
winner
```

```
## [1] "hamilton"
```

The `raceWinner()` function makes a specific call to the ergast API to pull back the driverId for a particular position in a particular year's race.

To inspect the construction of the `raceWinner()` function, we just enter its name without any argument brackets:

```
raceWinner
```

```
## function (year, raceNum)
## {
##     wURL = paste(API_PATH, year, "/", raceNum, "/results/1.json",
##         sep = "")
##     wd = fromJSON(wURL, simplify = FALSE)
##     wd$MRData$RaceTable$Races[[1]]$Results[[1]]$Driver$driverId
## }
```

We see how the URL for the corresponding request takes the form `http://ergast.com/api/f1/YEAR/RACE` (API_PATH is set to *http://ergast.com/api/f1/*). For the winner, the construction of the URL thus includes the term **1**.json).We could create a more general function that makes a call for information relating to an arbitrary position, not just first place by parameterising this part of the URL's construction.

That is, we might try something of the form:

```
#Pass in a race position, by default setting it to first place
racePosition = function (year, raceNum, racePos=1) {
     wURL = paste(API_PATH,
                  year, "/", raceNum, "/results/", racePos,
                  ".json", sep = "")
     wd = fromJSON(wURL, simplify = FALSE)
     wd$MRData$RaceTable$Races[[1]]$Results[[1]]$Driver$driverId
}

racePosition(2014,2,3)
```

```
## [1] "vettel"
```

As and when you develop new fragments of R code, it often makes sense to wrap them up into a function to make the code easier to reuse. By adding *parameters* to a function, you can write create *general* functions that return *specific* results dependent on the parameters you pass into them. For data analysis, we often want to write very small pieces of code, or particular functions, that do very specific things, rather than writing large very large software programmes. Writing small code fragments in this way, and embedding them in explanatory or discursive text, is an approach referred to as *literate programming*. Perhaps we need to start to think of programming-as-coding as more to do with writing short haikus than long epics?!

If you compare the two functions above, you will see how they resemble each other almost completely. By learning to *read* code functions, you can often recognise bits that can be modified to create new functions, or more generalised ones. We have taken the latter approach in the above case, replacing a specific character in the first function with a parameter in the second. (That is, we have further *parameterised* the original function.)

Indexing in to a dataframe

The `racePosition()` function lets us pull back the details of the driver who finished a particular race in a particular year in a particular position. Another way of finding the driver who finished a particular eace in a particular position is by indexing into the results dataframe as defined by the ergast API call `resultsData.df(YEAR, RACENUMBER)`. Let's filter that dataframe by selecting the *row* corresponding to a particular position, and the column that contains the driver ID.

```
results.df[results.df$pos==1,c('driverId')]
```

```
## [1] hamilton
## 22 Levels: vettel ricciardo chilton rosberg raikkonen ... sutil
```

Don't worry about the reporting of the other factor levels in the result that is displayed. If we call on the particular result, only the request value is returned; for example, I can embed the driver ID that is returned here: hamilton.

Merging dataframes in R

As you might imagine, one of the very powerful tools we have to hand when working in R is the ability to merge two dataframes, in whole or in part.

We can *merge* data from two different tables if they each contain a column whose unique values match each other. For example, the `results.df` dataframe contains a column `driverId` that contains a unique ID for each driver (*hamiliton*, *vettel*, and so on). The `driverId` column in the `drivers.df` datafram pulls from the same set of values, and contains additional information about each driver. If we want to augment `results.df` with an additional column that contains the three letter driver code for each driver, we can do that using R's `merge()` function, assigning the result back to `results.df`.

```
#We can pull just the columns we want from drivers.df
#We want all rows from drivers.df,
#but just the 'driverId' and 'code' columns
head( drivers.df[,c('driverId','code')] )
```

	driverId	code
driverId	alonso	ALO
driverId1	jules_bianchi	BIA
driverId2	bottas	BOT
driverId3	button	BUT
driverId4	chilton	CHI
driverId5	ericsson	ERI

To merge the dataframes, we specify which dataframes we wish to merge and the column on which to merge. The *order* in which we identify the dataframes is important because there are actually several different sorts of merge possible that take into account what to do if the the merge column in the first table contains a slightly different set of unique values than does the merge column in the second table. *We will review the consequences of non-matching merge column values in a later section.*

```
results.df = merge( results.df, drivers.df[,c('driverId','code')], by='driverId')
```

driverId	carNum	pos	constructor	grid	laps	status	fastlapnum	fastlaptime	fastlaprank	code
alonso	14	4	ferrari	4	56	Finished	47	104.165	3	ALO
bottas	77	8	williams	18	56	Finished	31	105.475	9	BOT
button	22	6	mclaren	10	56	Finished	47	106.039	11	BUT

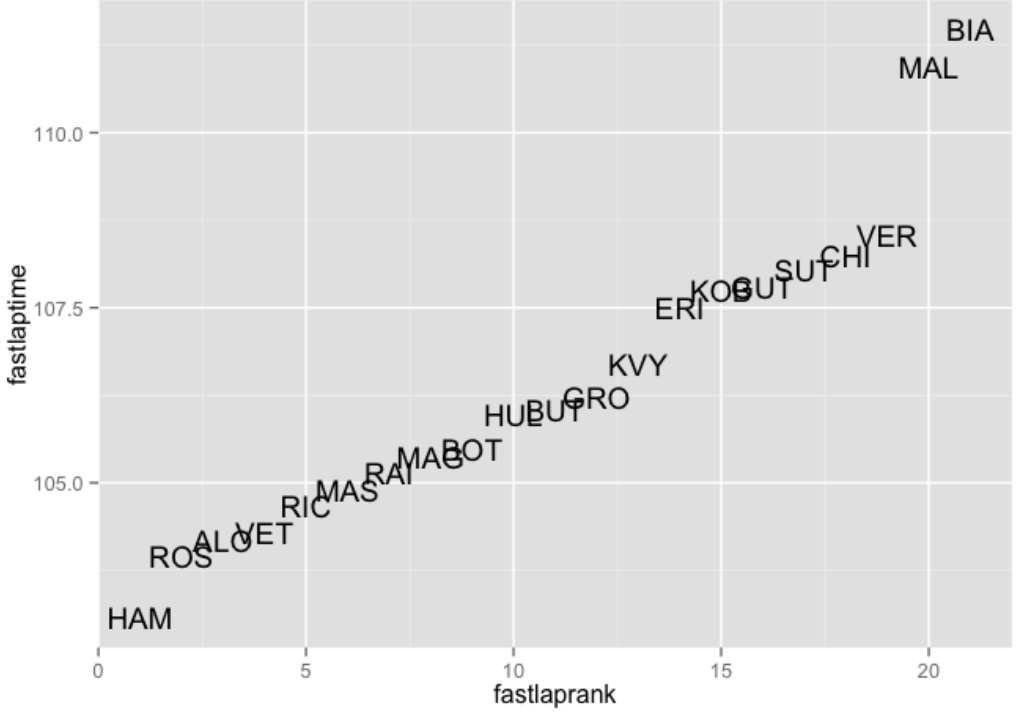
If the columns you want to merge on actually have *different* names, they can by specified explicitly. The first dataframe is referred to as the *x* dataframe and the second one as the *y* dataframe; their merge columns names are then declared explicitly:

```
#Filter the drivers.df dataframe to just the driverId and code columns
driverIds.df = drivers.df[,c('driverId','code')]
#The "x" dataframe is the first one we pass in, the "y" dataframe the second
laps.df = merge( laps.df, driverIds.df, by.x='driverId', by.y='driverId')
head( laps.df, n=3 )
```

```
##   driverId lap position  strtime rawtime    cuml   diff code
## 1   alonso   1        5 1:56.440 116.440 116.440     NA  ALO
## 2   alonso   2        5 1:49.154 109.154 225.594 -7.286  ALO
## 3   alonso   3        5 1:48.219 108.219 333.813 -0.935  ALO
```

Having the three letter code available in the dataframe directly means we can access it easily when it comes to producing a chart. For example, we might plot the fastest lap time against the fastest lap rank, using the code to identify each point:

```
#Load in the required charting library
require(ggplot2)

#Generate a text plot, a scatterplot with textual labels placed at each scatterplot p\
oint
g = ggplot(results.df) + geom_text(aes( x=fastlaprank, y=fastlaptime, label=code))
g


## Warning: Removed 1 rows containing missing values (geom_text).
```



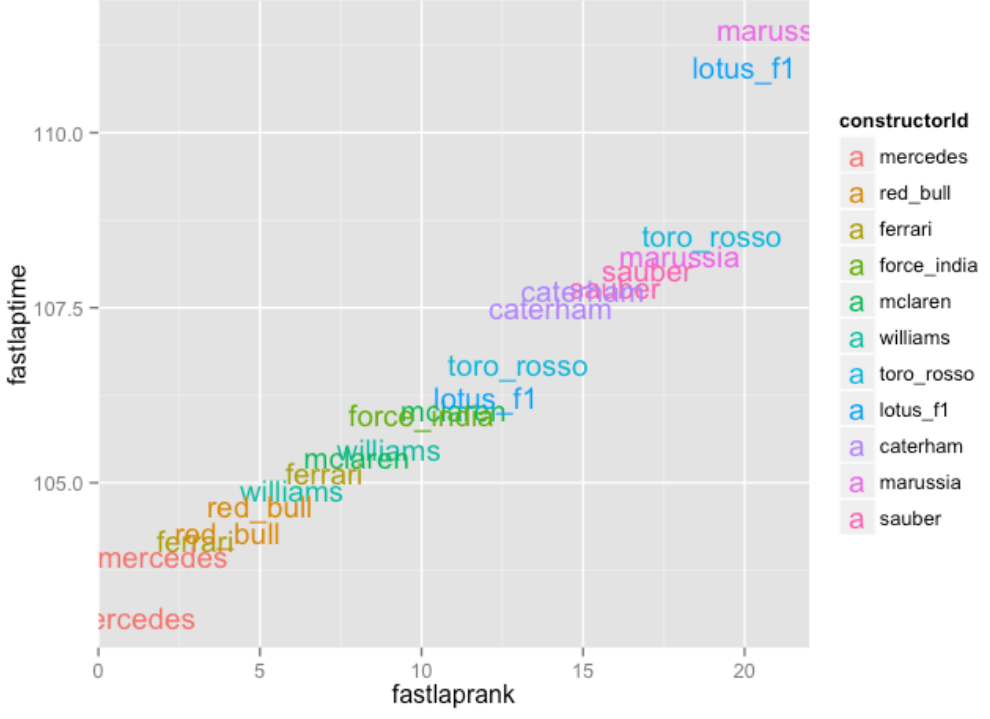
Text plot showing fast lap time versus rank

The warning tells us that data from one row in the dataframe was not plotted, presumably because one or other of the x or y values was missing (that is, set to NA).

Although it's just a simple chart, we can see how the drivers' fastest laptimes split into several groups. Are these actually grouped by team? Let's see... Rather than use the driver code for the labels, let's use `constructorId`, further colouring the labels based on the value of the `constructorId`.

