

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 is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.



This work is licensed under a Creative Commons Attribution 3.0 Unported License

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>

Also By Tony Hirst

Course Analytics - Wrangling FutureLearn Data With Python and R

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) required the following changes to be made to earlier editions of this book:

```
### 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

Acknowledgements	ix
Errata	xi
Foreword	1
A Note on the Data Sources	1
The Lean and Live Nature of This Book	2
Introduction	3
Preamble	3
What are we trying to do with the data?	4
Choosing the tools	5
The Data Sources	7
Additional Data Sources	11
Getting the Data into RStudio	12
Example F1 Stats Sites	13
How to Use This Book	14
The Rest of This Book...	15
An Introduction to RStudio and R dataframes	17
Getting Started with RStudio	17
Getting Started with R	18
Summary	46
Getting the data from the <i>ergast</i> Motor Racing Database API	47
Accessing Data from the <i>ergast</i> API	48
Summary	60
Getting the data from the <i>ergast</i> Motor Racing Database Download	61
Accessing the <i>ergast</i> Data via a SQLite Database	61

CONTENTS

The Virtual Machine Approach	61
Getting Started with the <i>ergast</i> Database	62
Asking Questions of the <i>ergast</i> Data	68
Summary	84
Addendum	85
Data Scraped from the Formula One Website (Pre-2015)	89
Format of the Original <i>scraperwiki.sqlite</i> Database	89
Format of the <i>f1com_results_archive.sqlite</i> Database	94
Problems with the Formula One Data	99
How to use the Formula1.com Data alongside the <i>ergast</i> data	99
Reviewing the Practice Sessions	101
The Weekend Starts Here	101
Practice Session Data from the Official Formula One Website Prior up to 2014 . .	102
Sector Times (Prior to 2015)	125
Summary	139
Practice Session Utilisation	141
Session Utilisation Charts	144
Finding Purple and Green Times	150
Stint Detection	154
Revisiting the Session Utilisation Chart - Annotations	167
Session Summary Annotations	170
Session Utilisation Lap Delta Charts	173
Summary	175
Useful Functions Derived From This Chapter	176
A Quick Look at Qualifying	179
Qualifying Progression Charts	181
Improving the Qualifying Session Progression Tables	183
Qualifying Session Rank Position Summary Chart - Towards the Slopegraph . .	185
Rank-Real Plots	189
Ultimate Laps	190
Summary	190
A Further Look at Qualifying	191
Clustering Qualifying Laptime by Session	193
Purple and Green Laptimes in Qualifying	196

How do Session Cut-off Times Evolve Over the Course of Qualifying?	203
Summary	207
Lapcharts and the Race Slope Graph	209
Creating a Lap Chart	209
Lap Trivia	226
Lap Position Status Charts	229
The Race Summary Chart	235
Position Change Counts	237
The Race Slope Graph	239
Further Riffs on the Lapchart Idea	242
Summary	246
Race History Charts	247
The Simple Laptime Chart	250
Accumulated Laptimes	253
Gap to Leader Charts	256
The Lapalyzer Session Gap	259
Eventually: The Race History Chart	259
Summary	263
From Battlemaps to Track Position Maps	265
Identifying Track Position From Accumulated Laptimes	265
Calculating DIFF and GAP times	269
Battles for a particular position	281
Generating Track Position Maps	284
Summary	287
Pit Stop Analysis	289
Pit Stop Data	289
Pit Stops Over Time	301
From Pitstops to Stints	303
Summary	304
Career Trajectory	305
The Effect of Age on Performance	308
Statistical Models of Career Trajectories	312
Modeling the Perfromance of F1 Drivers In General	319
The Age-Productivity Gradient	321

CONTENTS

Summary	321
Streakiness	323
Spotting Runs	325
Generating Streak Reports	329
Streak Maps	333
Team Streaks	336
Time to N'th Win	340
Looking for Streaks Elsewhere	342
Summary	342
Keeping an Eye on Competitiveness - Tracking Churn	343
Calculating Adjusted Churn - Event Level	345
Calculating Adjusted Churn - Across Seasons	352
Taking it Further	359
Summary	359
Laps Completed and Laps Led	361
Calculating Laps Completed and Laps Led Percentages	363
Comparing laps led counts over seasons	365
Comparing Laps Led Counts for Specified Circuits Across Several Years	368
Laps Led From Race Position Start	371
Summary	377
Event Detection	379
Detecting Position Change Groupings	379
Detecting Undercuts	390
Summary	393
Comparing Intra-Team Driver Performances	395
Intra-Team League Tables	395
Race Performance	401
Summary	405
Points Performance Charts	407
Grid Points Productivity	407
Maximising Team Points Hauls	412
Intra-Team Support	420
Points Performance Charts - One-Way	422

Points Performance Charts - Two-Way	428
Summary	437
End of Season Showdown	439
Modeling the Points Effects of the Final Championship Race	439
Visualising the Outcome	441
Summary	442
Charting the Championship Race	443
Getting the Championship Data	443
Charting a Championship Points Race	444
Charting the Championship Race Standings	448
Summary	453
Conclusion	455
Bibliography	457
Appendix - Converting the ergast Database to SQLite	459

Foreword

For several years I've spent Formula One race weekends dabbling with F1 data, posting the occasional review on the F1DataJunkie website (f1datajunkie.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. Where the data is openly licensed, such as the *ergast* motor racing results API (<http://ergast.com>) maintained by Chris Newell, 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 an archival copy of the data I have made available for the purpose of reporting, personal research and and/or criticism.

To make it easier to get started working with the datasets, I have put together a set of *f1datajunkie* Docker containers that together make up a lightweight virtual machine that contains all you need to get started analysing and visualising the data. In addition, source code for many of the analyses contained within this book can be found in the *f1datajunkie* Github repository.

Source code - and data sets (as *.sqlite* files) - are available from:

<https://github.com/psychimedia/wranglingf1datawithr/tree/master/src>

Note that the contents of this repository may lag the contents of this book quite significantly.

The Lean and Live Nature of This Book

This book was originally published using *Leanpub* (leanpub.com), a “lean book production” website that allows books to be published as they are being written. This reduced the production time and allowed the book to be published in an incremental way and allows any errors that are identified to be corrected as soon as they are spotted; purchasers of the book on the Leanpub site get access to updates of the book, as they are posted, for no additional cost.

This book has also generated been generated from a “live document” in the sense that much of the content was generated automatically *from code*. All the data tables and charts that appear in the book (and even some example text statements) were created by the code that directly precedes the tables and charts in the pages that follow.

But enough of the background... it's time to begin...

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 what we might term *sports statistics* or *sports stats*: descriptive summary statistics about previous races, previous championships, and the performance of current or previous drivers. Sports statistics are unusual compared to many datasets in that they are exact and unambiguous in terms of what they record: the results are the actual results of actual races, rather than sampled results based on uncertain opinion polls, for example.

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...

As well as sports stats, we'll have a look at some simple statistical modeling, taking inspiration from academic papers on economics and statistics and seeing whether we can build explanatory or predictive models about various aspects of Formula One. Academic papers are supposedly written to provide detailed enough explanations that a third party can attempt to reproduce any claimed findings, a practice that we shall put to the test several times! (It is only in recent years that publishing code to support research papers has become more widespread, in part through the advocacy of the Open Science movement.)

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 visualise data in general, it's also the case that we want to use these techniques to learn something about the world of F1. You might quite reasonably ask why we should even bother looking at the data, given the wide variety of practice and qualifying session, as well as race and championship reports that are produced around each race and across the course of a season. When interpreted correctly, data can provide a valuable source of stories that can help us better understand what actually happened in a particular race or over the course of a particular season, as well as highlighting stories that we might otherwise have missed. A good example of this is in the midfield of a particular race, where the on-track action may not receive much television coverage, no matter how exciting it is, particularly if the race at the head of the field is fierce.

So what is the data actually good for?

Firstly, we can use the data as a knowledgeable source we can have a conversation with about F1, if we know how to phrase the questions in the right way and we know how to interpret the answers. *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, although you may not think of it in such terms: you put in a keyword, Google gives you some links back. You skim 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 and in doing so developed your own understanding of the topic, mediated by the questions you asked, the responses you got, and the follow-on questions they provoked. After reading this book, you'll hopefully be similarly able to have a conversation with some raw datasets!

In the second case, we can look for stories in the data that tell us about *what has happened* in a particular race, championship season or driver career - 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; how a particular race or championship battle evolved. And so on. This is one of the main motivations for developing the charts described in this book - trying to find ways of visualising the story so that we can more easily see the stories that may be hidden in the data. It's often said that a picture can save a thousand words; but if those thousand words are a race review, how can a picture help us tell that story, and how can we find that story in, or read it from, that picture?

Thirdly, 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. For the teams, predicting the possible ways a race might evolve can the strategy or tactics employed by the team during a race. For the gambler, forecasts may influence betting strategy, not something I will cover in this book, except to say that one should always remember that previous outcomes are no guarantee of future success! For the commentator, knowing how a race may evolve can help add context to a commentary whilst trying to explain the actions of a particular driver or team. Modeling and predicting races is not something that is covered in this particular book, though I hope to cover it in a future one.

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 first 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/IPython notebook/matplotlib*. Both these triumvirates are popular among those emerging communities of data scientists and data journalists. A third possibility was 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.com* blog. The R milieu has also benefited 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 environment (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 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 Aslett¹ or Running RStudio via Docker in the Cloud²). 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, including the *ergast motor racing results database* and data scraped from official FIA and Formula One sources.

ergast Motor Racing Database - Overview

The ergast experimental Motor Racing Developer API³ provides a historical record of Formula One results data dating back to 1950.

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

- *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 onward
- *Standings* - driver and constructor championship 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

¹http://www.louisaslett.com/RStudio_AMI/

²<http://www.magesblog.com/2014/09/running-rstudio-via-docker-in-cloud.html>

³<http://ergast.com/mrd/>

- *Finishing Status* - describes the finishing status for each competitor
- *Lap Times* - race lap times from the 2011 season onward
- *Pit Stops* - pit stop data for each race from 2012 onward

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.

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 official online websites and put it into a database, such as a SQLite database.

Up until the start of the 2015 season, the formula1.com website published current season and historical results data dating back to 1950. From 1950 to 2002 only race results were provided. Since 2003, the data included results from practice and qualifying sessions. From 2004, best sector times and speed trap data was also available for practice and qualifying sessions, and fastest laps and pit stop information for the race.

At the start of the 2015 season, a redesign of the official Formula One website removed all but classification results from the public areas of the site at least. As such, the official website is now of little use to the F1 data junkie. However, the current season results that used to previously appear on the F1 website now appear on pages on the FIA website.

In the original drafts of this book, an appendix described a python screenscraper used to scrape the data from the Formula One website. With the historical results pages no longer available (although you could always try the Internet Archive...), I have posted an archival version of the data as a SQLite database (`f1com_results_archive.sqlite`) in the github repository along with a copy of the original scraper (`code/f1-megascrapercode.py`).

In place of the screenscraper, I have produced a set of R functions that scrape the race classification pages on the FIA website so that it can be worked with *as data*.

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 and

intended for use primarily by the media. These documents represent a primary source for F1 timing information - experience shows that results data posted to the results tables on the public facing F1 and FIA websites is not always correct...

Until 2012, the FIA timing sheet documents for each race would remain available until the next race, at which point they would disappear from the public FIA website. From 2013, an archive site⁴ kept the documents available although following a redesign of the FIA website at the start of 2015, trying to track down any archived event and timing information documents, if indeed they are still available, has become all but impossible.

Timing and event information for the 2015 season is currently available from URLs rooted on <http://www.fia.com/events/fia-formula-1-world-championship/season-2015/>. Documents for the first race can be found at `event-timing-information`, for the second race `event-timing-information-0`, for the third `event-timing-information-1` and so on. As well as official PDF timing sheets, the FIA website also publishes session classification tables for each sessions and summary tables for qualifying and the race.

Downloading the official 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* - brief introduction to the race, quick facts, summary of standings to date, circuit map
- *Circuit Information* - graphics of FIA circuit map, F1 circuit map
- *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
- *Stewards Decisions* - notices about Stewards' decisions for each day, with information broke down into separate list items (No/Driver, Competitor (i.e. the team), Time, Session, Fact, Offence, Decision, Reason)

⁴<http://www.fia.com/championships/archives/formula-1-world-championship/2013>

- *Championship Standings* - drivers and constructors championship standings once the race result is confirmed

The FIA PDF Timing Sheets in Detail

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

- **Practice**
 - Classification
 - Lap Times
- **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 was historically also published as HTML data tables on the previously mentioned formula1.com *results* data area; from 2015, the HTML summary tables appear on the FIA website.

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.

Several tools are available that can help extract data directly from PDF documents. For example, *Tabula*⁵, open source, cross-platform desktop application or the Scraperwiki *PDFTables*⁶ service, which is commercial, although with a limited free tier. Several programming libraries are also available if you want to write your own scrapers.

Getting the HTML webpage data is much easier and a webscraper will be described that shows how the web page datatables can be automatically captured into an R dataframe.

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.

Additional Data Sources

Several additional data sources are also available that interested readers may like to explore, though the list is subject to change as new websites come online and others disappear or lapse by the wayside.

Viva F1 - Race Penalties

For the 2012 and 2013 seasons, the Viva F1⁷ site publish a summary list of race penalties⁸ 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.

⁵<http://tabula.technology/>

⁶<https://pdftables.com/>

⁷<http://www.vivaf1.com>

⁸<http://www.vivaf1.com/penalties.php>

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 a dashboard 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, as well as a limited amount of cornering speed information, but the data is not made openly available. The commercial licensing decisions surrounding this particular set of F1 data thus makes fan driven innovation around it very difficult.

A Note on Data Licensing

Although an increasing number of publishers, such as the *ergast* data service, make data available under a permissive open license that allows the data to be freely shared and reused, rights to much of the data associated with motorsport extends no further than fair use conditions that apply to copyrighted material released with no additional license terms other than a standard “All Rights Reserved” limitation. Data may be used for timely reporting or personal research, and the educational use of copyrighted material also benefits from certain freedoms. Restrictions may still apply to sharing of data so used, however, which is why I have tried to avoid the sharing of data that may have rights associated with it that prevent sharing. In such cases, I have tried to describe ways in which you might be able to get hold of the data, *as such*, so that you can analyse it *as data* yourself.

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 repository⁹. 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 *ergast* 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 the database is configured with whatever permissions are required to allow R to access the data. To simplify the database route, I have converted 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. A *docker container image containing the sqlite version of database, along with scripts for importing the SQL database from the *ergast website* is also available as part of a bundle associated with this book on the Leanpub website.*

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

Sample datasets (in sqlite form) can be downloaded from [github/psychimedia/wranglingf1datawithr](https://github.com/psychimedia/wranglingf1datawithr)^a as: * *ergast* database - *ergastdb13.sqlite* * F1 results scrape (original version) - *scraperwiki.sqlite* * F1 results scrape (archive to end of 2014) - *f1com_results_archive.sqlite*

^a<https://github.com/psychimedia/wranglingf1datawithr/tree/master/src>

Example F1 Stats Sites

Several websites produce comprehensive stats reports around F1 that can provide useful inspiration for developing our own analyses and visualisations, or act as a basis for trying to replicate the analyses produced by other people.

I have already mentioned the intelligentF1¹⁰ website, which used to analyse 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, from the 2014

⁹<https://github.com/psychimedia/wranglingf1datawithr/blob/master/src/ergastR-core.R>

¹⁰<http://intelligentf1.wordpress.com/>

season, fuel saving. The James Allen On F1¹¹ features a strategy review of each race a day or two after each race.

Applied mathematician Andrew Phillips' F1Metrics blog¹² describes a wealth of detailed analyses of F1 data and provides a far more rigorous and formal approach than the approaches described in this book; it represents an excellent resource if for taking some of the ideas hinted at in these pages further. And in some cases, *much* further!

For tracking a season on the race stats side, F1fanatic¹³ 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.

During race weekends, data fragments tend to appear in the race week end thread on the relevant *f1technical.net* forum.

Over the 2015 season, the *f1forensics* tag on *The Judge13*¹⁴ website collated reviews of laptimes and technical documentation associated with each race, with the associated Chancery¹⁵ archive maintaining a running database covering a range of measures including engine usage and the distances run by a variety of technical components on a per car basis. The spreadsheet associated with that archive looks to be a hugely valuable resource, although I discovered it too late to include any analyses based on the data in this edition of the book.

Although not an F1 stats site *per se*, I always enjoy visiting *sidepodcast.net*¹⁶.

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. All the tables and all the charts are produced directly from code snippets described in the text. 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, and

¹¹jamesallenonf1.com

¹²<https://f1metrics.wordpress.com/>

¹³<http://www.f1fanatic.co.uk/statistics/2014-f1-statistics/>

¹⁴<http://thejudge13.com/category/f1-forensics/>

¹⁵<http://thejudge13.com/f1-forensics/>

¹⁶<http://sidepodcast.net>

then experiment with changing recognisable bits of the code yourself. Non-executed code 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 explicitly in the main text.

Several of the chapters include one or more *Exercise* sections that describe recreational data puzzles and exercises for you to practice some of the things covered in the chapter, or that suggest ways of applying or extending the ideas in new ways. During the production of this book, some sections originally included *TO DO* items; these reflected the work-in-progress nature of the book and provided placeholders for activities or analyses to be included in future rolling editions of the text. *TO DO* items often went beyond simply rehearsing or stretching the ideas covered in the respective chapter and typically required some new learning to be done, problems to be solved, or things to be figured out!

The Rest of This Book...

For the first 12 months of its existence, this book was largely a living book, which meant that it was subject to change on a regular basis. The book has now reached a relatively stable state of development and henceforth will be subject mainly to revisions arising from errata or code improvements. A version of the book will also be made available in paperback form on *Lulu.com* when any remaining errata have been rectified. Any significant new chapters will be included in a new book: *More Motorsport Data Wrangling With R*.

The chapters are 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 tools and techniques for analysing and visualising results across a championship and comparing performances year on year;

Original versions of the book hinted at the inclusion of chapters on:

- *map views* - a look at how what geo and GPS data is available, and how we might be able to make use of 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.

With this version of the book already reaching several hundred pages, the description of interactive data displays will now appear in *More Motorsport Data Wrangling With R* with live demonstrations on an associated website.

If you spot any problems with the code included in this book, please post an issue to Wrangling F1 Data with R - github¹⁷.

If you would like to buy this book, or make a donation to support its development, please visit Wrangling F1 Data with R - leanpub¹⁸.

¹⁷<https://github.com/psychimedia/wranglingf1datawithr/issues>

¹⁸<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 visualisation 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 *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, are then 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 Computing¹⁹. The download area²⁰ 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

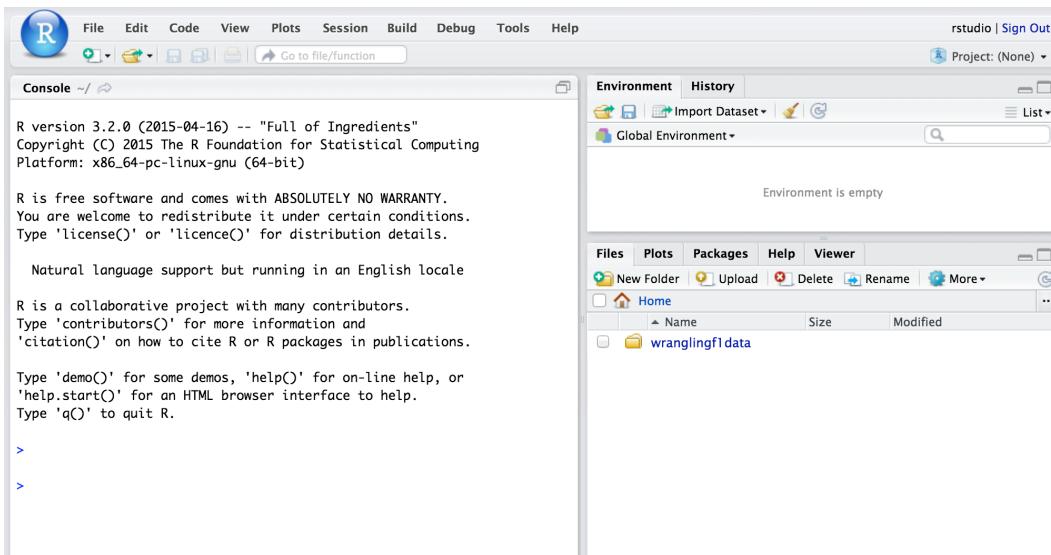
¹⁹<http://www.r-project.org/>

²⁰<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.org²¹.**
- **download and run the F1DataJunkie docker container from wranglingf1data-docker²²**

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 of its own!). 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.

²¹<https://www.rstudio.com/ide/download/>

²²<https://github.com/psychimedia/wranglingf1data-docker/blob/master/README.md>

Note that there is no trickery involved in this book - the data table and chart outputs are generated by running only and exactly the code shown prior to them. Indeed, it is the output from running each code block that is automatically inserted into the original manuscript of this book as the manuscript is “compiled”.

Functions for everything...

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.