```
g = ggplot(results.df)
g = g + geom_text(aes( x=fastlaprank, y=fastlaptime, label=constructorId, col=constru\
ctorId))
g
```

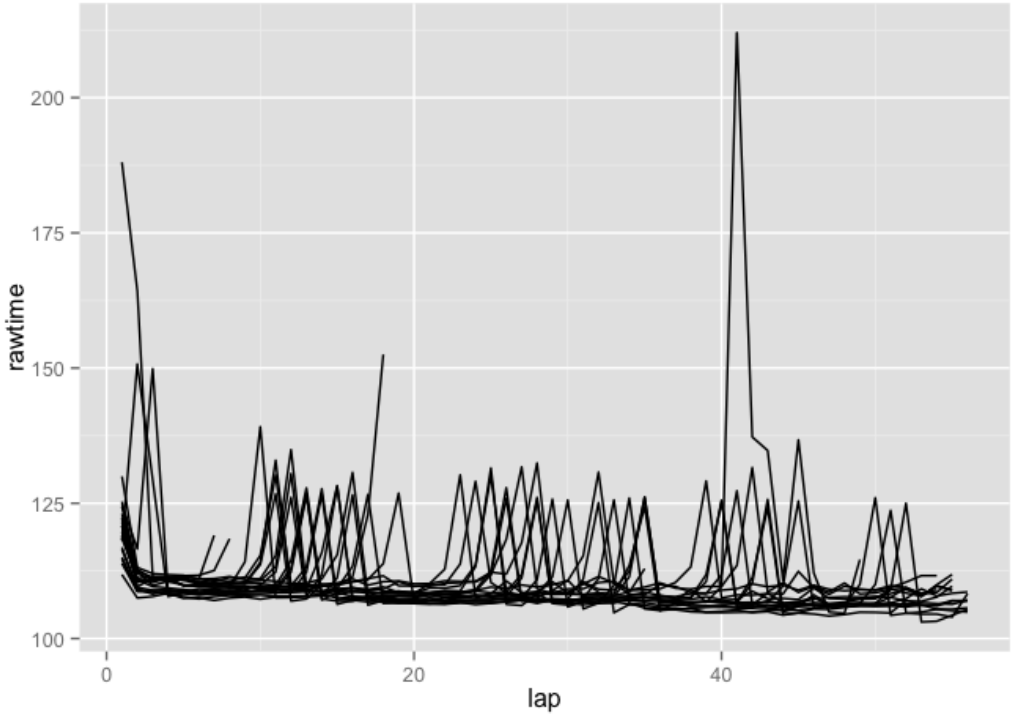
```
## Warning: Removed 1 rows containing missing values (geom_text).
```



Text plot showing fast lap time versus rank, coloured by team

We can also generate line charts - for example, here's a quick look at the lap times recorded in a particular race:

```
ggplot(laps.df) + geom_line(aes(x=lap , y=rawtime, group=driverId))
```



We'll see later how we can tidy up charts like these by adding in a chart title, tweaking the axis labels and playing with the overall chart style. But for now, you can see how quick it is to start sketching out graphical views of data, if the data is in the right shape and format to start with.

Summary

In this chapter, we have seen how we can make calls to the ergast API using some predefined R functions contained in the ergastR-core.R source file[26]. The R functions in this file *wrap* the

[26]https://gist.github.com/psychemedia/11187809#file-ergastr-core-r

original ergast API with typically dataframe returning functions that you can use directly in your own R programmes.

In particular, we have seen how we can pull back data covering the drivers or races involved in a particular chanpionship; the results for a particular race; the winning driver from a particular race (and by extension of that function, the driver finishing in a any specified position of a given race); the lap times for a particular race; and the careerwise standings for a particular driver in terms of their position at the end of each season in which they competed.

Whilst the source file does not cover the whole of the ergast API (though perhaps future versions will!) it does provide a good starting point in the form of access to some of the key data sets. By inspecting the current functions, and looking at the data returned from unwrapped ergast API functions, you may find you are able to extend, and probably even improve on, the library by yourself.

We also saw how to filter and merge dataframes, putting some simple combined data into a shape that we could plot using the *ggplot2* package. Whilst the charts we generated were quite scruffy, they hopefully gave you a taste of what's possible. The chapters that follow are filled with a wide range of different visualisation techniques, as well as covering in rather more detail several ways of making you charts look rather tidier!

Exercises

*The functions in **ergastR-core.R** are a little scrappy and could be defined so that they more closely resemble the ergast API definition as described the API URLs. They should also really be put into a proper R package. I don't know how to do this (yet!) so if you'd like to help - or take on - the development of a properly defined ergast API R package, please let me know...*

A Quick Look at Qualifying

The qualifying session differs from the other race weekend sessions in that the session is split into three parts, with results available from each for the participating drivers. The qualifying session itself determines the grid position for the race for each driver (penalties aside) and as such may be a predictor of the race winner. When trying to predict target split times for each of the qualifying sessions, the third practice times may give a useful steer. We will explore just how good a predictor the P3 times are for qualifying split times in a later chapter.

As with the practise sessions, we can get results data from the ergast database and results, speeds and sector times from the F1 website. To begin with, let's look at various ways in which we can summarise the results of the qualifying sessions. I'm going to use the F1 data - so let's see which table we need.

```
library(DBI)
f1 =dbConnect(RSQLite::SQLite(), './/scraperwiki.sqlite')
## list all tables
dbListTables(f1)
```

```
##  [1] "p1Results"    "p1Sectors"    "p1Speeds"     "p2Results"
##  [5] "p2Sectors"    "p2Speeds"     "p3Results"    "p3Sectors"
##  [9] "p3Speeds"     "qualiResults" "qualiSectors" "qualiSpeeds"
## [13] "raceFastlaps" "racePits"     "raceResults"
```

The data we want is in the *qualiResults* table. Let's have a quick look at a sample of the data from the 2014 Chinese Grand Prix.

```
qualiResults=dbGetQuery(f1,
                        'SELECT * FROM qualiResults
                        WHERE race="CHINA" AND year="2014"')
```

q1time	driverNum	pos	q1natTime	q2time	race	q3time	year	q2natTime	q3natTime	team	qlaps	driverName
115.926	1	3	1:55.926	114.499	CHINA	114.96	2014	1:54.499	1:54.960	Red Bull Racing-Renault	23	Sebastian Vettel
119.260	10	18	1:59.260	0.000	CHINA	0.00	2014			Caterham-Renault	10	Kamui Kobayashi
118.362	11	16	1:58.362	118.264	CHINA	0.00	2014	1:58.264		Force India-Mercedes	17	Sergio Perez

Note that we could make this a little more reusable by splitting out the arguments and wrapping the query in a function. For example:

```
f1.getQualiResults =function (race='CHINA',year='2014'){
  q=paste('SELECT * FROM qualiResults
          WHERE race="',race,'" AND year="', year,'"', sep='')
  dbGetQuery(f1,q)
  }
#Call the function using an expression of the form:
## f1.getQualiResults('AUSTRALIA','2013')
```

By inspection of the data table, we see that there are separate columns for the best time recorded by each driver in each session, and a rank position for the session overall. One way of plotting the results data on a single chart is to use the session number (Q1, Q2 or Q3) as a categorical horizontal x-axis value, and the time (or rank) achieved by a driver in a particular session on the vertical y-axis.

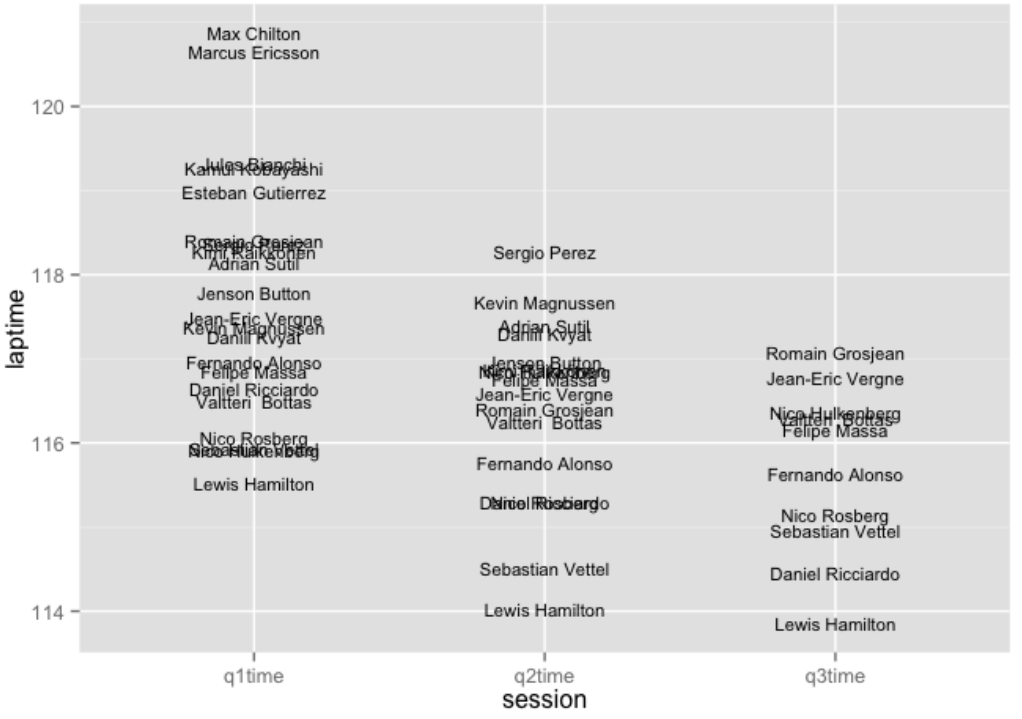
The easiest way to generate such a chart is to assemble the data so that the session time (or rank) is in one column, and the session identifier in another. We can use the *melt()* command to reshape the data into this form.

```
library(reshape2)

#Generate a new table with columns relating to driverNum, driverName, session and ses\
sion laptime
qm=melt(qualiResults,
        id=c('driverNum','driverName'),
        measure=c('q1time','q2time','q3time'),
        variable.name='session',
        value.name='laptime')

#Make sure that the laptime values are treated as numeric quantities
qm$laptime=as.numeric(qm$laptime)
#If a laptime is recorded as 0 seconds, set it to NA
qm[qm == 0] <- NA
#Drop any rows with NA laptime values
qm=qm[with(qm,!is.na(laptime)),]
#This should give us 16 drivers in Q2 and 10 in Q3, assuming the drivers set times
#If a driver doesn't set a time in a session they are in, we may need to rethink...
```

driverNum	driverName	session	laptime
1	Sebastian Vettel	q1time	115.926
10	Kamui Kobayashi	q1time	119.260
11	Sergio Perez	q1time	118.362

With the data in shape, let's have a look at it. To start with, we'll order drivers by laptime within each session.

```
library(ggplot2)

ggplot(qm)+geom_text(aes(x=session,y=laptime,label=driverName),size=3)
```



Text plot showing relative qualifying session times

This chart shows several things, and the potential to show many more. For example, it *does* show:

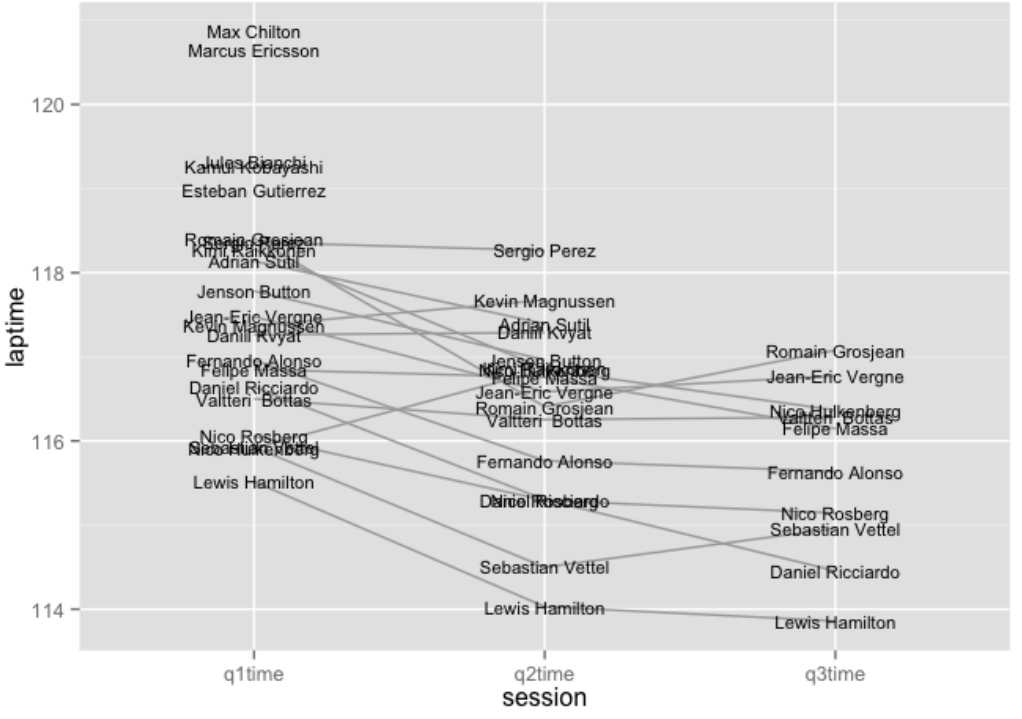
- how laptimes improved session on session;
- how there is separation (or not) between drivers.

However, it *does not* readily show which drivers did not make it through from one session to the next, nor where the split time was that separates the drivers who make it through from those who don't. (Remember, at the end of Q1, only the top 16 drivers go on to Q2, and from there only the top 10 make it into Q3.)

The chart also suffers when drivers' laptimes are very close, as they are quite likely to be in qualifying.

We can use lines to act as connectors that show how any particular driver faired in their attempt to progress from one qualifying session to the next.

```
g = ggplot(qm,aes(x=session,y=laptime))
g = g+ geom_line(aes(group=driverName),colour='darkgrey')
g+geom_text(aes(label=driverName),size=3)
```



Text plot of qualifying session times with driver connector lines

That's a slight improvement, but there are still questions around overlap and the highlighting of drivers that didn't make the cut.

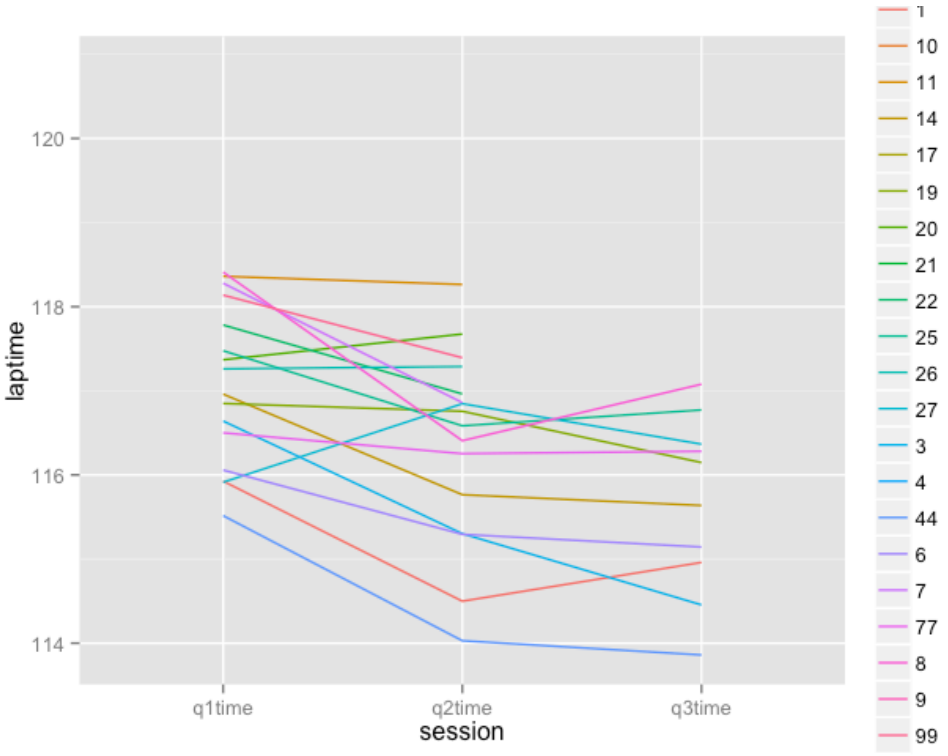
One way we can get round the overlap problem is to introduce an equal amount of separation between the driver names by plotting them against position (that is, rank) values rather than laptime. The upside of this approach which be separation of labels, the downside a loss of information about separation. Perhaps we could combine the appraoches?

In the next section we'll exploring the ranking, and the bare bones (unlabelled) qualifying session position summary charts, and then we'll look at how we might combine rank and laptime based views.

Qualifying Session Position Summary Chart

One useful way of summarising the qualifying session is to put together a session results summary chart that shows position of each driver at the end of each qualifying session.

```
ggplot(qm)+geom_line(aes(x=session,y=laptime,group=driverNum,col=driverNum))
```



Custom line chart showing lap times by qualifying session

The above table shows the relative positioning of the drivers in terms of laptime during each qualifying session. Another perspective might be to look at the just the rank positions. In order to do that, we need to select the drivers who participated in each session (which, as described above, we're taking to be drivers with non-zero laptimes recorded in a session) and then rank them.

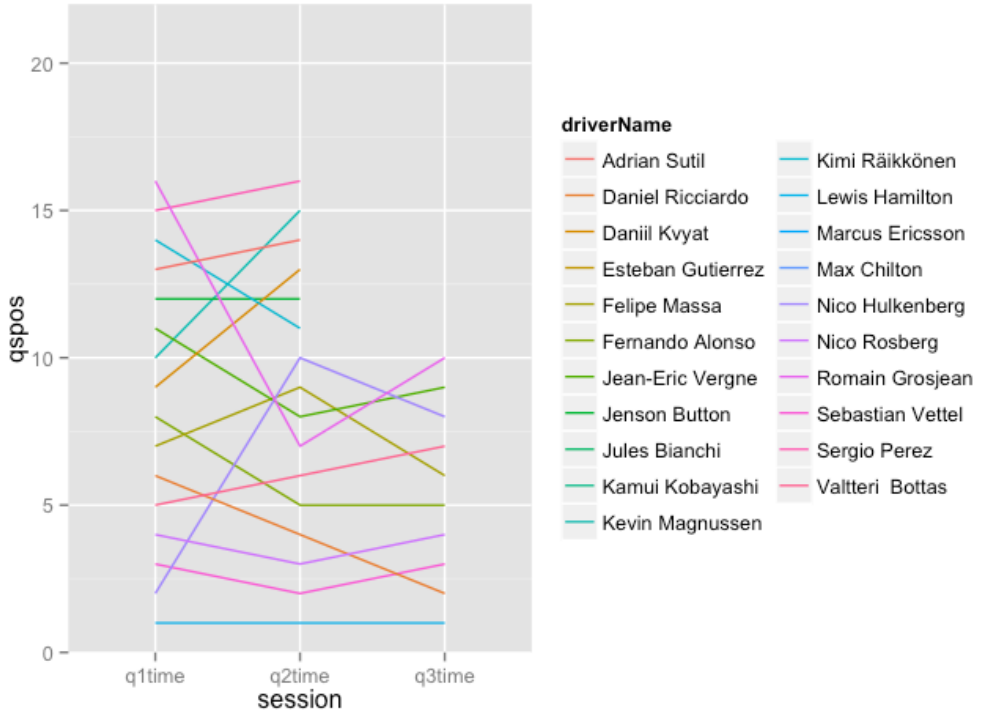
If we group the laptimes by session, in decreasing position order, we can rank each row within the group.

```
library(plyr)
qm=ddply(qm,'session',mutate,qspos=rank(laptime))
```

driverNum	driverName	session	laptime	qspos
8	Romain Grosjean	q2time	116.407	7
99	Adrian Sutil	q2time	117.393	14
1	Sebastian Vettel	q3time	114.960	3
14	Fernando Alonso	q3time	115.637	5
19	Felipe Massa	q3time	116.147	6
25	Jean-Eric Vergne	q3time	116.773	9
27	Nico Hulkenberg	q3time	116.366	8
3	Daniel Ricciardo	q3time	114.455	2
44	Lewis Hamilton	q3time	113.860	1
6	Nico Rosberg	q3time	115.143	4
77	Valtteri Bottas	q3time	116.282	7
8	Romain Grosjean	q3time	117.079	10

We can now plot the rank positions within each session. Let's additionally tweak the legend to tidy it up a bit, by putting the names into two columns.

```
g=ggplot(qm)+geom_line(aes(x=session,y=qspos,group=driverNum,col=driverName))
g+guides(col=guide_legend(ncol=2))
```



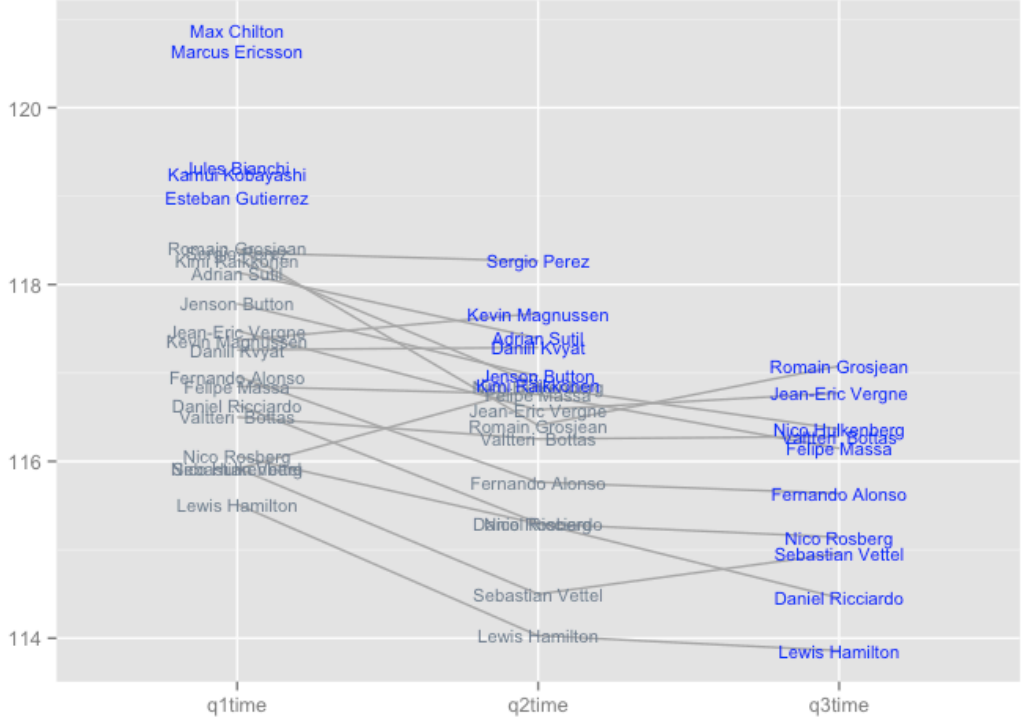
Custom line chart showing rank by qualifying session

Another Look at the Session Tables

In many reporting situations we may be as keen to know who went *out* in a particular session as much as who got through to the next session. So let's highlight the drivers who didn't make the cut in the first and second qualifying sessions, along with the drivers who did get through to Q3.

With the rank position available within each session, we can highlight the drivers likely to be of interest in each qualifying session.

```
g = ggplot(qm,aes(x=session,y=laptime))
g = g+ geom_line(aes(group=driverName),colour='darkgrey')

#Highlight the drivers of interest in each session
g= g+geom_text(aes(label=driverName,
                   colour=(((qspos>16 & session=='q1time') |
                            (qspos>10 & session=='q2time') |
                            (session=='q3time') ))
                   ), size=3)
#Define the colouring for the text labels
g=g+scale_colour_manual(values=c('slategrey','blue'))
#Hide the legend for the colouring
g=g+guides(colour=FALSE)
g+xlab(NULL)+ylab(NULL)
```



Session times table, with highlights

(Note, if we had included overall position in the *qm* dataframe originally we could have used

the *pos* value to highlight the values earlier. Even though positions may change between drivers moving from session to session, the fact that they made it through to a particular session will be given by their overall position in qualifying as a whole.)

The above chart is reminiscient of, athough far more cluttered than, a very clean chart type known as a *slopegraph* originally proposed by Edward Tufte. We will explore the use of slopegraphs in a future update to this chapter.

Ultimate Lap Positions

As the aim of the qualifying session is to put together the fastest lap, it can be useful to know whether the best laptime acheived by each driver was actually the same as their ultimate lap. Where the two differ, *if* a driver *had* hooked up their ultimate lap, might the final outcome in terms of grid position have been any different? That's what we'll explore in this section… but not just now: in a future update to this chapter!

Career Trajectory

*This chapter was originally inspired by the **Career Trajectories** chapter of **Analyzing Baseball Data with R**[27] (2013) by Max Marchi & Jim Albert.*

With ever younger drivers entering Formula One, and a culture in which certain teams drop drivers before they really have time to mature, an interesting to question to ask is what sort of career trajectory, or profile, do longlasting drivers tend to have. Did early success guarantee them a long dotage on the gird? Or have they been better than average journeymen, good team players who were always thereabouts in the top 10 but rarely actually there on the podium? Or did they begin as outperformers in a lowly team and steadily make their way up the grid to later success?

In this chapter we'll explore various ways of looking at - and modeling - career trajectories. Some statistical modeling is involved, but it will be presented in mainly a graphical form.

The data we're going to use comes from the *ergast* database, and represents the career history of selected drivers.