```
getwd()
```

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

In this case, I am working in the `src` subfolder of the `wranglingf1datawithr` directory in my Dropbox folder.

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 complementary `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 (or 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), 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 two-dimensional tabular data set 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”.

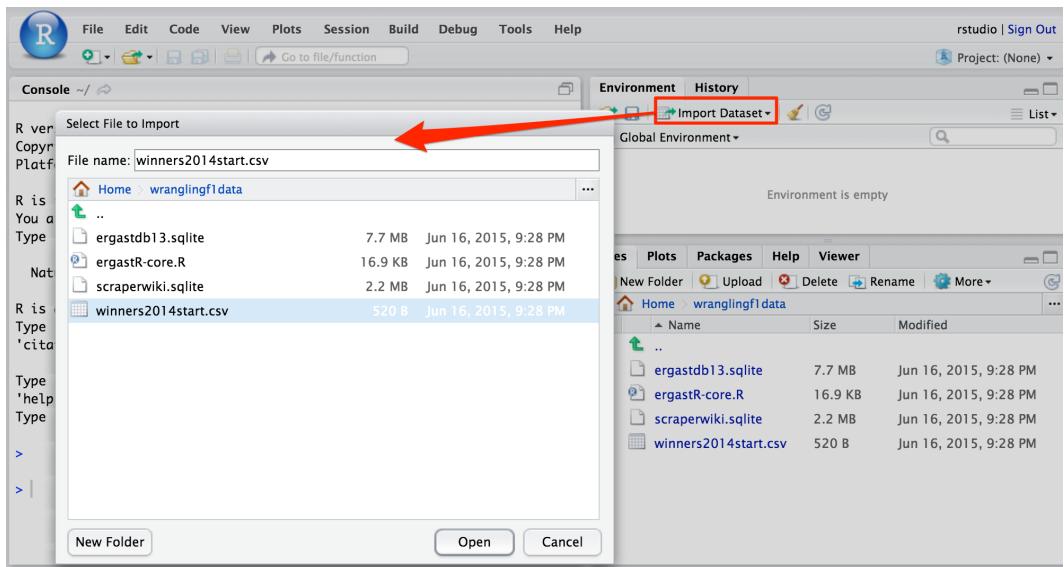
The style of computation used by R is often referred to as *vectorised* computation, and is reminiscent of the column based computing that will be familiar to many spreadsheet users. Such a computational style means that the same operation is often applied to each cell in a particular column simply by applying the operation to the column as a whole. For example, a single command can let you add the same number to each cell in a particular column, or add the values across two columns in the same row together.

To explore some of the features of a dataframe, I have produced a sample dataset that contains data from the first three races of the 2014 season, identifying who was on the podium for each race 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.csv²³. 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 to, you can load the data into RStudio:

²³<https://gist.github.com/psychemedia/11187809#file-winners2014start-csv>



You can also use the `read.csv()` function to load the data in yourself and assign it to a *variable*, in this case the variable `winners`:

```
# "Comments" are preceded by a hash symbol
# Comments are unexecuted text within a code area.
# They are provided as a commentary to aid the reader / programmer

#Use the read_csv() function to read in the contents of a CSV file as a dataframe
#Data file available at:
#https://gist.github.com/psychimedia/11187809/raw/winners2014start.csv
#Assign the dataframe to the variable: winners
winners=read.csv('~/Dropbox/wranglingf1datawithr/src/winners2014start.csv')

#Display the contents of the variable
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

```



Variables

A variable is like a named container into which you can place, and then retrieve, different things, such as a single particular value - the name of a driver who won the Driver Championship in a particular year for example - up to a complete set of data, such as the laptime for every driver in a particular race.

In this case, the `winners` variable contains the data that identifies the podium finishers from the first three races of 2014.

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:

```

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.