```
require(RSQLite)

con_ergastdb = dbConnect(drv='SQLite', dbname='./ergastdb13.sqlite')
#Helper function to display database query result as a formatted table
kdb=function(q){ kable(dbGetQuery(con_ergastdb,q)) }

dbGetQuery(con_ergastdb,'SELECT name FROM sqlite_master WHERE type = "table"')
```

```
                name
```

1 circuits 2 constructorResults 3 constructorStandings 4 constructors 5 driverStandings 6 drivers 7 lapTimes 8 pitStops 9 qualifying 10 races 11 results 12 seasons 13 status

The data itself is spread across several tables in the local database. For example, the *drivers* table includes the date of birth, which allows us to calculate a driver's age, and nationality. Knowing a driver's nationality would allow us to compare it with the nationality of any

[27] http://baseballwithr.wordpress.com/about/

teams they drive for to see if there are any effects on performance there; or with the country in which a particular race takes place to see if there appears to be a "home race" effect on performance.

```
kdb('SELECT * FROM drivers limit 3')
```

driverId	driverRef	code	forename	surname	dob	nationality	url
1	hamilton	HAM	Lewis	Hamilton	1985-01-07	British	http://en.wikipedia.org/wi Hamilton
2	heidfeld	HEI	Nick	Heidfeld	1977-05-10	German	http://en.wikipedia.org/wi Heidfeld
3	rosberg	ROS	Nico	Rosberg	1985-06-27	German	http://en.wikipedia.org/wi Rosberg

The *driverStandings* table contains the Drivers' Championship standing of each driver at the end of each year, which allows us to keep track of how well they performed overall across several seasons. To know which team a driver was competing for, we'd need to reference yet another table.

If we access the ergast API directly, some of the linking and annotation work is done for us. For example, the *driverStandings* API call (example[28]) includes information about the constructor a driver (primarily?) drove for, along with the driver's championship position at the end of each round, the number of their wins and the number of points they collected.

```
#ergastR-core.R contains utility functions for accessing the ergast API
#and returning the results in a dataframe
source('./ergastR-core.R')
alonso = driverCareerStandings.df('alonso')
kable(alonso)
```

[28] http://ergast.com/api/f1/drivers/alonso/driverStandings

year	driverId	pos	points	wins	car
2001	alonso	23	0	0	minardi
2003	alonso	6	55	1	renault
2004	alonso	4	59	0	renault
2005	alonso	1	133	7	renault
2006	alonso	1	134	7	renault
2007	alonso	3	109	4	mclaren
2008	alonso	5	61	2	renault
2009	alonso	9	26	0	renault
2010	alonso	2	252	5	ferrari
2011	alonso	4	257	1	ferrari
2012	alonso	2	278	3	ferrari
2013	alonso	2	242	2	ferrari

For convenience, we can create a local temporary table that includes much of the same information. To start with, we need to identify the final round of each championship year - the *CREATE TEMPORARY VIEW* SQL command creates a temporary table that we can work with as if it were any other table.

```
dbGetQuery(con_ergastdb,
           'CREATE TEMPORARY VIEW lastRounds AS
           SELECT s.year, r.raceId, r.round, r.name, s.maxRound
           FROM (SELECT year, MAX(round) maxRound FROM races GROUP BY year) s
           JOIN races r
           WHERE r.round=s.maxRound AND r.year=s.year')
```

```
kdb('SELECT * from lastRounds LIMIT 3')
```

year	raceId	round	name	maxRound
1950	839	7	Italian Grand Prix	7
1951	832	8	Spanish Grand Prix	8
1952	824	8	Italian Grand Prix	8

We can then generate a view similar to the API championship *driverStandings* results. Note that the team affiliation is the team the driver was competing for in the last round of the championship, rather than the team they competed for most in the season, for example.

```
dbGetQuery(con_ergastdb,
           'CREATE TEMPORARY VIEW driverChampionship AS
           SELECT year, ds.driverId, driverRef, constructorRef,
                  ds.points, ds.position AS pos, wins
           FROM driverStandings ds
              JOIN drivers d
              JOIN lastRounds lr
              JOIN results r
              JOIN constructors c
           WHERE ds.driverId=d.driverId
              AND r.driverId=ds.driverId
              AND ds.raceId=lr.raceId
              AND r.raceId=lr.raceId
              AND r.constructorId=c.constructorId')
```

```
kdb('SELECT * from driverChampionship WHERE driverRef="alonso"')
```

year	driverId	driverRef	constructorRef	points	pos	wins
2001	4	alonso	minardi	0	23	0
2003	4	alonso	renault	55	6	1
2004	4	alonso	renault	59	4	0
2005	4	alonso	renault	133	1	7
2006	4	alonso	renault	134	1	7
2007	4	alonso	mclaren	109	3	4
2008	4	alonso	renault	61	5	2
2009	4	alonso	renault	26	9	0
2010	4	alonso	ferrari	252	2	5
2011	4	alonso	ferrari	257	4	1
2012	4	alonso	ferrari	278	2	3
2013	4	alonso	ferrari	242	2	2

The Effect of Age on Performance

At first glance, it might seem that asking whether performance appears to track age appears to be a relatively simple and straightforward question: *is a driver's performance somehow related to his age?*

But what do we mean by *age*? If we're using "number of years old" as our age figure when

keeping track of how well a driver performs in a particular season, is that their age (in years) at the start of the season? Or the end of the season? Or midway through the season? Would age in months be better? For example, does the time of year in which their birthday falls make a difference? Or how about if we want to compare the career performance of drivers with birthdays in early January, mid-July and late December? Or is age more a function of their "F1 age", the number of years they have been competing at that level, or the number of races they have competed in, or finished?

To start with, let's try to keep things as simple as possible and consider the career in terms of season standings of a single driver, in this case Fernando Alonso. As his age, will we use the number of years between his year of birth and championship years.

We could get the driver data back from the API using a function with the following form:

```
getYearFromDate=function(date){
  as.numeric(format(as.Date(date), "%Y"))
}

driverData.list=function(driverRef){
  dURL=paste(API_PATH,'drivers/',driverRef,'.json',sep='')
  drj=getJSONbyURL(dURL)
  dd=drj$MRData$DriverTable$Drivers[[1]]
  list(
    dateOfBirth=as.Date(dd$dateOfBirth),
    driverId=dd$driverId,
    nationality=dd$nationality,
    yearOfBirth=getYearFromDate(as.Date(dd$dateOfBirth))
  )
}

driverData.list('alonso')
```

$dateOfBirth [1] "1981-07-29"

$driverId [1] "alonso"

$nationality [1] "Spanish"

$yearOfBirth [1] 1981

The *getYearFromDate()* function extracts the birth year from the date of birth.

Alternatively, we can call the local database and then annotate the result with the birth year:

```
driverData=function (driverRef){
  q= paste('SELECT * FROM drivers WHERE driverRef== "',driverRef,'"',sep='')
  df=dbGetQuery(con_ergastdb,q)
  df$yearOfBirth=getYearFromDate(as.Date(df$dob))
  df
}
kable( driverData('alonso') )
```

driverId	driverRef	code	forename	surname	dob	nationality	url	yearOfBirth
4	alonso	ALO	Fernando	Alonso	1981-07-29	Spanish	http://en.wikipedia.org/wiki/Fernando_Alonso	1981

We can then use the birth year to find the age of a driver (at least, approximately) in each year of their career, calculated as *championship year - birth year*.

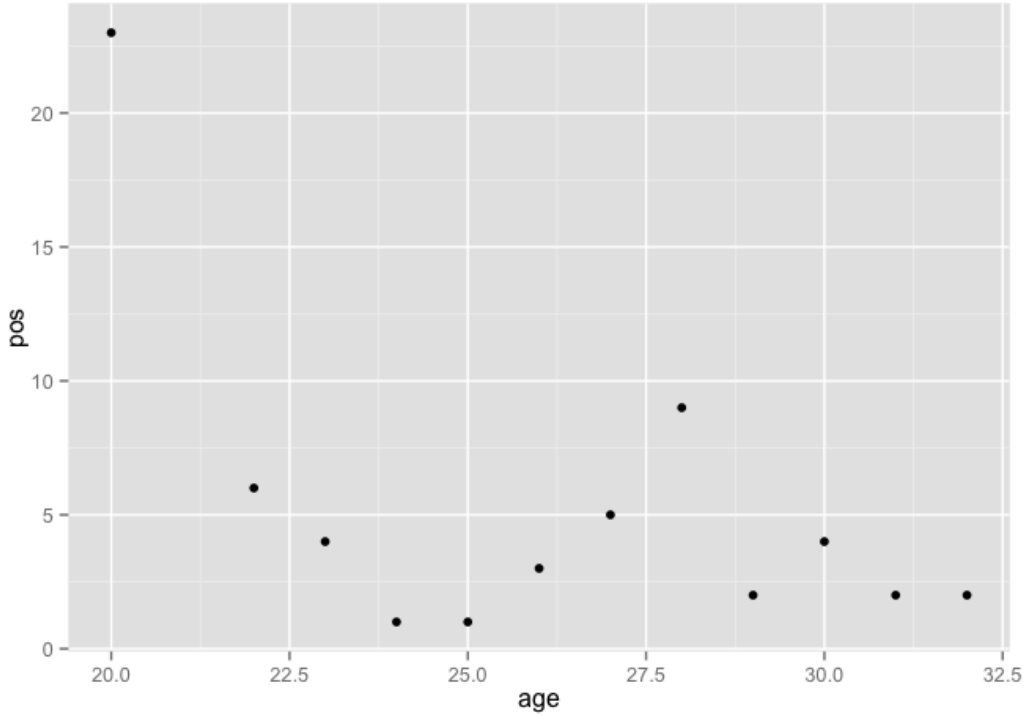
```
drivercareer.aug=function(driverRef){
  ##API equivalent calls:
  #drivercareer=driverCareerStandings.df(driverRef)
  #driverdata=driverData.list(driverRef)
  q=paste('SELECT * from driverChampionship WHERE driverRef="',driverRef,'"',sep='')
  drivercareer=dbGetQuery(con_ergastdb, q)
  driverdata=driverData(driverRef)
  drivercareer$age=drivercareer$year-driverdata$yearOfBirth
  drivercareer
}

drivercareer=drivercareer.aug('alonso')
kable( drivercareer )
```

year	driverId	driverRef	constructorRef	points	pos	wins	age
2001	4	alonso	minardi	0	23	0	20
2003	4	alonso	renault	55	6	1	22
2004	4	alonso	renault	59	4	0	23
2005	4	alonso	renault	133	1	7	24
2006	4	alonso	renault	134	1	7	25
2007	4	alonso	mclaren	109	3	4	26
2008	4	alonso	renault	61	5	2	27
2009	4	alonso	renault	26	9	0	28
2010	4	alonso	ferrari	252	2	5	29
2011	4	alonso	ferrari	257	4	1	30

year	driverId	driverRef	constructorRef	points	pos	wins	age
2012	4	alonso	ferrari	278	2	3	31
2013	4	alonso	ferrari	242	2	2	32

For Fernando Alonso, let's see how his career faired according to age by plotting his driver championship position against his age in years.