As you can see, if we just enter the name of a variable, such as `winners`, the R interpreter will print out the contents of that variable.

Dataframes are a common currency in R for representing tabular data. In many respects, a dataframe resembles a worksheet in a spreadsheet application, with individual records organised in rows and particular aspects of each record arranged in columns.

Within a dataframe, if we want to reference just the values in a single column of a data frame 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
```

This form of indexing returns another dataframe, albeit one with just a single column. We can also use the column number, with the leftmost column counting as column 1, to index the column: `winners[3]`, for example.

We can also get what essentially amounts to a *list* of the values contained within a column several other ways:

- DATAFRAME [['COLNAME']]: for example, `winner[['driverId']]`
- DATAFRAME\$COLNAME: for example, `winners$driverId`
- DATAFRAME [[COLNUMBER]]: for example, `winner[[3]]`

```
winners[['driverId']]

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

We can also index into rows as well as columns. The notation `DATAFRAME[ROWS, COLUMNS]` allows you to select rows by index value and column number, or name.

If we wish to inspect the contents of a dataframe in particular in a more convenient way, we can use 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:

The screenshot shows the RStudio interface. In the top right, it says "rstudio | Sign Out" and "Project: (None)". The main area has a red arrow pointing from the "Environment" tab in the top navigation bar to the "winners2014start" data frame listed under "Data". The "winners2014start" frame is highlighted with a red border. Below the environment pane, there's a file browser showing files like "ergastdb13.sqlite", "ergastR.core.R", "scraperwiki.sqlite", and "winners2014start.csv". At the bottom, the "Files" tab is selected.

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 **integer** value contents, (for example, the *carNum* and *pos* columns), *fastlaptime* has been identified as a **numeric**, 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 or factor. We can convert 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()`:

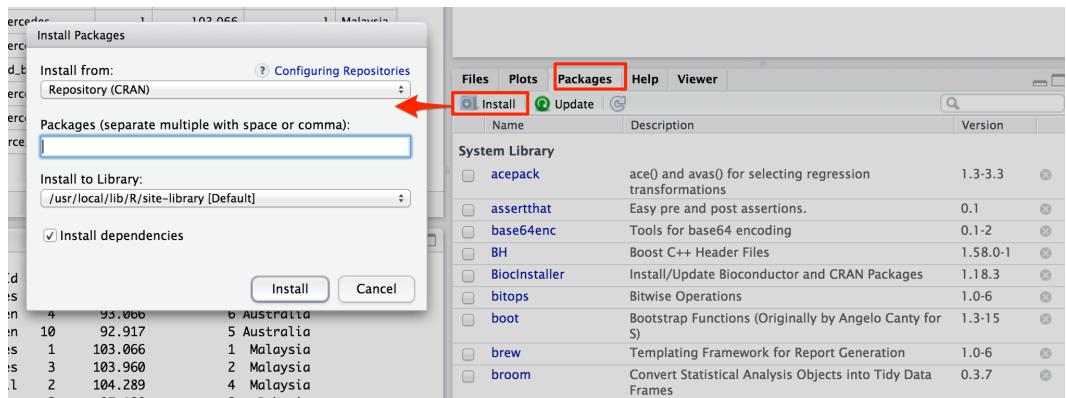
```
summary(winners)
```

```

## 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, if the data file is being served from a secure *https* link, rather than a simpler *http* web address, 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()[, "Packag\
e"])]
if(length(new.packages)) install.packages(new.packages)
lapply(list.of.packages, function(x){library(x, character.only=TRUE)})

## [[1]]
## [1] "RCurl"      "bitops"     "gridExtra"   "grid"       "ggplot2"
## [6] "plyr"       "knitr"      "RSQLite"    "DBI"       "stats"
## [11] "graphics"   "grDevices"  "utils"      "datasets"   "methods"
## [16] "base"
##
## [[2]]
## [1] "RCurl"      "bitops"     "gridExtra"   "grid"       "ggplot2"
## [6] "plyr"       "knitr"      "RSQLite"    "DBI"       "stats"
## [11] "graphics"   "grDevices"  "utils"      "datasets"   "methods"
## [16] "base"
```

So how can we use the RCurl package to load in data from a secure URL? The following recipe provides and example of loading a raw data file from a Github gist:

```
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'

#Join the filepath to the urlstub with a / between the parts
url=paste(urlstub,urlpath,sep='/')
url
```

```
## [1] "https://gist.githubusercontent.com/psychemedia/11187809/raw/38295d24d5e2e35e0\ b92cb2cf4082a500a691ffd/winners2014start.csv"
```

The `paste()` command concatenates a comma separated list, joining values in the order presented with a separator specified by the `sep=' '` parameter; use `sep=''` for a seamless join.

```
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 one or more variables into a function, we put it inside the brackets. The order in which the values are included is determined for required values; optional values appear in any order if we explicitly identify which parameter we are referring to. 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, or *parameters*, 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` data frame, 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
```

If you look up the documentation for a function by entering the function name preceded by a `?`, it should clearly identify the parameters associated with the function.

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 each data table. (`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 print it out using the markdown 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

As you should see, the presentation style of this table is much improved over the raw printout styling.

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 real or floating point number. For the vector based computations used to process R dataframes, each column in a data frame should be capable of being represented as a *vector* in which all the values take on the same data 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*. Remember, 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. One of the reasons that vectorised computation is useful because it can be efficiently scaled across multiple processors. Whilst we will not be processing huge datasets, it is often worth remembering that choices about how data is represented can often determine how quickly or efficiently it can be processed.

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, by default 1 to N for seq(N)
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

head( winners2, 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
posdelta 1 Australia 2 2 Australia 2 3 Australia 7

In many programming languages, *packages* or *libraries* are used to collect together functions that are related in some particular way, such as their usefulness for a particular range of tasks, or for processing a particular sort of data. For example, this book makes considerable use of Hadley Wickham's `ggplot2` library for generating graphics. Other libraries, such as `plyr`, contain a range of functions that help us reshape a dataset.

For example, 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 )
head( winners3, 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
posdelta 1 Australia 2 2 Australia 2 3 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 )
head( winners4, 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
posdelta poslapdelta 1 Australia 2 0 2 Australia 2 4 3 Australia 7 2

If you want to create a column that is used in turn to define a further column within the same function call, 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.

Being able to transform a dataset is often an important step in getting the data into a shape that allows us to readily generate a visual representation of it. In later parts of the book, you will see how we can also apply transformations to particular *subgroups* of a dataframe generated as a result of grouping rows based on some common property.

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, techniques: a “cryptic” way, and a more verbose way.

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

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

```
kable( subset(winners, subset=(pos==2 | pos==3)), row.names=FALSE )
```

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

Remember, `kable()` simply styles the output value of the function it encloses in a “prettified” way. The `row.names=FALSE` parameter suppresses the display of any index values.

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.



Boolean Operators

The `|` symbol is a logical (Boolean) operator that represents the operation OR; `A | B` reads *A or B* and is true if either *A or B or A and B* is true. We can test for inequality using the `!=` operator. We can also test for Boolean *anded* conditions using the operator `&`. Numerical operators can also be applied as a the basis of a comparison that returns a Boolean (true/false) response; for example, setting the filtering parameter `subset=(pos>=2)` would give the same result as shown in the example above.

Typically, we enclose logical (Boolean) statements such as `pos>=2` within a set of round brackets - `(pos>=2)` that we assume returns a Boolean TRUE or FALSE result.

Operationally, we can achieve the same thing by passing in a “truth vector” that says whether or not a logical test applied to each 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.

Recalling the ability to select rows and columns by explicitly indexing into a dataframe, `df`, using the formulation `df[ROWSELECTION, COLUMNSELECTION]`, we can select just a subset of the output columns by passing in a list of the names of the columns we want to select as index values:

```
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	Australia
4	Malaysia
7	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.



Exercises

In the `winners` data frame, what levels are associated with the `constructorId` and `driverId` factors?

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

Sorting dataframes

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

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

```
#The arrange function accepts a data frame 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
```

To specify the reverse, or descending, order direction, use the - (minus) operator. (In fact, using the minus sign retains the sort order, we just reverse the number line we are sorting against.)

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 values contained within the columns are ordered differently. It can also 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. We will see how to apply this form of sorting throughout the book.

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. The ordered value list should contain a single series of items, with the values across a row listed next to each other.

```

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-called *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

```

The sense of this is to reshape the dataset so that each row identifies a driver by `driverId` and each column represents a different race.



Long, Wide and Tidy Datasets

Conventionally, *wide* data is thought of as data in which each row represents an entity and the columns represent properties associated with that entity. In a *long* (sometimes referred to as *narrow*) dataset, each row represents the value of particular property of a given entity. In a long dataset, there is typically a key or index column, a “variable” column that identifies a particular property, and a “value” column containing the value of that property for that variable. As Hadley Wickham suggests in Wickham, H, *Tidy Data*, Journal of Statistical Software, August 2014, Volume 59, Issue 10²⁴, the notions of *wide* and *long* format are imprecise, although I still find them convenient! Instead, Wickham promotes the idea of *tidy* data, formally identified in the database literature as data organised according to Codd’s third normal form (3NF).

In reshaping the data, the `dcast` function identifies the unique values in the `race` column, and uses these as new column names. It then inspects each row in the original dataframe,

²⁴<http://www.jstatsoft.org/article/view/v059i10>

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, at the same time making it rather more explicit as to how exactly we want to reshape the dataset.

```
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. (See for example, Wickham, H. The Split-Apply-Combine Strategy for Data Analysis, *Journal of Statistical Software*, April 2011, Volume 40, Issue 1²⁵.)

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(winner, .(constructorId), summarise, meanpos=mean(pos))
```

²⁵<http://www.jstatsoft.org/article/view/v040i01>

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

In a *summarise* operation, the number of rows in the resulting data frame (that is, the *length* of the data frame) is the same as the number of groups we are summarising.

Another way of using `ddply` is to *transform* or *mutate* a dataframe. We have already met the `transform` and `mutate` operations, as applied to a column across all the rows within a dataframe. When applied using `ddply()`, these operations are called relative to the rows of particular groupings (that is, subsets of rows) contained within the 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 programming 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 documentation²⁶, or one of the increasing number of books on RStudio²⁷.

As to the R language itself, we have introduced many of the key techniques and functions that will be used throughout the rest of this book: how to construct 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. I will also introduce another powerful library produced by Hadley Wickham, the `ggplot2` charting library, which provides a powerful set of tools for generating common, as well as bespoke, data visualisations.

²⁶<https://support.rstudio.com/hc/en-us/categories/200035113-Documentation>

²⁷http://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 Gallery²⁸.

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 slightly 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 more timely, for example, when trying to access qualifying data over the course of a race weekend). 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!

²⁸<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.

You can inspect the data published by the *ergast* API by following the example method call links from the API documentation webpages. Clicking through on a link for a particular example API call leads to a web page that previews the data associated with that call in one or more HTML tables.

The *ergast* API also publishes data as a JSON or XML data feed, obtained by adding a `.json` or `.xml` suffix respectively to the base web address for a particular API call. Handling the data directly is a little bit 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. The library can be found in the file `ergastR-core.R`²⁹ at the URL specified in the footnote 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 race number;
- `qualiResults.df(YEAR?,RACENUMBER?,DRIVERID?,CONSTRUCTORID?)`: qualifying results selected according to defined parameters;
- `raceWinner(YEAR,RACENUMBER)`: the winner of a race specified by year and race number;
- `pitsData(YEAR,,RACENUMBER)`: details of each pit event during the race;
- `driverResults.df(YEAR,DRIVERID)`: results for a specified driver for a specified year;
- `lapsData.df(YEAR,RACENUMBER)`: information about laptimes during a particular race;
- `lapsDataDriver.df(YEAR,RACENUMBER,DRIVERID)`: information about laptimes during a particular race for a particular driver;
- `driverCareerStandings.df(DRIVERID)`: information about the career standing in terms of end of season classifications for a particular driver;

²⁹<https://gist.github.com/psychimedia/11187809#file-ergastr-core-r>

- *seasonStandings(YEAR,RACE?)*: drivers' championship standings at the end of the year (for a previous season) or, for the current year, the current standings, or (optionally), the standings end of a particular race;
- *constructorStandings.df(YEAR,RACE?)*: constructors' championship standings at the end of the year (for a previous season) or, for the current year, the current standings, or (optionally), the end of a particular race.

Where a variable is qualified with a ?, the question mark/query symbol identifies the corresponding variable as an *optional* element; that is, you do not need to provide it explicitly, in which case a default behaviour will result.

Introducing some of the simple *ergastR* functions

To load the core *ergastR* functions in, download the *raw* file³⁰ 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.