```
require(ggplot2)
ggplot(drivercareer)+geom_point(aes(x=age,y=pos))
```



Fernando Alonso's championship positions versus his age in years

Statistical Models of Career Trajectories

When it comes to trying to *model* career trajectories in baseball, Marchi and Albert suggested using a linear model of the form:

$$y = A + B(Age - 30) + C(Age - 30)^2$$

That may sound a little complicated, but that's statisticians for you - hiding simple, yet powerful, ideas amidst arcane terminology!;-)

Let's break down that equation a little. It says that championship position (*y*) can be modelled as a mathematical function of the age of the driver. We actually use the *(Age - 30)* in the equation so that the value *A* is a prediction of the championship position for the driver aged 30.

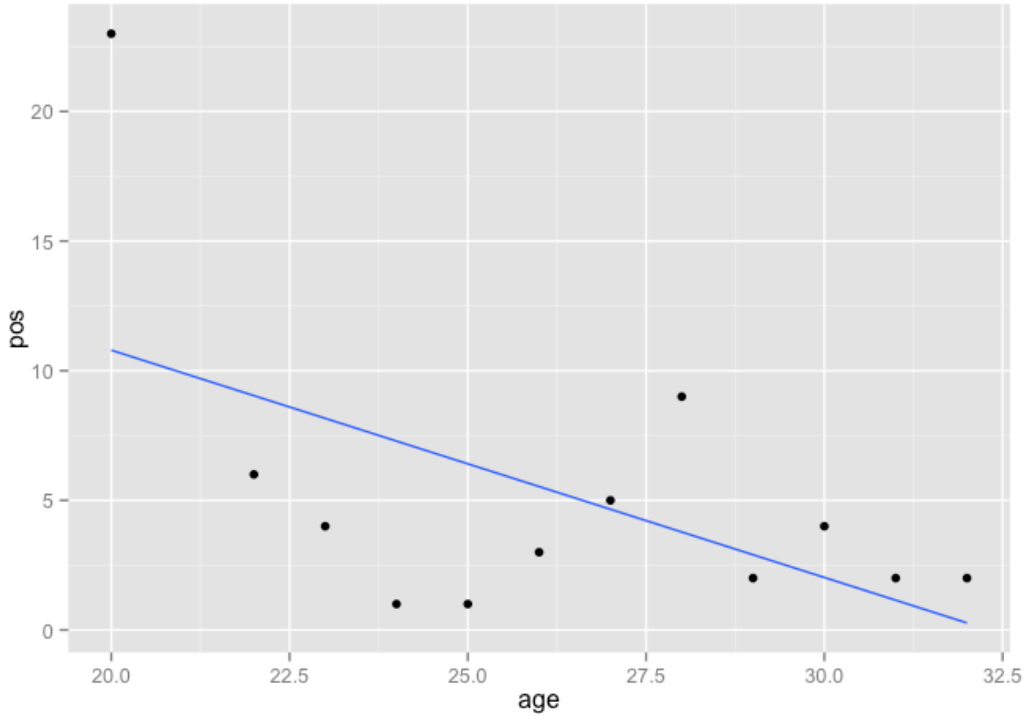
The equation itself defines a best fit curve through the data. The values of *A*, *B* and *C* are chosen so to minimise the difference or distance (also referred to as error, or residual value) between each data point and the line.

Before looking at how well that line fits the data, let's consider a simpler case:

$$y = A + B(Age - 30)$$

This sort of equation defines a straight line as the line of best fit:

```
g=ggplot(drivercareer,aes(x=age,y=pos))
g=g+stat_smooth(method = "lm", formula = y ~ I(x-30) , se=FALSE)
g+geom_point()
```

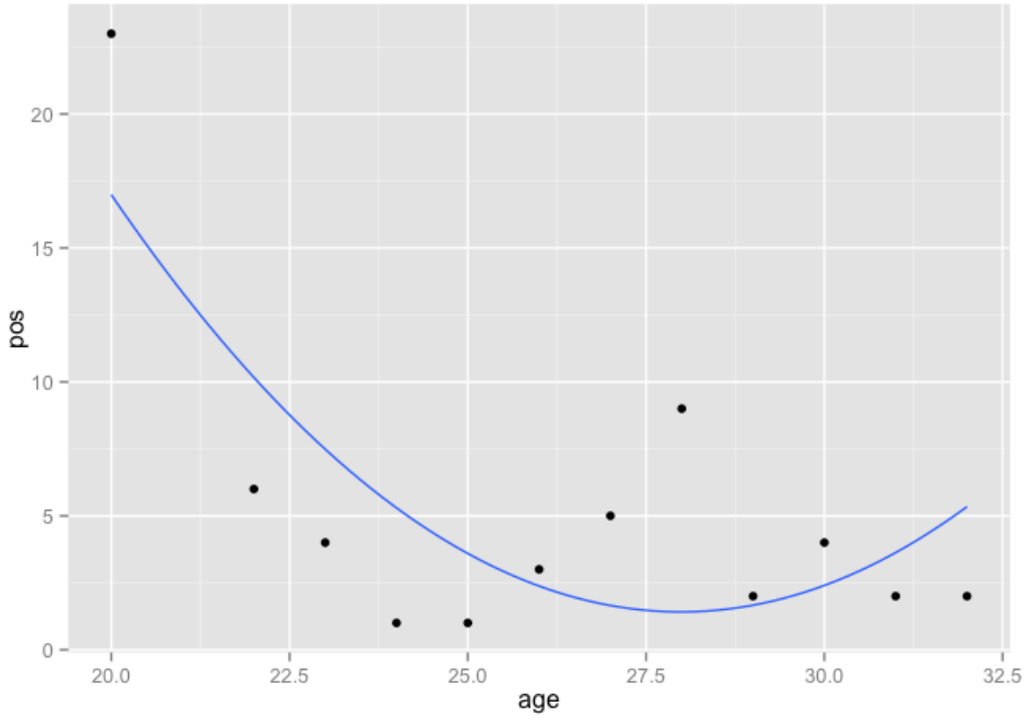


Fernando Alonso's championship positions versus his age in years, with a straight line best fit model

The problem with this sort of line is that it it is a straight line, suggesting that a career can only go in one direction.

In the model Marchi and Albert suggest, the squared ("quadratic") term means we expect a curved line with a single hump in it; this allows for careers to go up and then down, or down and then up. We might also fit such a curve so that it only goes up, or only goes down within particular limits. Let's see how well a curve with that sort of shape fits our data:

```
g=ggplot(drivercareer,aes(x=age,y=pos))
g=g+stat_smooth(method = "lm", formula = y ~ I(x-30) +I( (x-30)^2 ), se=FALSE)
g+geom_point()
```

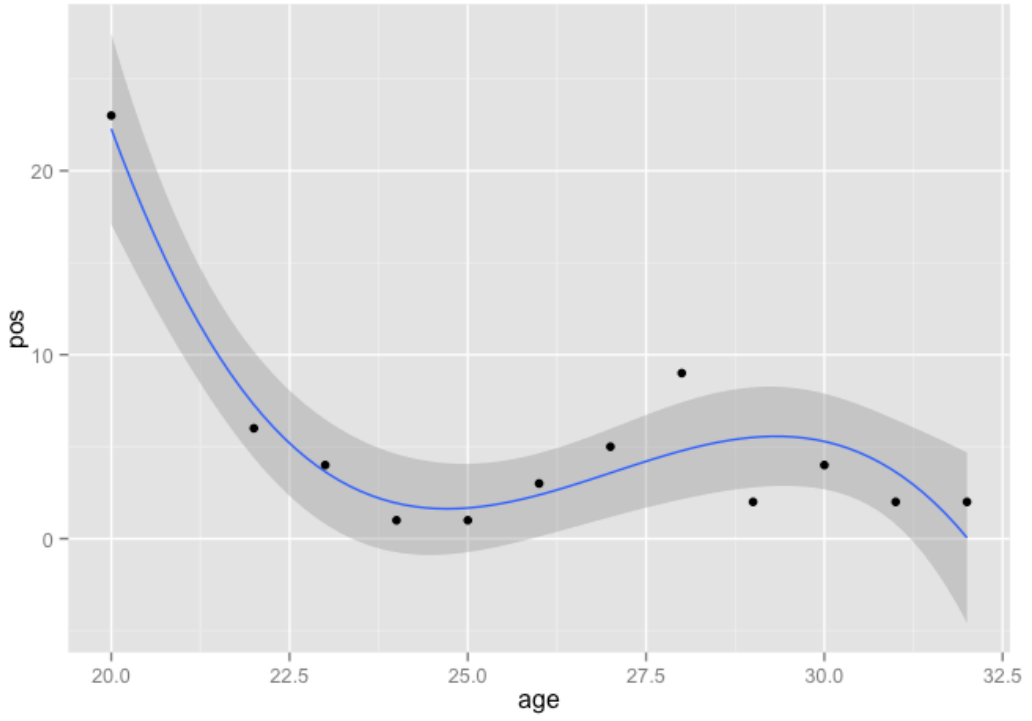


Fernando Alonso's championship positions versus his age in years modelled as described by Marchi and Alvert

We can add increasing numbers of terms to the model, but the effect of these higher powered terms is often negligible. For example, if we ad a cubic term (power 3), we allow the line to have a couple of wiggles (that is, changes of direction).

We can also add confidence limits to the line (by default) to show how confident we are that a point is modelled by the line, subject to some amount of random variation in the values of the actual data points.

```
g=ggplot(drivercareer,aes(x=age,y=pos))
g=g+stat_smooth(method = "lm", formula = y ~ I(x-30) +I( (x-30)^2 ) + I( (x-30)^3 ) )
g+geom_point()
```



Fernando Alonso's championship positions versus his age in years with a cubic best fit line

Let's go back to Marchi and Albert's model:

$$y = A + B(Age - 30) + C(Age - 30)^2$$

As well as using the graphical approach to see how well this model fit, we can run some numbers.

```
lm(pos ~I(age-30) + I( (age-30)^2 ), data=drivercareer)
```

Call: lm(formula = pos ~ I(age - 30) + I((age - 30)^2), data = drivercareer)

Coefficients: (Intercept) I(age - 30) I((age - 30)^2)
2.4003 0.9820 0.2441

The *(Intercept)* value is the coefficient that corresponds to the value of A in the model - that is, the expected championship position for that driver aged 30. The other two values

(corresponding to *B* and *C* in the equation) are harder to decipher, although C does indicate how "curved" the line is (that is, how quickly the driver reaches his peak, and then falls from it (or *vice versa*)).

However, we can do some sums with them. In fact, we can do some *calculus* with them to find the acge at which the model suggests the driver is supposed to be performing at their peak (or, if the curve is the other way up, their low point).

We can find this point by locating the point on the graph at which it changes direction (a so called *stationary point*). This point is the point at which the gradient is zero. We can find the gradient of the line by differentiating it (which is where the calculus comes in).

$$\frac{dy}{dAge} = B + 2C(Age - 30)$$

The gradient is zero when $\frac{dy}{dAge} = 0$. Rearranging, we get:

$$-B = 2C(Age_{peak} - 30)$$

then:

$$\frac{-B}{2C} = Age_{peak} - 30$$

and hence:

$$Age_{peak} = 30 - \frac{B}{2C}$$

which is the age at which we expect the performance to be best (or worst).

We can also predict expected championship position at this age as:

$$y_{peak} = A + B(Age_{peak} - 30) + C(Age_{peak} - 30)^2$$

which is to say:

$$y_{peak} = A + B(30 - \frac{B}{2C} - 30) + C(Age_{peak} - 30)^2$$

This comes out as:

$$y_{peak} = A - \frac{B^2}{2C} + \frac{CB^2}{4C^2}$$

which after a bit of cancelling and subtraction gives:

$$y_{peak} = A - \frac{B^2}{4C}$$

Marchi and Albert suggest the following function to capture these values:

```
fit.model= function(d){
  fit = lm(pos ~I(age-30) +I((age-30)^2),data=d)
  b=coef(fit)
  age.max=30-b[2]/b[3]/2
  y.peak=b[1]-b[2]^2/b[3]/4
  list(fit=fit,age.max=age.max,y.peak=y.peak)
}

fit.model(drivercareer)
```

$fit

Call: lm(formula = pos ~ I(age - 30) + I((age - 30)^2), data = d)

Coefficients: (Intercept) I(age - 30) I((age - 30)^2)
2.4003 0.9820 0.2441

$age.max I(age - 30) 27.9884

$y.peak (Intercept) 1.412555

Confidence limits

TO DO - a note on reading confidence limits

For a more robust, which is to say, general, model, we might take the data from a large number of drivers who have had several years experience in F1 and see what that tells us about career profiles.

We can do this in two ways - one to use the age in years, the other to use the number of years in F1.

So how can we find longlasting drivers, and how can we find their final rank in the championship for each year they competed?

In our local *ergast* database, the *driverStandings* table includes the championship position at the end of every race. We can find the championship position at the end of a season by looking up the final race of the season (the one with the highest *round* number in that year). This also gives us a *raceId* which we can use to look up drivers' standings.

We can then select driver standings from the last rounds of each year, given the *raceId* of those rounds and filter the results based on the number of final rounds the driver has a standing in. To plot the career chart, let's get data for drivers who competed in at least 10 seasons.