```
#If the R file is in the current working directory
source('ergastR-core.R')
#Example of specifying the path to the file located on my own computer
#The ~ symbol denotes my home/default directory
#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)
#url='https://gist.githubusercontent.com/psychemedia/11187809/raw/ergastR-core.R'
#source_url(url)
```

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.

³⁰<https://gist.github.com/psychemedia/11187809#file-ergastr-core-r>

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

#The knitr library contains a handy function, kable(), for generating tabular markdown
#We can use it in an Rmd script by setting an Rmd chunk
#with the option {r results='asis'}
#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 Malaysian Grand Prix, which happened to be round 2 of that year:

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

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

Having access to laptime data is essential for many race reports. In later chapters, we will see a wide variety of ways in which can put this form of data to use, such as generating race history charts, that capture the dynamics of a complete race, or battle charts that show a race from a particular driver's perspective.

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.



Exercise

Try out some of the other functions contained in `ergastR-core.R`, such as `driverResults.df(YEAR, DRIVERID)` or `lapsDataDriver.df(YEAR, RACENUMBER, DRIVERID)`.

Making Function Calls to the *ergast* API

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

```
raceWinner
```

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

If you want to edit the function, enter `fix(FUNCTION_NAME)`; for example, `fix(raceWinner)`.

We see how the URL for the corresponding request takes the form `http://ergast.com/api/f1/YEAR/RACENUMBER` (`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) {
  dataPath=paste(year,raceNum,"results",racePos,sep='/')
  wURL=paste(API_PATH,dataPath,".json",sep='')

  wd = fromJSON(wURL, simplify = FALSE)
  wd$MRData$RaceTable$Races[[1]]$Results[[1]]$Driver$driverId
}

#For the 2014 championship, second round, show who was classified in third place
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 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, `raceWinner` and `racePosition`, 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.)

Many code libraries are released under what are known as *open licenses*, which means that you are free to use them, repurpose them, and extend them yourself. When developing your own code, it often makes to sense to build on top of openly licensed code that already does elements of what you want, rather than starting from scratch every time.

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 race in a particular position is by indexing into the results dataframe as defined by the *ergast* API call `resultsData.df(YEAR, RACENUMBER)`.

You may recall that we can index into an R dataframe using the construction `df[ROW_SELECTOR, COLUMN_SELECTOR]`. Let's see if we can filter our 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 requested value is returned; for example, I can embed the driver ID that is returned here using an **inline code** expression which returns the value: 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 (*hamilton*, *vettel*, and so on). The `driverId` column in the `drivers.df` dataframe pulls from the same set of values, and contains additional information about each driver in its other columns. 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 merge column in the first table contains a slightly different set of unique values than does the merge column in the second table. These correspond to the INNER JOIN and OUTER JOIN methods of relational algebra, as used in query languages such as SQL, for example. *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	constructgrid	laps	status	fastlapnum	fastlaptime	fastlaprank	node	
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 be specified explicitly. The first dataframe is referred to as the *x* dataframe and the second one as the *y* dataframe; the 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 strttime 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
```

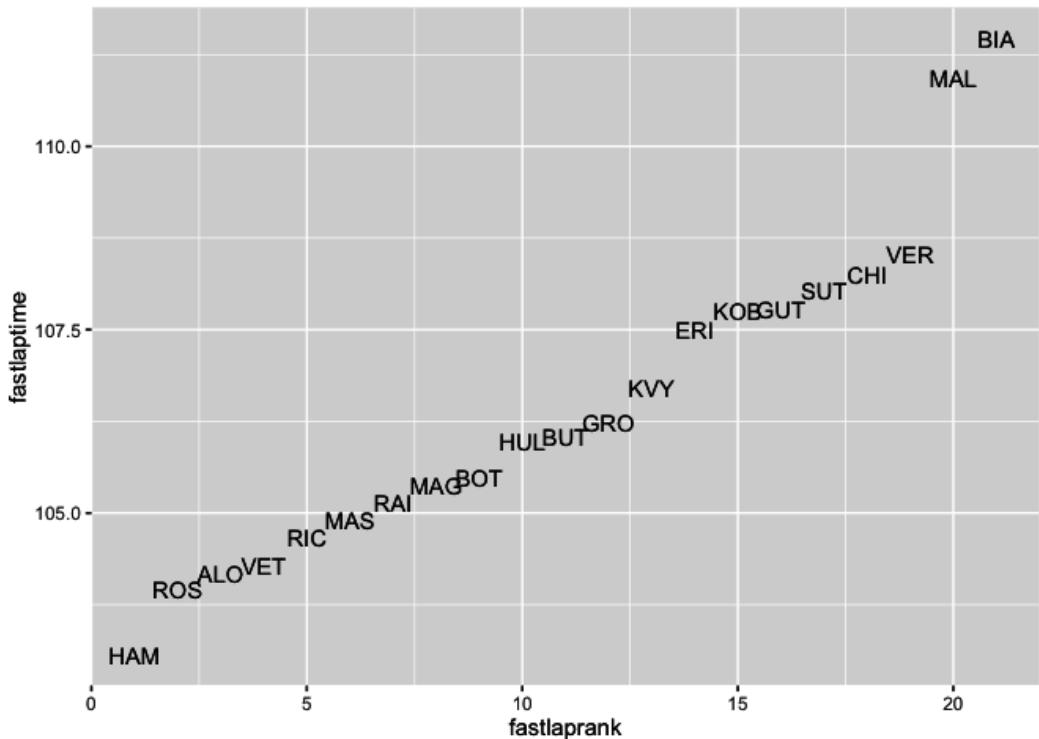
Having the three letter code available in the dataframe directly means we can access it easily when it comes to producing a chart. For example, we might plot the fastest lap time against the fastest lap rank, using the code to identify each point:

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

## Warning: package 'ggplot2' was built under R version 3.2.4

#A text plot is a scatterplot with text labels at each scatterplot point
g = ggplot(results.df) + geom_text(aes( x=fastlaprank, y=fastlaptme, 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).

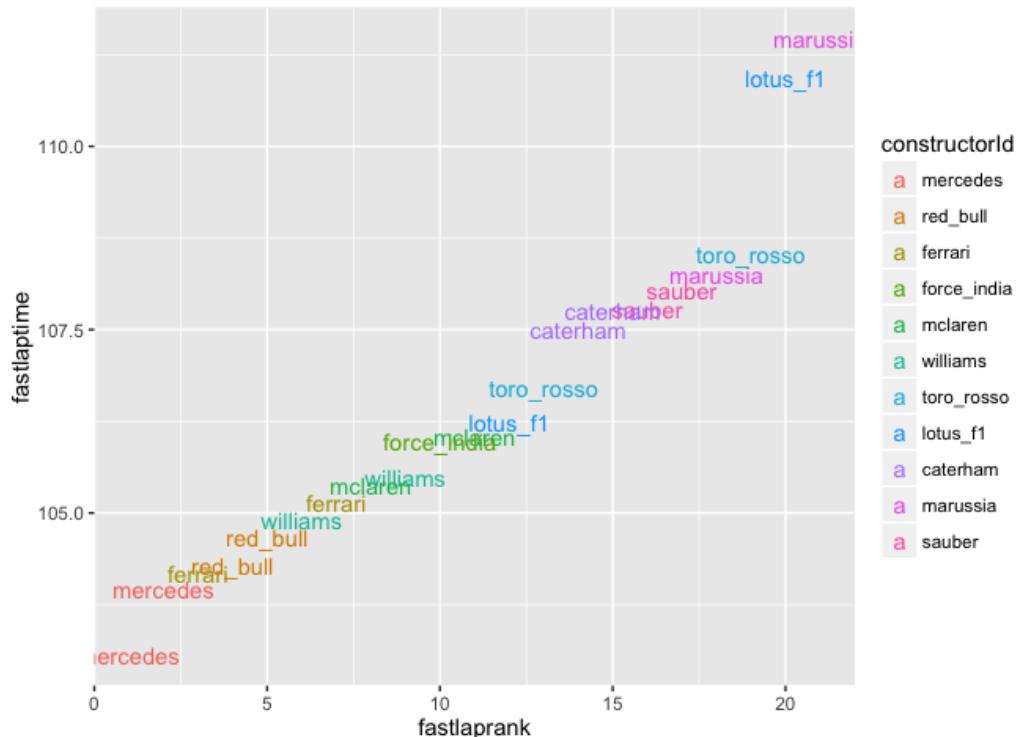
Although it's just a simple chart, we can see how the drivers' fastest laptimes split into several groups. Are these actually grouped by team? Let's see...

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=fastlapttime,
                      label=constructorId, col=constructorId))
g

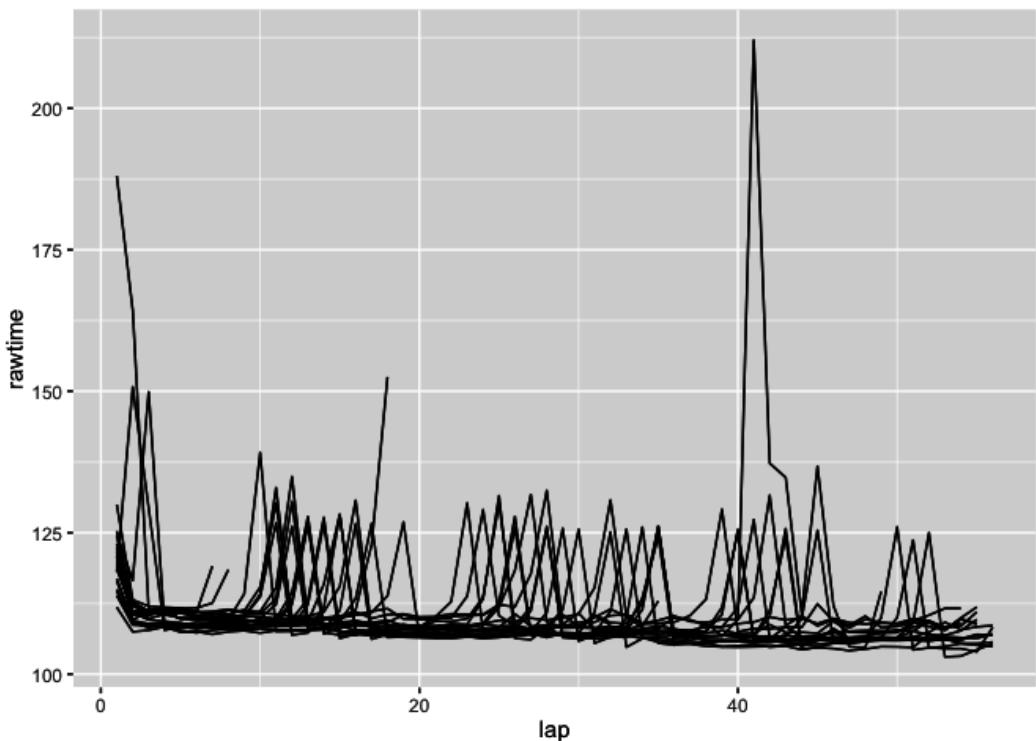
```



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.

The Grammar of Graphics

The *gg* in *ggplot* actually stands for *The Grammar of Graphics*, a formal approach to describing the design of data charts that make sense logically, as well as visually, in a grammatically sensible way. *The Grammar of Graphics* was originally developed by Leland Wilkinson and is described in a book of the same name. The *ggplot2* R package, which contains the *ggplot* R library, was originally developed by Hadley Wickham as an implementation of many of the ideas proposed in Wilkinson's book. For more information see Wilkinson, L. (2006) *The grammar of graphics*, Springer Science & Business Media, or Wickham, H. (2010) *A layered grammar of graphics*, Journal of Computational and Graphical Statistics 19.1 (2010) 3-28^a [subscription content].

^a<http://www.tandfonline.com/doi/abs/10.1198/jcgs.2009.07098#.VmTCzWThCGQ>

For now, you can see how quick it is to start sketching out graphical views of data using the *ggplot2* package, 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 file³¹. The R functions in this file *wrap* the 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 championship; the results for a particular race; the winning driver from a particular race (and by extension of that function, the driver finishing in 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 your 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...**

³¹<https://gist.github.com/psychemedia/11187809#file-ergastr-core-r>

Getting the data from the *ergast* Motor Racing Database Download

As well as being made available over the web via a JSON API, Chris Newell also releases the *ergast* data as a MySQL database export file updated at the end of each race.

If you want to use Formula One results data as a context for learning how to write database queries using SQL, one of the most popular and widely used database query languages, then this download is probably for you...

Accessing the *ergast* Data via a SQLite Database

MySQL is a powerful database that is arguably overkill for our purposes here, but there is another database we can draw on that is quick and easy to use - once we get the data into the right format for it: SQLite³². *For an example of how to generate a SQLite version of the database from the MySQL export, see the appendix.*

Unfortunately, the recipe I use to generate a SQLite version of the database requires MySQL during the transformation step, which begs the question of why I don't just connect R to MySQL directly. My reasoning is to try to use lightweight database tools where possible, and sqlite offers just such a solution: no database management system is required, just the sqlite database file and the RSQLite library to query into it.

*You can download a copy of the *ergast* database as a sqlite database from the wranglingf1datawithr repository³³ as ergastdb13.sqlite*

The Virtual Machine Approach

Another approach to setting up an environment in which to work with the *ergast* database is to load it into a MySQL database running as a service and then connect to that service from RStudio. However, this requires that a user can install the database server and populate it with the *ergast* database. Another approach is to run either the database server alone, or

³²<http://www.sqlite.org/>

³³<https://github.com/psychimedia/wranglingf1datawithr>

the database server *and* RStudio in a virtual machine running either on your own computer or on a cloud hosted machine.

In the latter case, we can use Docker virtual machine containers to run and interconnect a variety of services, such as a MySQL database server and an RStudio server, and then access the RStudio server through a browser based interface.

*For an example of connecting to the `*ergast` database running in a MySQL docker container, see Connecting RStudio and MySQL Docker Containers – an example using the `ergast db`*^{34*}

Getting Started with the *ergast* Database

We can access a SQLite database from R using the RSQLite package³⁵:

```
#Start off by creating a connection to the database
library(DBI)
ergastdb =dbConnect(RSQLite::SQLite(), './ergastdb13.sqlite')
```

The *ergast* database contains the following distinct tables:

```
## list all tables
#dbListTables() is provided as part of the DBI library
tables = dbListTables(ergastdb)

## exclude sqlite_sequence (contains table information)
tables = tables[tables != "sqlite_sequence"]

tables

## [1] "circuits"           "constructorResults"   "constructorStandings"
## [4] "constructors"        "driverStandings"      "drivers"
## [7] "lapTimes"            "pitStops"             "qualifying"
## [10] "races"                "races2"               "results"
## [13] "seasons"              "status"
```

If we're feeling adventurous, we can load the whole of the *ergast* database into memory as a set of R dataframes, one per database table, in a single routine. The dataframes take the name of the corresponding table in the database.

³⁴<http://blog.ouseful.info/2015/01/17/connecting-rstudio-and-mysql-docker-containers-the-ergastdb/>

³⁵<http://cran.r-project.org/web/packages/RSQLite/index.html>

```
1DataFrames = vector("list", length=length(tables))

for (i in seq(along=tables)) {
  assign(tables[i],
         dbGetQuery(conn=ergastdb,
                     statement=paste("SELECT * FROM '", tables[[i]], "'", sep="")))
}
```

The following shows the sort of data we can find in the *circuits* table (there is also a column, *not shown*, that contains a link to the Wikipedia page for the circuit. It may be possible to use this to automate a route for pulling in a circuit map from Wikipedia or DBpedia):

```
kable( head(circuits[,1:7],n=5),format='markdown' )
```

circuitId	circuitRef	name	location	country	lat	lng
1	albert_park	Albert Park Grand Prix Circuit	Melbourne	Australia	-37.84970	144.96800
2	sepang	Sepang International	Kuala Lumpur	Malaysia	2.76083	101.73800
3	bahrain	Circuit Bahrain International	Sakhir	Bahrain	26.03250	50.51060
4	catalunya	Circuit de Catalunya	Montmeló	Spain	41.57000	2.26111
5	istanbul	Istanbul Park	Istanbul	Turkey	40.95170	29.40500

One way of making use of this data might be to use a package such as RMaps to plot the circuits visited in each season on a world map, perhaps using great circles to connect the consecutive races to show just how far the race teams travel year on year.

Here's an example of the *constructorResults*:

constructorResultsId	raceId	constructorId	points
1	18	1	14
2	18	2	8
3	18	3	9
4	18	4	5
5	18	5	2

In and of itself, this is not very interesting - we would probably need to blend this data with a meaningful explanation of the *constructorId* and/or *raceId* from another table. For example, the *constructors* table provides descriptive information about each team. (Note, there is also a column (not shown) that gives the URL for the team's Wikipedia page.):

constructorId	constructorRef	name	nationality
1	mclaren	McLaren	British
2	bmw_sauber	BMW Sauber	German
3	williams	Williams	British
4	renault	Renault	French
5	toro_rosso	Toro Rosso	Italian

We might therefore merge the *constructors* and *constructorResults* dataframes using the R `merge()` function:

```
merge(constructors[,1:4], constructorResults, by='constructorId'), n=5)
```

constructorId	constructorRef	name	nationality	constructorResultsId	points	status
1	mclaren	McLaren	British	1	18	14
1	mclaren	McLaren	British	4375	376	15
1	mclaren	McLaren	British	11338	674	0
1	mclaren	McLaren	British	3406	314	13
1	mclaren	McLaren	British	6609	522	0

We will see later on in this chapter how to join data from two or more tables as part of the same SQL query using the SQL `JOIN` command.

The *constructorStandings* also give information keyed by the *constructorId* and *raceId*:

constructorStandingsId	raceId	constructorId	points	position	positionText	wins
1	18	1	14	1	1	1
2	18	2	8	3	3	0
3	18	3	9	2	2	0
4	18	4	5	4	4	0
5	18	5	2	5	5	0

The *driverStandings* table identifies the standings for each driver after each race. Rows are identified (“keyed”) by unique *raceId* and *driverId* combinations:

driverStandingsId	raceId	driverId	points	position	positionText	wins
1	18	1	10	1	1	1
2	18	2	8	2	2	0
3	18	3	6	3	3	0
4	18	4	5	4	4	0
5	18	5	4	5	5	0

In order to find the driver standings at the end of a particular race, we need to find the *raceId* for the corresponding race. This can be done via the *races* table described below.

The *drivers* table gives some descriptive information about each driver. Again, there is an additional column (not shown) that contains a link to the driver's Wikipedia page. The three letter code column is particularly useful as it provides us with a short, recognisable label by means of which we can refer to each driver on many of the charts we'll be producing.

driverId	driverRef	code	forename	surname	dob	nationality
1	hamilton	HAM	Lewis	Hamilton	1985-01-07	British
2	heidfeld	HEI	Nick	Heidfeld	1977-05-10	German
3	rosberg	ROS	Nico	Rosberg	1985-06-27	German
4	alonso	ALO	Fernando	Alonso	1981-07-29	Spanish
5	kovalainen	KOV	Heikki	Kovalainen	1981-10-19	Finnish

The *lapTimes* table is one that we shall pull data from extensively. Keyed by *raceId* and *driverId*, it gives the position of the driver at the end of each lap in a race, along with the laptime for that lap in the form *min:sec.millisecond* as well as the laptime in milliseconds. Lap time data is only available from the 2011 season onward.

raceId	driverId	lap	position	time	milliseconds
841	20	1	1	1:38.109	98109
841	20	2	1	1:33.006	93006
841	20	3	1	1:32.713	92713
841	20	4	1	1:32.803	92803
841	20	5	1	1:32.342	92342

The *pitStops* table provides data about the duration of each individual pit stop and records the time taken between the pit entry and pit exit markers. The duration is given in seconds/milliseconds, as well as the total number of milliseconds. Note that the duration is the sum of the stop time plus the pit loss time.

raceId	driverId	stop	lap	time	duration	milliseconds
841	153	1	1	17:05:23	26.898	26898
841	30	1	1	17:05:52	25.021	25021
841	17	1	11	17:20:48	23.426	23426
841	4	1	12	17:22:34	23.251	23251
841	13	1	13	17:24:10	23.842	23842

The *qualifying* table contains qualifying session times for each driver in each race, along with their position at the end of qualifying. We'll split it into two to make it easier to read.

qualifyId	raceId		driverId	constructorId
1	18		1	1
2	18	9		2
3	18	5		1
4	18	13		6
5	18	2		2
number	position	q1	q2	q3
22	1	1:26.572	1:25.187	1:26.714
4	2	1:26.103	1:25.315	1:26.869
23	3	1:25.664	1:25.452	1:27.079
2	4	1:25.994	1:25.691	1:27.178
3	5	1:25.960	1:25.518	1:27.236

The *races* table contains descriptive information about each actual race. (There is an additional column, not shown, that contains the URL for the Wikipedia page of the actual race):

raceId	year	round	circuitId	name	date	time
1	2009	1	1	Australian Grand Prix	2009-03-29	06:00:00
2	2009	2	2	Malaysian Grand Prix	2009-04-05	09:00:00
3	2009	3	17	Chinese Grand Prix	2009-04-19	07:00:00
4	2009	4	3	Bahrain Grand Prix	2009-04-26	12:00:00
5	2009	5	4	Spanish Grand Prix	2009-05-10	12:00:00

The *results* table provides results data for each car in each race. (The *positionOrder* field is used for ranking drivers who are unclassified by virtue of not completing enough of the race distance.) Let's split the table into three parts so we can see all the columns clearly:

resultId	raceId	driverId	constructorId	number	grid	position
1	18	1	1	22	1	1
2	18	2	2	3	5	2
3	18	3	3	7	7	3
4	18	4	4	5	11	4
5	18	5	1	23	3	5
positionText	positionOrder	points	laps	time	milliseconds	
1		10	58	1:34:50.616	5690616	
2		8	58	+5.478	5696094	
3		6	58	+8.163	5698779	
4		5	58	+17.181	5707797	
5		4	58	+18.014	5708630	
fastestLap	rank	fastestLapTime	fastestLapSpeed	statusId		
39	2	1:27.452	218.300	1		
41	3	1:27.739	217.586	1		
41	5	1:28.090	216.719	1		
58	7	1:28.603	215.464	1		
43	1	1:27.418	218.385	1		

The *time* column gives the overall race time of each race winner in the form *hour:min:sec.millisecond*, the gap to the winner in seconds for other cars finishing the race on the lead lap, and a count of laps behind for any other placed cars.

The *seasons* table provides a link to the Wikipedia page for each season:

year	url
2009	http://en.wikipedia.org/wiki/2009_Formula_One_season
2008	http://en.wikipedia.org/wiki/2008_Formula_One_season
2007	http://en.wikipedia.org/wiki/2007_Formula_One_season
2006	http://en.wikipedia.org/wiki/2006_Formula_One_season
2005	http://en.wikipedia.org/wiki/2005_Formula_One_season

The *status* table gives a natural language description of each status code:

statusId	status
1	Finished
2	Disqualified
3	Accident
4	Collision
5	Engine

At this point, you may be wondering why the data is spread across so many different data tables. The reason is that the data has been *normalised* to minimise the amount of duplication or redundancy in the data, with each table representing data about a particular thing. Key index values - the `Id` elements - are used to link data items across the various tables.

Asking Questions of the *ergast* Data

As you may have noticed, the data we can get from the online *ergast* API comes in a form that we can make sense of immediately. For example, if we get the results of a particular race, we can see the name of the driver, the constructor name, the status at the end of the race, and so on. The full *ergast* API also supports queries that allow us to view data based on just the results associated with a particular driver, year, constructor or circuit, for example, or even based on some combination of those things.

In the section introducing R dataframes, we saw how it is possible to use the routines in the `ergastr-core.R` source file to query the *ergast* API and then run some simple “queries” on the returned R dataframes in order to select certain rows or columns. With access to our own copy of the *ergast* database, we might prefer to call on the full support of the SQL query language to pull back this specific data.

However, there is cost associated with making our own queries on a local copy of the *ergast* database versus calling the *ergast* API directly: the *ergast* API returns datasets that have been created by making queries over several *ergast* database data tables. In order to get a similar response from the database, we need to do one of two things. Either run a query on the database that pulls results back from several tables that we have JOINed together ourselves via the database query; or find an alternative way of combining data that has been pulled from separate database requests into several separate dataframes, for example by merging those separate dataframes.

JOINing Tables Within SQLite Database Queries

If we want to find out the names of the drivers associated with the standings at the end of a particular race, we need to do several things:

- find the *raceId* for the race we are interested in from the *races* table
- get the standings associated with that race from the *driverStandings* table
- get the driver details for each driver from the *drivers* table

Let's see how to write those queries using SQL, and then apply them to our database using the `dbGetQuery()` function.

In the first case, we can attach a series of conditions to a query in which the results are pulled *FROM* a particular table; the rows that are returned are those rows *WHERE* the associated conditions are evaluated as true. The *SELECT* statement declares which columns to return: the * character denotes “all columns”; we could also provide a comma separated list of column names in order to just pull back data from those columns.

```
#Find the raceId for the 2013 British Grand Prix
q='SELECT * FROM races WHERE year=2013 AND name="British Grand Prix"'
#The dbGetQuery() function is provided by the DBI library
dbGetQuery(ergastdb, q)
```

```
##   raceId year round circuitId           name      date      time
## 1     887  2013      8          9 British Grand Prix 2013-06-30 12:00:00
##                                         url
## 1 http://en.wikipedia.org/wiki/2013_British_Grand_Prix
```

The = operator in the SQL query statement is a test for equality, rather than an assignment operator. The logical AND statement allows us to combine selection clauses. As you may expect, a complementary OR statement is also available that lets you select from across several situations, as are numerical comparison operators such as > (*greater than*), <= (*less than or equal to*).

Note that we can order the results from a search by adding *ORDER BY* to the end of the query, followed by one or more column names we wish to sort by. The result is returned in ASCending order by default, but we can also specify a DESCending order.

To limit the number of results that are returned (similar to the R *head()* command), add *LIMIT N* to the end of the query to return at most *N* results. (If you ask for results in any order, the *LIMIT* will return the first *N* results that are found and the query execution will terminate. If you sort the results first, the query needs to execute in full, finding all the results of the query, before then ordering the results.)

What we want to do is get the driver standings at the end of this race. If we just had the *raceId* we could get the standings with the following sort of query:

```
#Obtain the driver standings from a specific race using the race's raceId
q='SELECT * FROM driverStandings WHERE raceId=887'
dbGetQuery(ergastdb, q)
```

	driverStandingsId	raceId	driverId	points	position	positionText	wins
## 1	65734	887	807	6	15	15	0
## 2	65733	887	813	0	16	16	0
## 3	65731	887	817	11	14	14	0
## 4	65730	887	823	0	22	22	0
## 5	65728	887	819	0	20	20	0
## 6	65727	887	824	0	19	19	0
## 7	65725	887	821	0	18	18	0
## 8	65724	887	818	13	12	12	0
## 9	65719	887	16	23	11	11	0
## 10	65718	887	17	87	5	5	0
## 11	65715	887	20	132	1	1	3
## 12	65716	887	13	57	7	7	0
## 13	65717	887	1	89	4	4	0
## 14	65732	887	3	82	6	6	2
## 15	65729	887	820	0	21	21	0
## 16	65726	887	822	0	17	17	0
## 17	65720	887	814	36	8	8	0
## 18	65713	887	8	98	3	3	1
## 19	65714	887	4	111	2	2	2
## 20	65723	887	815	12	13	13	0
## 21	65722	887	154	26	9	9	0
## 22	65721	887	18	25	10	10	0

However, it is often more convenient to be able to ask for a result by the name of a grand prix in a particular year. We can do these by combining the clauses from the previous two queries, further limiting the results to show just the driver standings:

```
#What were the driver standings in the 2013 British Grand Prix?
dbGetQuery(ergastdb,
  'SELECT ds.driverId, ds.points, ds.position
   FROM driverStandings ds JOIN races r
    WHERE ds.raceId=r.raceId AND r.year=2013 AND r.name="British Grand Prix"')

##      driverId points position
## 1        807      6       15
## 2        813      0       16
## 3        817     11       14
## 4        823      0       22
## 5        819      0       20
## 6        824      0       19
## 7        821      0       18
## 8        818     13       12
## 9         16     23       11
## 10       17     87        5
## 11       20    132        1
## 12       13      57        7
## 13        1     89        4
## 14        3     82        6
## 15       820      0       21
## 16       822      0       17
## 17       814     36        8
## 18        8     98        3
## 19        4    111        2
## 20       815     12       13
## 21      154     26        9
## 22       18     25       10
```

In this case, the *JOIN* command declares which tables we want to return data from, providing each with a shorthand name we can use as prefixes to identify columns from the different tables in the *WHERE* part of the query. The first part of the *WHERE* condition is used to merge the rows from the two tables on common elements in their respective *raceId* values, with the second and third conditions limiting which rows to return based on column values in the *races* table.

We can combine *JOIN* statements over multiple tables, not just pairs of tables. For example, to pull in the driver names we can add a further join to the *drivers* table:

```
#Annotate the results of a query on one table by joining with rows from other tables
dbGetQuery(ergastdb,
```

```
'SELECT d.surname, d.code, ds.points, ds.position
FROM driverStandings ds JOIN races r JOIN drivers d
WHERE ds.raceId=r.raceId AND r.year=2013
AND r.name="British Grand Prix" AND d.driverId=ds.driverId')
```

	surname	code	points	position
## 1	Hülkenberg	HUL	6	15
## 2	Maldonado	MAL	0	16
## 3	Ricciardo	RIC	11	14
## 4	van der Garde	VDG	0	22
## 5	Pic	PIC	0	20
## 6	Bianchi	BIA	0	19
## 7	Gutiérrez	GUT	0	18
## 8	Vergne	VER	13	12
## 9	Sutil	SUT	23	11
## 10	Webber	WEB	87	5
## 11	Vettel	VET	132	1
## 12	Massa	MAS	57	7
## 13	Hamilton	HAM	89	4
## 14	Rosberg	ROS	82	6
## 15	Chilton	CHI	0	21
## 16	Bottas	BOT	0	17
## 17	di Resta	DIR	36	8
## 18	Räikkönen	RAI	98	3
## 19	Alonso	ALO	111	2
## 20	Pérez	PER	12	13
## 21	Grosjean	GRO	26	9
## 22	Button	BUT	25	10

Let's just tidy that results table up a little and order by the position, then limit the results to show just the top 3:

```
#Who were the top 3 drivers in the 2013 British Grand Prix?
dbGetQuery(ergastdb,
  'SELECT d.surname, d.code, ds.points, ds.position
   FROM driverStandings ds JOIN races r JOIN drivers d
   WHERE ds.raceId=r.raceId AND r.year=2013
     AND r.name="British Grand Prix" AND d.driverId=ds.driverId
   ORDER BY ds.position ASC
   LIMIT 3')

##      surname code points position
## 1      Vettel  VET     132        1
## 2      Alonso  ALO     111        2
## 3 Räikkönen  RAI      98        3
```

As you can see, we can build up quite complex queries that pull data in from several different tables. The trick to writing the queries is to think clearly about the data you want (that is, the question you want to ask) and then work through the following steps:

- identify which tables that data appears in;
- work out what common key columns would allow you to combine data from the different tables;
- identify what key values give you a way in to the question (for example, in the above case we had to identify the race name and year to get the *raceId*);
- add in any other search limits or ordering terms.

As well as pulling back separate results rows, we can also aggregate the results data. For example, suppose we wanted to count the number of second place finishes Alonso has ever had. We could get the separate instances back as follows:

- find Alonso's *driverId* (so something like *SELECT driverId FROM drivers WHERE code="ALO"*)
- find the races in 2013 where he was in second position (the base query would be something like *SELECT raceId FROM results WHERE driverId=??? AND position=2*. We can find the *driverId* from a JOIN: *SELECT r.raceId FROM results r JOIN drivers d WHERE d.code="ALO" AND r.driverId=d.driverId AND r.position=2*)

We can now count the number of instances as follows:

```
dbGetQuery(ergastdb,
  'SELECT COUNT(*) secondPlaceFinishes
   FROM results r JOIN drivers d
   WHERE d.code="ALO" AND r.driverId=d.driverId AND r.position=2')

##   secondPlaceFinishes
## 1               36
```

We can then go further - who are the top 5 drivers with the greatest number of podium (top 3) finishes, and how many?

```
dbGetQuery(ergastdb,
  'SELECT d.code, d.surname, COUNT(*) podiumFinishes
   FROM results r JOIN drivers d
   WHERE r.driverId=d.driverId AND r.position>=1 AND r.position<=3
   GROUP BY d.surname
   ORDER BY podiumFinishes DESC
   LIMIT 5')

##   code      surname podiumFinishes
## 1  MSC  Schumacher           182
## 2 <NA>    Prost              106
## 3  ALO    Alonso             95
## 4 <NA>    Hill               94
## 5 <NA>    Senna              80
```

One thing we might need to bear in mind when writing queries is that whilst Id elements are guaranteed to be unique identifiers, other values may not be. For example, consider the following search on driver surname:

```
dbGetQuery(ergastdb,
  'SELECT DISTINCT driverRef, surname
   FROM drivers
   WHERE surname=="Hill"')
```

```
##   driverRef surname
## 1 damon_hill    Hill
## 2      hill    Hill
## 3 phil_hill    Hill
```

The surname is an ambiguous identifier, so we need to be a little more precise in our query of podium finishers:

```
dbGetQuery(ergastdb,
  'SELECT d.code, d.driverRef, COUNT(*) podiumFinishes
   FROM results r JOIN drivers d
  WHERE r.driverId=d.driverId AND r.position>=1 AND r.position<=3
    GROUP BY driverRef
   ORDER BY podiumFinishes DESC
  LIMIT 5')

##   code          driverRef podiumFinishes
## 1 MSC michael_schumacher      155
## 2 <NA>           prost        106
## 3 ALO            alonso       95
## 4 <NA>           senna        80
## 5 RAI            raikkonen     77
```

In this case, the COUNT operator is applied over groups of rows returned from the GROUP BY operator. Other summarising operators are also available. For example, MAX() returns the maximum value from a group of values, MIN() the minimum, SUM() the sum of values, and so on. (See also: SQLite aggregate functions³⁶.)

Exercise

See if you can work out what queries can be used to generate some of the other results tables described on the List of Formula One driver records³⁷ Wikipedia page.

³⁶http://sqlite.org/lang_aggfunc.html

³⁷http://en.wikipedia.org/wiki/List_of_Formula_One_driver_records



Exercise

The *ergast* API offers several “high level” patterns for querying F1 results data via a URL that you can construct yourself.

Explore some of the queries you can make on the *ergast* website. Choose two or three of these rich data requests and see if you can create equivalent queries onto the *ergast* SQLite database. Check the results of running your query against the results returned from the *ergast* API.

Nested SELECTs and TEMPORARY VIEWS

Sometimes we may want to run a SELECT query that draws on the results of another query. For example, consider this query, which finds the distinct *driverIds* for drivers competing in the 2013 season:

```
#What drivers competed in the 2013 season?
dbGetQuery(ergastdb,
  'SELECT DISTINCT ds.driverId
   FROM driverStandings ds JOIN races r
   WHERE r.year=2013 AND r.raceId=ds.raceId')

##      driverId
## 1          8
## 2          4
## 3         20
## 4         13
## 5          1
## 6         17
## 7         16
## 8        814
## 9         18
## 10        154
## 11        815
## 12        818
## 13        821
## 14        822
## 15        824
## 16        819
## 17        820
```

```
## 18      823
## 19      817
## 20      3
## 21      813
## 22      807
## 23      5
```

We can pull on this list of *driverIds* to return the full driver information from the *drivers* for each driver competing in 2013.