```
longstanding=dbGetQuery(con_ergastdb,
                        'SELECT d.driverId, d.driverRef, d.dob, COUNT(*) years
                        FROM driverStandings ds JOIN drivers d
                        WHERE raceId IN (SELECT raceId FROM lastRounds)
                        AND d.driverId=ds.driverId
                        GROUP BY ds.driverId
                        HAVING years>=10
                        ORDER BY years DESC')

kable( head(longstanding,n=5) )
```

driverId	driverRef	dob	years
22	barrichello	1972-05-23	19
30	michael_schumacher	1969-01-03	19
289	hill	1929-02-15	18
119	patrese	1957-04-17	17
347	bonnier	1930-01-31	16

Lets create a temporary table - *firstchampionship* - that shows the year in which a driver first competed.

```
dbGetQuery(con_ergastdb,
           'CREATE TEMPORARY VIEW firstchampionship AS
           SELECT ds.driverId, driverRef, dob, MIN(year) AS firstYear
           FROM driverStandings ds JOIN races r JOIN drivers d
           WHERE r.raceId=ds.raceId
           AND d.driverId=ds.driverId
           GROUP BY ds.driverId')
```

```
kdb('SELECT * FROM firstchampionship LIMIT 3')
```

driverId	driverRef	dob	firstYear
1	hamilton	1985-01-07	2007
2	heidfeld	1977-05-10	2000
3	rosberg	1985-06-27	2006

We can then modify the *longstanding* query to include year in which a driver first competed.

```
dbGetQuery(con_ergastdb,
           'CREATE TEMPORARY VIEW longstanding AS
           SELECT d.driverId, d.driverRef, d.dob, firstYear, COUNT(*) years
           FROM driverStandings ds JOIN drivers d JOIN firstchampionship fc
           WHERE raceId IN (SELECT raceId FROM lastRounds)
           AND d.driverId=ds.driverId
           AND d.driverId=fc.driverId
           GROUP BY ds.driverId
           HAVING years>=10
           ORDER BY years DESC')
```

```
kdb('SELECT * FROM longstanding LIMIT 3')
```

driverId	driverRef	dob	firstYear	years
22	barrichello	1972-05-23	1993	19
30	michael_schumacher	1969-01-03	1991	19
289	hill	1929-02-15	1958	18

?should really normalise the points by the number of starts?

The Age-Productivity Gradient

The relationship between age and performance is also explored in *The age-productivity gradient: evidence from a sample of F1 drivers* by Fabrizio Castellucci, Giovanni Pica, Mario Padula, Labour Economics 18.4 (2011): 464-473 (also available as Caâ€™ Foscari University of Venice, Department of Economics, Working Paper No. 16/WP/2009[29]).

To do: replicate elements of this paper

Summary

In this chapter, we have started to explore something of the relationship between a driver's age and his performace. In updates to this chapter, and additional chapters, we will explore additional models, as well as considering performance related to their "F1 age" - the number of years a driver has spent in F1 - not just their physical age.

[29] http://www1.unive.it/media/allegato/DIP/Economia/Working_papers/Working_papers_2009/WP_DSE_castellucci_pica_padula_16_09.pdf

Streakiness

*This chapter was originally inspired by the **Exploring Streaky Performances** chapter of **Analyzing Baseball Data with R**[30] (2013) by Max Marchi & Jim Albert. See also: Albert, Jim. "Streaky hitting in baseball." Journal of Quantitative Analysis in Sports 4.1 (2008).*

In the search for "interesting things to talk about", references to winning streaks or other streaks in performance are a trusty standby of many sports commentators and writers. In Formula One, there are plenty of options to consider: from runs in which a particular driver starts from pole position, or finishes with a win or a podium place, to streaks in consecutives races with a points finish or even just race completions. We can also look to team performances - the number of consecutive races where a team locked out the front row of the grid, perhaps, or the number of races in a row where a team finished with both drivers on the podium.

The *ergast* database contains the data we need to identify these sorts of behaviour, though as you'll see, we typically need to process it a little first.

```
library(DBI)
ergastdb =dbConnect(RSQLite::SQLite(), './ergastdb13.sqlite')

#Helper function to display database query result as a formatted table
kdb=function(q){ kable(dbGetQuery(ergastdb,q)) }
```

Let's start by grabbing some data for a particular driver:

[30] http://baseballwithr.wordpress.com/about/

```
getDriver=function(driverRef,year=''){
  if (year!='')
    year_q=paste(' AND year in (',
                 paste(c(year),collapse=','),
                 ')',sep='')
  else year_q=' '
  dbGetQuery(ergastdb,
             paste('SELECT year, round, c.name, grid, position, points
                   FROM results r JOIN drivers d JOIN races rc JOIN circuits c
                   WHERE driverRef="',driverRef,'"
                   AND d.driverId=r.driverId
                   AND r.raceId=rc.raceId
                   AND rc.circuitId=c.circuitId
                   ',year_q,'
                   ORDER BY year, round',
                   sep=''))
}

alonso=getDriver('alonso')
#We can also query for a driver's race positions for a single year
#getDriver('alonso',2013)
#Or set of years
#getDriver('alonso', c(2012,2013))
kable(head(alonso,n=3),format='markdown')
```

year	round	name	grid	position	points
2001	1	Albert Park Grand Prix Circuit	19	12	0
2001	2	Sepang International Circuit	21	13	0
2001	3	Autódromo José Carlos Pace	19	NA	0

There are several columns we need to take an interest in: the *year* and *round* columns will be used to sort the races so we can look for streaks or runs of behaviour across consecutive races, and even from one season to the next. The *position* and *grid* columns contain data relevant to particular events, such as front row starts or podium finishes.

Performance Streaks

What performance streaks do you think might be of interest to an F1 stats fan, or perhaps a pub quiz questionmaster? I've given some crude examples in the opening pargraph above, but can you think of some rather more well defined questions, and how you might set about trying to answer them?

One typical sort of question might relate to a particular driver or team, such "what is the longest run of podium finishes that Fernando Alonso has ever had in F1 in a particular season?". This requires us to look for streaks within a particular year, but also for every year in which the driver competed.

Another typical question might be "who has had the longest run of wins, and how many was it?" This question actually requires some refinement: does it mean within a particular season, or does in mean over consecutive races more generally, rolling over from one season to the next?

Questions might also relate to runs that span seasons: "which team has had the longest run of podiums over consecutive seasons at Silverstone?", for example. Or they may add an additional constraint, by requiring that runs take place in a particular part of a season: who has has the most consecutive wins at the start of a season, for example, or at the end of a season?

Whilst not necessarily *useful* questions - and some might even argue as to whether they even count as *interesting* ones! - trying to answer questions such as these provides us with a data-related recreational activity that lets us polish our data wrangling skills in a harmless way, if nothing else!

Spotting Runs

The function used by Marchi & Albert to detect streaks takes a list of 1s and 0s that describe the outcome of a particular event we want to detect streaky behaviour in, and then calculates how long each run of 1s is, listing them as the result.

The function I am going to use counts runs of both 1s and 0s, distinguishing which is which by use of numerical sign (plus for runs of ones, minus for runs of zeroes). The function also returns the row numbers in the original dataset that correspond to the start and end of the run.

The following functions generate result flags that we can use detect different sorts of feature - front row of the grid start, or podium finish, for example - and hence different sorts of run.

```
podium=function(pos) {if (!is.na(pos) & pos<4) 1 else 0}
frontrow=function(pos) {if (!is.na(pos) & pos<=2) 1 else 0}
topNfinish=function(pos,N) {if (!is.na(pos) & pos<=N) 1 else 0}
unclassified=function(pos) {if (is.na(pos)) 1 else 0}
inpoints=function(points) {if (points>0) 1 else 0}
```

We can apply these functions as follows:

```
alonso$podium=mapply(podium,alonso$position)
alonso$frontrow=mapply(frontrow,alonso$grid)
alonso$top5=mapply(topNfinish,alonso$position,5)

kable(tail(alonso,n=4),row.names = FALSE)
```

year	round	name	grid	position	points	podium	frontrow	top5
2013	16	Buddh International Circuit	8	11	0	0	0	0
2013	17	Yas Marina Circuit	10	5	10	0	0	1
2013	18	Circuit of the Americas	6	5	10	0	0	1
2013	19	Autódromo José Carlos Pace	3	3	15	1	0	1

As you can see from the above table, the *podium* flag is set to one when the driver is on the podium, the *topN* flag denotes whether the driver finished with the top however many drivers, and so on.

To detect runs, we generate a list of results ordered by increasing year and round and look for sequences of 1s or 0s, counting the number of consecutive 1s (or 0s) in a row, and also identify (using a positive or negative sign) whether a run applied to a streak of 1s or 0s.

```
#y contains a list of result flags
#val defines polarity
streaks=function(y,val=0){
  #Start by initialising run length to 0
  run=0
  #Define a list to capture all the runs, in sequence
  runs=c()
  #Initialise a variable that contains the previous result
  prev=y[1]
  #The last flag identifies the last result as part of a run
  last=TRUE
  #Search through each result flag
  for (i in y) {
    #Is the current result is the same as the previous one?
    if (i!=prev) {
      #If not, record the length of the previous run, and its polarity
      runs=c(runs,run*(if (prev==val) -1 else 1))
      #Initialise the next run length
      run=0
      #This result is the first in a new run
      last=FALSE
    } else {
      #We are still in a run
      last=TRUE
    }
    #Keep track of what the previous result flag was
    prev=i
    #Increase the length of the run counter
    run=run+1
  }
  #If the last result was part of a run, record that run
  if  (last | (run==1)) runs=c(runs,run*(if (prev==val) -1 else 1))
  #Create a dataframe from run list
  ss=data.frame(l=runs)
  #Tally how many results in total have been counted after each run
  #That is, record the result number for the last result in each run
  ss$end=cumsum(abs(ss$l))
  #Identify the result number for the start of each run
  ss$start=ss$end-abs(ss$l)+1
  #Reorder the columns
  ss[,c("start","end","l")]
}
```

Let's see whether there were any notable streaks of top5 placements in Alonso's career to the end of 2013:

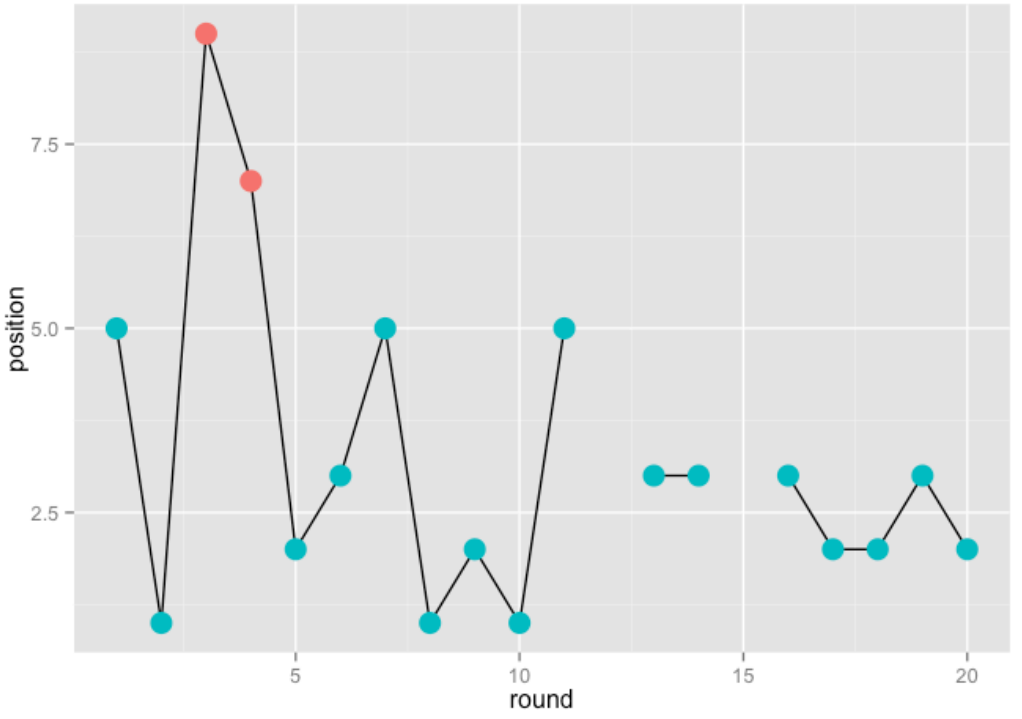
```
alonso=getDriver('alonso',2012)
alonso$top5=mapply(topNfinish,alonso$position,5)

kable( streaks(alonso$top5) )
```

start	end	l
1	2	2
3	4	-2
5	11	7
12	12	-1
13	14	2
15	15	-1
16	20	5

This suggests that Alonso finished in the top 5 for the first two races of the season, out of the top 5 for the next two, in the top 5 for the 7 races from round 5 to round 11 inclusive, then patchy runs until a set of 5 good finishes at the end of the season.

Let's see if we can confirm it with a plot of his finishing positions:

```
library(ggplot2)
g=ggplot(alonso,aes(x=round,y=position))+geom_line()
g+geom_point(aes(colour=factor(top5)), size=5,show_guide = FALSE)
```



Alonso's finishing positions for each round of 2012

That looks about right:-)

Generating Streak Reports

The following function packages up the *streaks* function with some logic that applies the appropriate results flag function to generate a streaks dataframe. It then annotates this dataframe with the information about the first and last races in the streak (by race name and year) as well as the name of the race that broke the streak (that is, the race that immediately follows the end of the streak).

```
streakReview=function(driverRef,years='',length=5,typ=2,mode='podium',topN=''){
  #The definitions are a bit messy...
  #If we set a topN parameter, force the topN mode
  if (topN!='') mode='topN'

  #Get the driver details
  driver=getDriver(driverRef,years)

  #Calculate the desired 0/1 stream based on the mode property
  if (mode=='podium') driver$mode=mapply(podium,driver$position)
  else if (mode=='frontrow') driver$mode=mapply(frontrow,driver$grid)
  else if (mode=='topN') driver$mode=mapply(topNfinish,driver$position,topN)
  else if (mode=='unclassified') driver$mode=mapply(unclassified,driver$position)
  else if (mode=='inpoints') driver$mode=mapply(inpoints,driver$points)
  else return(data.frame())

  #Calculate the streaks in the desired mode property
  streak=streaks(driver$mode)

  #Annotate the streak with start (first), end (last) and broken-by information
  streak$startc= mapply(function(x) driver[x,'name'],streak$start)
  streak$endc= mapply(function(x) driver[x,'name'],streak$end)
  streak$starty= mapply(function(x) driver[x,'year'],streak$start)
  streak$endy= mapply(function(x) driver[x,'year'],streak$end)
  nd=nrow(driver)
  streak$brokenbyy=mapply(function(x) if (nd<x+1) NA else driver[x+1,'year'],
                          streak$end)
  streak$brokenbyc= mapply(function(x) if (nd<x+1) NA else driver[x+1,'name'],
                           streak$end)

  #The typ argument lets us get all streaks, 1s streaks, or 0s streaks
  #greater than or equal to a specified length
  if (typ==2) streak[abs(streak["l"])>=length,]
  else if (typ==1) streak[abs(streak["l"])>=length & streak["l"]>0,]
  else streak[abs(streak["l"])>=length & streak["l"]<0,]
}
alonso=streakReview("alonso",length=1)
kable( head(alonso,n=5) )
```

start	end	l	startc	endc	starty	endy	brokenbyy	brokenbyc
1	18	-18	Albert Park Grand Prix Circuit	Albert Park Grand Prix Circuit	2001	2003	2003	Sepang International Circuit
19	20	2	Sepang International Circuit	Autódromo José Carlos Pace	2003	2003	2003	Autodromo Enzo e Dino Ferrari
21	21	-1	Autodromo Enzo e Dino Ferrari	Autodromo Enzo e Dino Ferrari	2003	2003	2003	Circuit de Catalunya
22	22	1	Circuit de Catalunya	Circuit de Catalunya	2003	2003	2003	A1-Ring
23	29	-7	A1-Ring	Hockenheimring	2003	2003	2003	Hungaroring

The following table shows long streak runs of podium finishes and periods without making the podium for Fernando Alonso:

```
kable( head(alonso[order(-abs(alonso$l)),]) )
```

	start	end	l	startc	endc	starty	endy	brokenbyy	brokenbyc
1	1	18	-18	Albert Park Grand Prix Circuit	Albert Park Grand Prix Circuit	2001	2003	2003	Sepang International Circuit
20	65	79	15	Istanbul Park	Circuit Gilles Villeneuve	2005	2006	2006	Indianapolis Motor Speedway
37	106	119	-14	Albert Park Grand Prix Circuit	Autodromo Nazionale di Monza	2008	2008	2008	Marina Bay Street Circuit
41	124	136	-13	Albert Park Grand Prix Circuit	Autodromo Nazionale di Monza	2009	2009	2009	Marina Bay Street Circuit
9	35	42	-8	Sepang International Circuit	Indianapolis Motor Speedway	2004	2004	2004	Circuit de Nevers Magny-Cours
5	23	29	-7	A1-Ring	Hockenheimring	2003	2003	2003	Hungaroring

So for example, for 15 consecutive races starting with Istanbul Park in 2005 up until, and including, the Circuit Gilles Villeneuve in 2006, Alonso made the podium; the run was broken when he finished out of the top 3 at the 2006 Indianapolis Motor Speedway. He's also suffered lean periods, such as the first 14 races of races of 2008 or the first 13 races of 2009, both of which were broken by podium finishes at the Marina Bay Street Circuit.

We can check that the code is generating streaks correctly by comparing the results it gives with articles such as The greatest winning streaks in F1 history[31] or Wikipedia's list of Formula One driver records[32].

Fact Checking News Reports

In the run up to the 2014 United States Grand Prix at the Circuit of the Americas, the Guardian newspaper reported how Lewis Hamilton "is attempting to become the first British driver since Nigel Mansell in 1992 to win five races in a row" ().

News fragments such as provide a good opportunity to try out, and extend, our data wrangling approaches. In this case, how would generate a report that lists all in-season winning streaks of length five or more driver and nationality?

Also see if you can generate a query that allows you to state a driver's nationality and minimum winning streak length, and get back a list of drivers of that meet those conditions.

Hint condition

One approach I considered was to get a list of *driverId* values for drivers who had had a minimum number of wins in a particular season and then use these values as the basis for streakiness searches. {lang="R"} ~~~~~~~~~~~~~~~~~~ multiwinners.gb = dbGetQuery(ergastdb,'SELECT driverRef, d.driverId, nationality, MAX(wins), year FROM driverStandings ds JOIN races r JOIN drivers d WHERE ds.raceId=r.raceId AND ds.driverId=d.driverId AND ds.driverId IN (SELECT DISTINCT driverId FROM drivers WHERE nationality="British") GROUP by year,d.driverId HAVING MAX(wins)>=5') ~~~~~~~~~~~~~~~~~~

We can then use something like *ddply()* to call the *streakReview()* function using the *driverRef* and *year* values (e.g. *ddply(multiwinners.gb,.(driverRef,year),function(x) streakReview(x$driverRef,length=5,topN=1,years=x$year,typ=1))*)

[31] http://www.enterf1.com/blog/166-the-greatest-winning-streaks-in-f1-history
[32] http://en.wikipedia.org/wiki/List_of_Formula_One_driver_records

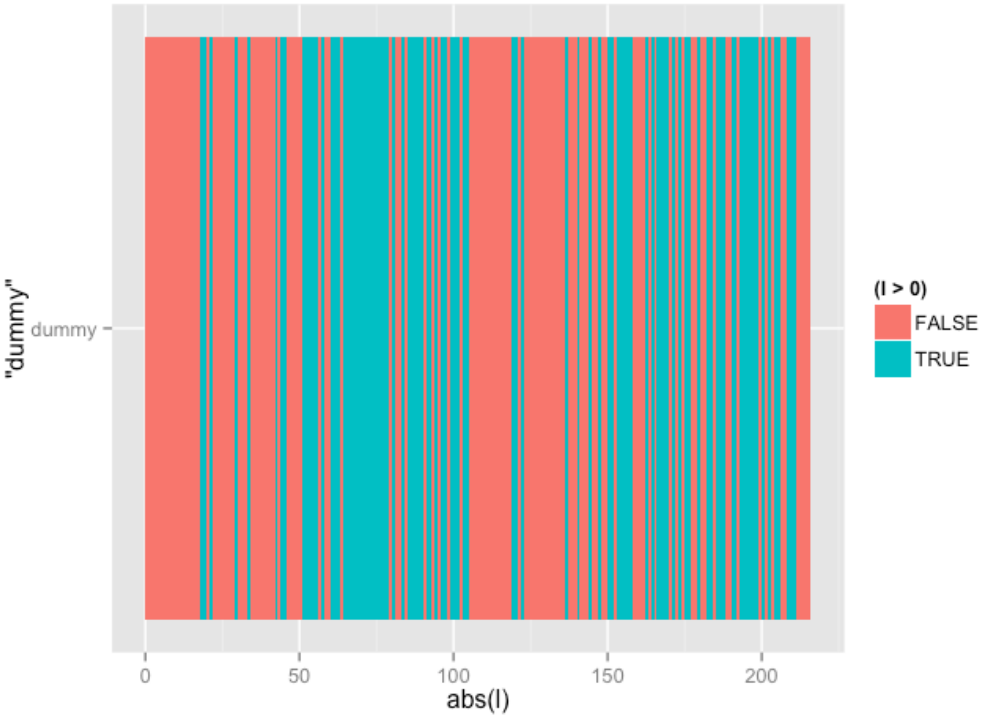
Streak Maps

Having generated the streaks data, how might we visualise it? One way is to use a horizontal *stacked bar chart* with the following properties:

- the stacked elements correspond to consecutive runs;
- the size of each stacked element is proportional to the length of the run;
- the fill colour of each element reflects the state of the results flag used to calculate the run.

We l value provides all the information we need: its magnitude helps set the element size, and its sign the fill colour.