```
dbGetQuery(ergastdb,
  'SELECT * FROM drivers
  WHERE driverId IN
  (SELECT DISTINCT ds.driverId
  FROM driverStandings ds JOIN races r
  WHERE r.year=2013 AND r.raceId=ds.raceId ) ')
```

	driverId	driverRef	code	forename	surname	dob
## 1	1	hamilton	HAM	Lewis	Hamilton	1985-01-07
## 2	3	rosberg	ROS	Nico	Rosberg	1985-06-27
## 3	4	alonso	ALO	Fernando	Alonso	1981-07-29
## 4	5	kovalainen	KOV	Heikki	Kovalainen	1981-10-19
## 5	8	raikkonen	RAI	Kimi	Räikkönen	1979-10-17
## 6	13	massa	MAS	Felipe	Massa	1981-04-25
## 7	16	sutil	SUT	Adrian	Sutil	1983-01-11
## 8	17	webber	WEB	Mark	Webber	1976-08-27
## 9	18	button	BUT	Jenson	Button	1980-01-19
## 10	20	vettel	VET	Sebastian	Vettel	1987-07-03
## 11	154	grosjean	GRO	Romain	Grosjean	1986-04-17
## 12	807	hulkenberg	HUL	Nico	Hülkenberg	1987-08-19
## 13	813	maldonado	MAL	Pastor	Maldonado	1985-03-09
## 14	814	resta	DIR	Paul	di Resta	1986-04-16
## 15	815	perez	PER	Sergio	Pérez	1990-01-26
## 16	817	ricciardo	RIC	Daniel	Ricciardo	1989-07-01
## 17	818	vergne	VER	Jean-Éric	Vergne	1990-04-25
## 18	819	pic	PIC	Charles	Pic	1990-02-15
## 19	820	chilton	CHI	Max	Chilton	1991-04-21
## 20	821	gutierrez	GUT	Esteban	Gutiérrez	1991-08-05
## 21	822	bottas	BOT	Valtteri	Bottas	1989-08-29
## 22	823	garde	VDG	Giedo van der Garde	Giedo van der Garde	1985-04-25

```

## 23      824 jules_bianchi  BIA      Jules      Bianchi 1989-08-03
##   nationality                               url
## 1   British          http://en.wikipedia.org/wiki/Lewis_Hamilton
## 2   German           http://en.wikipedia.org/wiki/Nico_Rosberg
## 3   Spanish          http://en.wikipedia.org/wiki/Fernando_Alonso
## 4   Finnish          http://en.wikipedia.org/wiki/Heikki_Kovalainen
## 5   Finnish          http://en.wikipedia.org/wiki/Kimi_R%C3%A4ikk%C3%B6nen
## 6   Brazilian        http://en.wikipedia.org/wiki/Felipe_Massa
## 7   German           http://en.wikipedia.org/wiki/Adrian_Sutil
## 8   Australian       http://en.wikipedia.org/wiki/Mark_Webber
## 9   British          http://en.wikipedia.org/wiki/Jenson_Button
## 10  German           http://en.wikipedia.org/wiki/Sebastian_Vettel
## 11  French           http://en.wikipedia.org/wiki/Romain_Grosjean
## 12  German           http://en.wikipedia.org/wiki/Nico_H%C3%BClkenberg
## 13  Venezuelan       http://en.wikipedia.org/wiki/Pastor_Maldonado
## 14  Scottish          http://en.wikipedia.org/wiki/Paul_di_Resta
## 15  Mexican          http://en.wikipedia.org/wiki/Sergio_P%C3%A9rez
## 16  Australian       http://en.wikipedia.org/wiki/Daniel_Ricciardo
## 17  French           http://en.wikipedia.org/wiki/Jean-%C3%89ric_Vergne
## 18  French           http://en.wikipedia.org/wiki/Charles_Pic
## 19  British          http://en.wikipedia.org/wiki/Max_Chilton
## 20  Mexican          http://en.wikipedia.org/wiki/Esteban_Guti%C3%A9rrez
## 21  Finnish          http://en.wikipedia.org/wiki/Valtteri_Bottas
## 22  Dutch            http://en.wikipedia.org/wiki/Giedo_van_der_Garde
## 23  French           http://en.wikipedia.org/wiki/Jules_Bianchi

```

To support the reuse of this dataset, we can CREATE a TEMPORARY VIEW that acts like a database table containing this data.

```

dbGetQuery(ergastdb,
  'CREATE TEMPORARY VIEW drivers2013 AS
    SELECT * FROM drivers
    WHERE driverId IN
      (SELECT DISTINCT ds.driverId
        FROM driverStandings ds JOIN races r
        WHERE r.year=2013 AND r.raceId=ds.raceId ) ')

```

We can then run SELECT queries FROM this view as if it were any other data table.

```
dbGetQuery(ergastdb, 'SELECT * FROM drivers2013')
```

##	driverId	driverRef	code	forename	surname	dob
## 1	1	hamilton	HAM	Lewis	Hamilton	1985-01-07
## 2	3	rosberg	ROS	Nico	Rosberg	1985-06-27
## 3	4	alonso	ALO	Fernando	Alonso	1981-07-29
## 4	5	kovalainen	KOV	Heikki	Kovalainen	1981-10-19
## 5	8	raikkonen	RAI	Kimi	Räikkönen	1979-10-17
## 6	13	massa	MAS	Felipe	Massa	1981-04-25
## 7	16	sutil	SUT	Adrian	Sutil	1983-01-11
## 8	17	webber	WEB	Mark	Webber	1976-08-27
## 9	18	button	BUT	Jenson	Button	1980-01-19
## 10	20	vettel	VET	Sebastian	Vettel	1987-07-03
## 11	154	grosjean	GRO	Romain	Grosjean	1986-04-17
## 12	807	hulkenberg	HUL	Nico	Hülkenberg	1987-08-19
## 13	813	maldonado	MAL	Pastor	Maldonado	1985-03-09
## 14	814	resta	DIR	Paul	di Resta	1986-04-16
## 15	815	perez	PER	Sergio	Pérez	1990-01-26
## 16	817	ricciardo	RIC	Daniel	Ricciardo	1989-07-01
## 17	818	vergne	VER	Jean-Éric	Vergne	1990-04-25
## 18	819	pic	PIC	Charles	Pic	1990-02-15
## 19	820	chilton	CHI	Max	Chilton	1991-04-21
## 20	821	gutierrez	GUT	Esteban	Gutiérrez	1991-08-05
## 21	822	bottas	BOT	Valtteri	Bottas	1989-08-29
## 22	823	garde	VDG	Giedo van der Garde	van der Garde	1985-04-25
## 23	824	jules_bianchi	BIA	Jules	Bianchi	1989-08-03
##		nationality				url
## 1		British				http://en.wikipedia.org/wiki/Lewis_Hamilton
## 2		German				http://en.wikipedia.org/wiki/Nico_Rosberg
## 3		Spanish				http://en.wikipedia.org/wiki/Fernando_Alonso
## 4		Finnish				http://en.wikipedia.org/wiki/Heikki_Kovalainen
## 5		Finnish				http://en.wikipedia.org/wiki/Kimi_R%C3%A4ikk%C3%B6nen
## 6		Brazilian				http://en.wikipedia.org/wiki/Felipe_Massa
## 7		German				http://en.wikipedia.org/wiki/Adrian_Sutil
## 8		Australian				http://en.wikipedia.org/wiki/Mark_Webber
## 9		British				http://en.wikipedia.org/wiki/Jenson_Button
## 10		German				http://en.wikipedia.org/wiki/Sebastian_Vettel
## 11		French				http://en.wikipedia.org/wiki/Romain_Grosjean
## 12		German				http://en.wikipedia.org/wiki/Nico_H%C3%BClkenberg
## 13		Venezuelan				http://en.wikipedia.org/wiki/Pastor_Maldonado
## 14		Scottish				http://en.wikipedia.org/wiki/Paul_di_Resta
## 15		Mexican				http://en.wikipedia.org/wiki/Sergio_P%C3%A9rez

```

## 16 Australian      http://en.wikipedia.org/wiki/Daniel_Ricciardo
## 17 French         http://en.wikipedia.org/wiki/Jean-%C3%89ric_Vergne
## 18 French         http://en.wikipedia.org/wiki/Charles_Pic
## 19 British        http://en.wikipedia.org/wiki/Max_Chilton
## 20 Mexican        http://en.wikipedia.org/wiki/Esteban_Guti%C3%A9rrez
## 21 Finnish        http://en.wikipedia.org/wiki/Valtteri_Bottas
## 22 Dutch          http://en.wikipedia.org/wiki/Giedo_van_der_Garde
## 23 French         http://en.wikipedia.org/wiki/Jules_Bianchi

```

More Examples of Merging Dataframes in R

As well as running compound queries and multiple joins via SQL queries, we can of course further manipulate data that is returned from a SQL query using R dataframe operations. For example, here are some examples of merging R dataframes pulled back from separate queries onto the *ergast* database.

For example, to find the names of the winners of the 2013 races, first we need to get the *raceIDs* from the *races* table:

```

#Limit the display of columns to the first three in the dataframe
raceIDs=races[races['year']=='2013',1:3]
raceIDs

```

```

##      raceId year round
## 879     880 2013     1
## 880     881 2013     2
## 881     882 2013     3
## 882     883 2013     4
## 883     884 2013     5
## 884     885 2013     6
## 885     886 2013     7
## 886     887 2013     8
## 887     888 2013     9
## 888     890 2013    10
## 889     891 2013    11
## 890     892 2013    12
## 891     893 2013    13
## 892     894 2013    14
## 893     895 2013    15
## 894     896 2013    16

```

```
## 895    897 2013    17
## 896    898 2013    18
## 897    899 2013    19
```

The next thing we need to do is pull in information about the winners of each race in 2013. The winners are in the `results` table. We want to pull in information about the person in the first position in each race, but to make sure we match on the correct thing we need to see whether or not we want to match on 1 as a digit or as a character. We can ask R what sort of thing it thinks is the type of each column in the `results` table:

```
str(results)
```

```
## 'data.frame':      22129 obs. of  18 variables:
## $ resultId       : int  1 2 3 4 5 6 7 8 9 10 ...
## $ raceId         : int  18 18 18 18 18 18 18 18 18 18 ...
## $ driverId       : int  1 2 3 4 5 6 7 8 9 10 ...
## $ constructorId  : int  1 2 3 4 1 3 5 6 2 7 ...
## $ number          : int  22 3 7 5 23 8 14 1 4 12 ...
## $ grid            : int  1 5 7 11 3 13 17 15 2 18 ...
## $ position        : int  1 2 3 4 5 6 7 8 NA NA ...
## $ positionText   : chr  "1" "2" "3" "4" ...
## $ positionOrder  : int  1 2 3 4 5 6 7 8 9 10 ...
## $ points          : num  10 8 6 5 4 3 2 1 0 0 ...
## $ laps             : int  58 58 58 58 57 55 53 47 43 ...
## $ time             : chr  "1:34:50.616" "+5.478" "+8.163" "+17.181" ...
## $ milliseconds   : int  5690616 5696094 5698779 5707797 5708630 NA NA NA NA ...
## $ fastestLap     : int  39 41 41 58 43 50 22 20 15 23 ...
## $ rank             : int  2 3 5 7 1 14 12 4 9 13 ...
## $ fastestLapTime : chr  "1:27.452" "1:27.739" "1:28.090" "1:28.603" ...
## $ fastestLapSpeed: chr  "218.300" "217.586" "216.719" "215.464" ...
## $ statusId        : int  1 1 1 1 1 11 5 5 4 3 ...
```

So, do we want to test on `position==1` (an integer), `positionText=="1"` (a character string), or `positionOrder==1` (another integer)? Looking carefully at the structure of the table, we see that the `position` element is occasionally undetermined (`NA`); that is, no position is recorded. If we test whether or not `1==NA`, we get `NA` rather than `FALSE` as a result. As the simple filter brings back results if the answer is not `FALSE`, we would get a false positive match if the position is `NA`, rather than 1. That is, if we were to trivially filter the dataframe by testing for `position==1`, it would pull back results where the position is either 1 or `NA`.

```
#To trap the filter against returning NA results, we might use something like:
#head(results[results['position']==1 & !is.na(results['position'])],)

firstPositions=results[results['positionOrder']=="1",]
```

We can now merge the first place results dataframe with the 2013 raceIDs dataframe. To show there's no sleight of hand involved, here are the columns we have in original *raceIDs* dataframe:

```
colnames(raceIDs)

## [1] "raceId" "year"    "round"
```

Now let's see what happens when we merge in *from the right* some matching data from the *firstPositions* dataframe:

```
raceIDs=merge(raceIDs, firstPositions, by='raceId')
colnames(raceIDs)

## [1] "raceId"      "year"       "round"
## [4] "resultId"   "driverId"   "constructorId"
## [7] "number"     "grid"      "position"
## [10] "positionText" "positionOrder" "points"
## [13] "laps"        "time"      "milliseconds"
## [16] "fastestLap"   "rank"      "fastestLapTime"
## [19] "fastestLapSpeed" "statusId"
```

We can also pull in information about the drivers:

```
raceIDs=merge(raceIDs, drivers, by='driverId')
colnames(raceIDs)
```

```

## [1] "driverId"      "raceId"        "year"
## [4] "round"          "resultId"       "constructorId"
## [7] "number"          "grid"           "position"
## [10] "positionText"   "positionOrder"  "points"
## [13] "laps"            "time"           "milliseconds"
## [16] "fastestLap"     "rank"           "fastestLapTime"
## [19] "fastestLapSpeed" "statusId"       "driverRef"
## [22] "code"            "forename"       "surname"
## [25] "dob"             "nationality"   "url"

```

And the constructors...