```
ggplot(alonso,aes(x="dummy",y=abs(l),fill=(l>0)))+geom_bar(stat='identity')+coord_fli\
p()
```

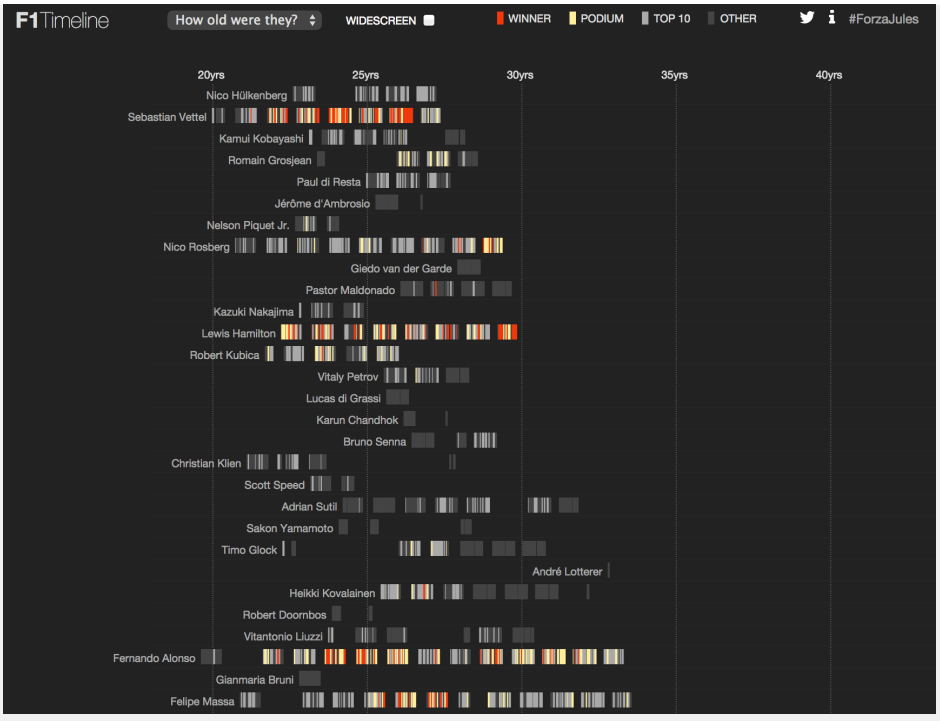


Visualising Alonso's podium streaks with a horizontal filled bar chart

One problem with this chart is that it does not show season breaks, or the extent to which races were back to back races or races with a break in-between. To depict such information, we would need to have an x-axis that is more representative of time, for example basing it on year and week number in which each race took place.

F1 Timeline - Peter Cook

In a striking visualisation built on top of the *ergast* database, Peter Cook's interactive *F1 Timeline* provides a view over the race positions achieved by F1 drivers according to their age, with the drivers also ordered reverse chronologically in terms of when they first competed in F1.



Peter Cook's *F1 Timeline* - http://charts.animateddata.co.uk/f1/

Peter Cook's notes also describe the design philosophy and implementation details behind the visualisation.

Creating your own F1 Timeline

In his F1 Timeline: Design[33] notes, Peter Cook describes an initial sketch using a simple scatterplot in which "*[e]ach row represents a driver and [a] circle a race result. Time progresses from left to right. Black indicates a win, dark grey a podium etc.*"

Driver ages are calculated from dates of birth recorded in the *ergast* drivers table, and (where values are missing) other sources. Using the *ergast* data, how would you create a version of the chart using *driver age* on the x-axis?

If you're feeling even more adventurous, try to create your own static version of the *F1 Timeline* for drivers who competed in the 2013 season.

[33] http://animateddata.co.uk/articles/f1-timeline-design/

Team Streaks

Searching for streaks in perdformance around a single thing, such as a particular driver or team, is one thing. But can we also search for streaks based on the performance of both members of a team? Could we, for example, identify races in which a team got a one-two on the podium (that is, taking first and second positions in the same race), or locked out the front row of the grid?

One-Two and Double Podium Streaks

To find out if two drivers of the same team are both in the top 2 positions we need to identify races where there are two different members of the same team in the top 2 positions of the same race. One way of thinking about this is to combine *two* copies of the results table, each providing the position for one of the drivers. By JOINing the tables appropriately - making sure the *driverId*s returned from each copy of the results table are different but their *constructorId* and *raceId* are the same - we can pull back results where one team member is in first position in a particular race and the other is in second. Finally, we join in the race table so that we can access the race name, year and round.

```
dbGetQuery(ergastdb,
                  'SELECT r.name,r.year,round,constructorRef
                  FROM results r1 JOIN results r2 JOIN races r JOIN constructors c
                  WHERE r1.constructorId=r2.constructorId
                  AND r1.driverId!=r2.driverId
                  AND r1.raceId=r2.raceId
                  AND r1.position=1 AND r2.position=2
                  AND r.raceId=r1.raceId
                  AND c.constructorId=r1.constructorId
                  ORDER BY r.year DESC, round DESC
                  LIMIT 5')
```

```
##                   name year round constructorRef
## 1 Brazilian Grand Prix 2013    19       red_bull
## 2 Abu Dhabi Grand Prix 2013    17       red_bull
## 3  Japanese Grand Prix 2013    15       red_bull
## 4 Malaysian Grand Prix 2013     2       red_bull
## 5    Korean Grand Prix 2012    16       red_bull
```

Another, more efficient approach, is to do some counting. For example, we can group the results by team for each race, and then count the number of drivers who finshed in the team in the top two, discounting drivers who were not placed. If we count two drivers with non-null positions in the top two places, we know we've got a one-two. If we bring in the constructor table, we can pull in the constructor reference too.

```
onetwo=dbGetQuery(ergastdb,
                  'SELECT r.name,r.year,round,constructorRef
                  FROM results res JOIN races r JOIN constructors c
                  WHERE r.raceId=res.raceId
                  AND c.constructorId=res.constructorId
                  AND res.position NOT NULL AND res.position<3
                  GROUP BY res.raceId,res.constructorid
                  HAVING COUNT(res.driverId)=2
                  ORDER BY r.year DESC, round DESC')
kable(head(onetwo))
```

name	year	round	constructorRef
Brazilian Grand Prix	2013	19	red_bull
Abu Dhabi Grand Prix	2013	17	red_bull
Japanese Grand Prix	2013	15	red_bull
Malaysian Grand Prix	2013	2	red_bull
Korean Grand Prix	2012	16	red_bull
Brazilian Grand Prix	2011	19	red_bull

This query returns details of the races where a team got a one-two. To calculate one-two streaks, we can again use the results flag approach using a conditional *CASE* element in the SELECT part of the query. This statement sets the flag value to 1 if neither position in the team is NULL (the count of position values is 2) and both positions are less that or equal to 2nd place, otherwise it sets the flag to 0.

```
onetwoFlag=dbGetQuery(ergastdb,
           'SELECT r.name,r.year,round,constructorRef,
           CASE WHEN (MAX(res.position)<=2 AND COUNT(res.position)=2) THEN 1 ELSE 0 E\
ND AS onetwo
           FROM results res JOIN races r JOIN constructors c
           WHERE r.raceId=res.raceId
           AND c.constructorId=res.constructorId
           AND constructorRef="red_bull"
           GROUP BY res.raceId,res.constructorid
           ORDER BY r.year DESC, round DESC')

kable(head(onetwoFlag))
```

name	year	round	constructorRef	onetwo
Brazilian Grand Prix	2013	19	red_bull	1
United States Grand Prix	2013	18	red_bull	0
Abu Dhabi Grand Prix	2013	17	red_bull	1
Indian Grand Prix	2013	16	red_bull	0
Japanese Grand Prix	2013	15	red_bull	1
Korean Grand Prix	2013	14	red_bull	0

We can tweak the previous query to find races where a team has a double podium (i.e. both drivers finish on the podium):

```
doublepodiumFlag = dbGetQuery(ergastdb,
           'SELECT DISTINCT r.name, year, round, constructorRef,
              CASE WHEN (COUNT(res.position)=2 and MAX(res.position)<=3)
                THEN 1 ELSE 0 END AS doublePodium
           FROM results res JOIN races r JOIN constructors c
           WHERE r.raceId=res.raceId
           AND c.constructorId=res.constructorId
           AND year=2009 AND constructorRef="red_bull"
           GROUP BY res.raceId,res.constructorid
           ORDER BY year DESC, round DESC')
kable(head(doublepodiumFlag,n=6))
```

name	year	round	constructorRef	doublePodium
Abu Dhabi Grand Prix	2009	17	red_bull	1
Brazilian Grand Prix	2009	16	red_bull	0
Japanese Grand Prix	2009	15	red_bull	0
Singapore Grand Prix	2009	14	red_bull	0
Italian Grand Prix	2009	13	red_bull	0
Belgian Grand Prix	2009	12	red_bull	0

It is straightforward enough to generalise this query to identify when both drivers are in the top N positions, whether both drivers are classified - or neither are - and so on.

Grid lockouts

Another team run we can look out for are front row lockouts on the grid, where both of a team's cars are on the front row (that is, between them the team has taken the first and second grid positions).

```
lockout = dbGetQuery(ergastdb,
                     'SELECT r.name,year,round,constructorRef
                     FROM results res JOIN races r JOIN constructors c
                     WHERE r.raceId=res.raceId
                     AND res.grid<3
                     AND c.constructorId=res.constructorId
                     GROUP BY res.raceId,res.constructorid
                     HAVING COUNT(res.driverId)=2
                     ORDER BY year DESC, round DESC')
kable(head(lockout,n=5))
```

name	year	round	constructorRef
United States Grand Prix	2013	18	red_bull
Abu Dhabi Grand Prix	2013	17	red_bull
Japanese Grand Prix	2013	15	red_bull
Italian Grand Prix	2013	12	red_bull
British Grand Prix	2013	8	mercedes

Again, we can modify this query in order to list all races but flag those in which a team locks out the front row of the grid.

```
lockoutFlag= dbGetQuery(ergastdb,
                        'SELECT DISTINCT r.name,year,round, constructorRef,
                            CASE WHEN (MAX(res.grid)=2 AND COUNT(res.position)=2)
                              THEN 1 ELSE 0 END AS gridLockout
                        FROM results res JOIN races r JOIN constructors c
                        WHERE r.raceId=res.raceId
                        AND c.constructorId=res.constructorId
                        AND constructorRef="red_bull"
                        GROUP BY res.raceId,res.constructorid
                        ORDER BY year DESC, round DESC')
kable(head(lockoutFlag))
```

name	year	round	constructorRef	gridLockout
Brazilian Grand Prix	2013	19	red_bull	0
United States Grand Prix	2013	18	red_bull	1
Abu Dhabi Grand Prix	2013	17	red_bull	1
Indian Grand Prix	2013	16	red_bull	0
Japanese Grand Prix	2013	15	red_bull	1
Korean Grand Prix	2013	14	red_bull	0

Time to N'th Win

At the start of a driver's Formula One career, they are likely to have a run of finishes that don't include taking the winner's trophy. Such a run of not-first-place finishes also identifies the time to the first win.

However, to calculculate the time to the first win, we donlt necessarily need to calculate the length of an intitial not-winning streak. For example, the first win can be identified quite straightforwardly using a query of the form:

```
dbGetQuery(ergastdb,
           'SELECT driverRef, code, dob, year, round, r.name, c.name, date
           FROM drivers d JOIN results rs JOIN races r JOIN circuits c
           WHERE rs.driverId=d.driverId
           AND rs.position=1 AND r.raceId=rs.raceId
           AND c.circuitId=r.circuitId
           AND driverRef="alonso"
           ORDER BY date LIMIT 1')
```

```
##   driverRef code        dob year round                name        name
## 1    alonso  ALO 1981-07-29 2003    13 Hungarian Grand Prix Hungaroring
##         date
## 1 2003-08-24
```

We can tweak this query to give us the *N*'th win by adding an offset equal to *N-1*. So for example, we can find Alonso's fifth win as follows:

```
dbGetQuery(ergastdb,
           'SELECT driverRef, code, dob, year, round, r.name, c.name, date
           FROM drivers d JOIN results rs JOIN races r JOIN circuits c
           WHERE rs.driverId=d.driverId
           AND rs.position=1 AND r.raceId=rs.raceId
           AND c.circuitId=r.circuitId
           AND driverRef="alonso"
           ORDER BY date
           LIMIT 1 OFFSET 4')
```

```
##   driverRef code        dob year round               name        name
## 1    alonso  ALO 1981-07-29 2005     7 European Grand Prix Nürburgring
##         date
## 1 2005-05-29
```

But how would you find the number of races a driver had competed in before his first win? Or N'th win?

TO DO

Examples of discovery of streaky runs in individual and team performances

Finding Streaky Runs

Using the datawrangling strategies developed above, or others of your own devising, explore the *ergast* dataset to find examples of runs and streaks in driver and team performance.

Summary

We have seen one way of calculating streaky behaviour in a driver's or team's performance by setting a results flag that identifes some performance feature (such as finishing first, or finishing on pole, or both cars in a team finishing on the podium) and then counting consecutive appearances of the same value of the flag.

What we haven't done are any statistical tests to see whether these runs are "unlikely" in any particular sense of the word, or might be expected by chance given the results profile of the driver of his compelete career or the team over several seasons. The interested reader who would like to explore such statistical comparisons further might find the works by Albert, and Marchi & Albert, that were referred to at the start of this chapter provide a useful starting point for such an investigation.