```

raceIDs=merge(raceIDs, constructors, by='constructorId')
colnames(raceIDs)

```

```

## [1] "constructorId"  "driverId"       "raceId"
## [4] "year"           "round"          "resultId"
## [7] "number"          "grid"           "position"
## [10] "positionText"   "positionOrder"  "points"
## [13] "laps"            "time"           "milliseconds"
## [16] "fastestLap"     "rank"           "fastestLapTime"
## [19] "fastestLapSpeed" "statusId"       "driverRef"
## [22] "code"            "forename"       "surname"
## [25] "dob"             "nationality.x" "url.x"
## [28] "constructorRef"  "name"           "nationality.y"
## [31] "url.y"

```

Note that where column names collide, an additional suffix is added to the column names, working “from the left”. So for example, there was a collision on the column name *nationality*, so new column names are derived to break that collision. *nationality.x* now refers to the nationality column from the left hand table in the merge (that is, corresponding to the driver nationality, which we had already merged into the *raceIDs* dataframe) and *nationality.y* refers to the nationality of the constructor.

Let’s also pull in the races themselves...

```

raceIDs=merge(raceIDs, races, by='raceId')
colnames(raceIDs)

```

```
## [1] "raceId"           "constructorId"    "driverId"
## [4] "year.x"           "round.x"          "resultId"
## [7] "number"            "grid"              "position"
## [10] "positionText"     "positionOrder"    "points"
## [13] "laps"               "time.x"            "milliseconds"
## [16] "fastestLap"        "rank"              "fastestLapTime"
## [19] "fastestLapSpeed"   "statusId"         "driverRef"
## [22] "code"               "forename"          "surname"
## [25] "dob"                "nationality.x"   "url.x"
## [28] "constructorRef"    "name.x"            "nationality.y"
## [31] "url.y"              "year.y"            "round.y"
## [34] "circuitId"          "name.y"            "date"
## [37] "time.y"             "url"
```

We can also merge across multiple columns, in which case a merge occurs when the values across all the specified merge columns match across the rows in the two separate merge tables.

Hopefully you get the idea?! What this means is that we can mix and match the way we work with data in a way that is most convenient for us.

Summary

In this chapter we have seen how we can write a wide variety of powerful queries over the *ergast* database, in this case managed via SQLite. (The same SQL queries should work equally well if the data is being pulled from a MySQL database, or PostgreSQL database.)

In particular, we have seen (albeit briefly) how to:

- retrieve data columns from a database table using `SELECT .. FROM ..`, along with the `DISTINCT` modifier to retrieve unique values or combinations of values
- use the `WHERE` operator to filter rows and match row values from separate tables
- use the `JOIN` statement to support the retrieval of data from multiple columns
- use the `GROUP BY` operator to group rows (and the `COUNT()` operator in the `SELECT` statement to count the number of rows in each group)
- use the `HAVING` operator to filter results based on GROUP operations
- use the `IN` statement to allow selection of data based on the results on another `SELECT` statement (a “nested” `SELECT`)

- generate a temporary view that acts like a custom datatable using CREATE TEMPORARY VIEW

We have also seen how we can manipulate or combine dataframes returned from one or more SQL queries using the R merge command.

Exercises



Practice your SQL skills by coming up with a range of trivia questions about historical Formula One statistics and then seeing whether you can write one or more SQL queries over the *ergast* database that will answer each question.

If you feel that SQL is easier to use than the native R based filtering and sorting operations, you might find the `sqldf` package on CRAN useful. This package allows you to execute SQL style queries over the contents of a dataframe.

Addendum

You can inspect the schemata for the tables included in the *ergast* database by running the following query on the SQLite3 database: `SELECT sql FROM sqlite_master;`

```
CREATE TABLE "circuits" (
    "circuitId" int(11) NOT NULL ,
    "circuitRef" varchar(255) NOT NULL DEFAULT '',
    "name" varchar(255) NOT NULL DEFAULT '',
    "location" varchar(255) DEFAULT NULL,
    "country" varchar(255) DEFAULT NULL,
    "lat" float DEFAULT NULL,
    "lng" float DEFAULT NULL,
    "alt" int(11) DEFAULT NULL,
    "url" varchar(255) NOT NULL DEFAULT '',
    PRIMARY KEY ("circuitId")
)

CREATE TABLE "constructorResults" (
    "constructorResultsId" int(11) NOT NULL ,
    "raceId" int(11) NOT NULL DEFAULT '0',
    "constructorId" int(11) NOT NULL DEFAULT '0',
    "points" float DEFAULT NULL,
```

```
"status" varchar(255) DEFAULT NULL,  
PRIMARY KEY ("constructorResultsId")  
)  
  
CREATE TABLE "constructorStandings" (  
    "constructorStandingsId" int(11) NOT NULL ,  
    "raceId" int(11) NOT NULL DEFAULT '0',  
    "constructorId" int(11) NOT NULL DEFAULT '0',  
    "points" float NOT NULL DEFAULT '0',  
    "position" int(11) DEFAULT NULL,  
    "positionText" varchar(255) DEFAULT NULL,  
    "wins" int(11) NOT NULL DEFAULT '0',  
    PRIMARY KEY ("constructorStandingsId")  
)  
  
CREATE TABLE "constructors" (  
    "constructorId" int(11) NOT NULL ,  
    "constructorRef" varchar(255) NOT NULL DEFAULT '',  
    "name" varchar(255) NOT NULL DEFAULT '',  
    "nationality" varchar(255) DEFAULT NULL,  
    "url" varchar(255) NOT NULL DEFAULT '',  
    PRIMARY KEY ("constructorId")  
)  
  
CREATE TABLE "driverStandings" (  
    "driverStandingsId" int(11) NOT NULL ,  
    "raceId" int(11) NOT NULL DEFAULT '0',  
    "driverId" int(11) NOT NULL DEFAULT '0',  
    "points" float NOT NULL DEFAULT '0',  
    "position" int(11) DEFAULT NULL,  
    "positionText" varchar(255) DEFAULT NULL,  
    "wins" int(11) NOT NULL DEFAULT '0',  
    PRIMARY KEY ("driverStandingsId")  
)  
  
CREATE TABLE "drivers" (  
    "driverId" int(11) NOT NULL ,  
    "driverRef" varchar(255) NOT NULL DEFAULT '',  
    "code" varchar(3) DEFAULT NULL,  
    "forename" varchar(255) NOT NULL DEFAULT '',  
    "surname" varchar(255) NOT NULL DEFAULT '',  
    "dob" date DEFAULT NULL,  
    "nationality" varchar(255) DEFAULT NULL,
```

```
"url" varchar(255) NOT NULL DEFAULT '' ,  
PRIMARY KEY ("driverId")  
)  
  
CREATE TABLE "lapTimes" (  
    "raceId" int(11) NOT NULL ,  
    "driverId" int(11) NOT NULL ,  
    "lap" int(11) NOT NULL ,  
    "position" int(11) DEFAULT NULL ,  
    "time" varchar(255) DEFAULT NULL ,  
    "milliseconds" int(11) DEFAULT NULL ,  
    PRIMARY KEY ("raceId","driverId","lap")  
)  
  
CREATE TABLE "pitStops" (  
    "raceId" int(11) NOT NULL ,  
    "driverId" int(11) NOT NULL ,  
    "stop" int(11) NOT NULL ,  
    "lap" int(11) NOT NULL ,  
    "time" time NOT NULL ,  
    "duration" varchar(255) DEFAULT NULL ,  
    "milliseconds" int(11) DEFAULT NULL ,  
    PRIMARY KEY ("raceId","driverId","stop")  
)  
  
CREATE TABLE "qualifying" (  
    "qualifyId" int(11) NOT NULL ,  
    "raceId" int(11) NOT NULL DEFAULT '0' ,  
    "driverId" int(11) NOT NULL DEFAULT '0' ,  
    "constructorId" int(11) NOT NULL DEFAULT '0' ,  
    "number" int(11) NOT NULL DEFAULT '0' ,  
    "position" int(11) DEFAULT NULL ,  
    "q1" varchar(255) DEFAULT NULL ,  
    "q2" varchar(255) DEFAULT NULL ,  
    "q3" varchar(255) DEFAULT NULL ,  
    PRIMARY KEY ("qualifyId")  
)  
  
CREATE TABLE "races" (  
    "raceId" int(11) NOT NULL ,  
    "year" int(11) NOT NULL DEFAULT '0' ,  
    "round" int(11) NOT NULL DEFAULT '0' ,  
    "circuitId" int(11) NOT NULL DEFAULT '0' ,
```

```
"name" varchar(255) NOT NULL DEFAULT '',
"date" date NOT NULL DEFAULT '0000-00-00',
"time" time DEFAULT NULL,
"url" varchar(255) DEFAULT NULL,
PRIMARY KEY ("raceId")
)

CREATE TABLE "results" (
"resultId" int(11) NOT NULL ,
"raceId" int(11) NOT NULL DEFAULT '0',
"driverId" int(11) NOT NULL DEFAULT '0',
"constructorId" int(11) NOT NULL DEFAULT '0',
"number" int(11) NOT NULL DEFAULT '0',
"grid" int(11) NOT NULL DEFAULT '0',
"position" int(11) DEFAULT NULL,
"positionText" varchar(255) NOT NULL DEFAULT '',
"positionOrder" int(11) NOT NULL DEFAULT '0',
"points" float NOT NULL DEFAULT '0',
"laps" int(11) NOT NULL DEFAULT '0',
"time" varchar(255) DEFAULT NULL,
"milliseconds" int(11) DEFAULT NULL,
"fastestLap" int(11) DEFAULT NULL,
"rank" int(11) DEFAULT '0',
"fastestLapTime" varchar(255) DEFAULT NULL,
"fastestLapSpeed" varchar(255) DEFAULT NULL,
"statusId" int(11) NOT NULL DEFAULT '0',
PRIMARY KEY ("resultId")
)

CREATE TABLE "seasons" (
"year" int(11) NOT NULL DEFAULT '0',
"url" varchar(255) NOT NULL DEFAULT '',
PRIMARY KEY ("year")
)

CREATE TABLE "status" (
"statusId" int(11) NOT NULL ,
"status" varchar(255) NOT NULL DEFAULT '',
PRIMARY KEY ("statusId")
)
```

Data Scrapped from the Formula One Website (Pre-2015)

Until a redesign of the formula1.com website for the 2015 season, data was available detailing results and session data from the current, as well as previous, sessions. The original website, including the results data, remains available on the Internet Archive Wayback Machine³⁸.

Data originally scraped from the Formula One website is available as a simple SQLite database (accessible from the *wranglingf1datawithr* repository³⁹ as *scraperwiki.sqlite*. A second, improved scrape into a newly formatted database - *f1com_results_archive.sqlite* - was completed at the end of 2014.

The current version of this book utilises data from *both* SQLite databases.

Format of the Original *scraperwiki.sqlite* Database

The original database contained the following tables:

```
library(DBI)
f1 =dbConnect(RSQLite::SQLite(), './scraperwiki.sqlite')

#dbGetQuery(f1, 'SET NAMES utf8;')
dbListTables(f1)

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

The pNResults tables record the classification for each of the three practice sessions (N=1,2,3) run over each race weekend:

³⁸<https://archive.org/web/>

³⁹<https://github.com/psychimedia/wranglingf1datawithr>

```
dbGetQuery(f1, ("SELECT * FROM p1Results LIMIT 5"))
```

```
##   driverNum    time laps year natGap   gap      race pos      driverName
## 1          3 87.560  11 2012  0.000 0.000 AUSTRALIA  1 Jenson Button
## 2          4 87.805  14 2012  0.245 0.245 AUSTRALIA  2 Lewis Hamilton
## 3          7 88.235  17 2012  0.675 0.675 AUSTRALIA  3 Michael Schumacher
## 4          5 88.360  21 2012  0.800 0.800 AUSTRALIA  4 Fernando Alonso
## 5          2 88.467  21 2012  0.907 0.907 AUSTRALIA  5 Mark Webber
##           team  natTime
## 1 McLaren-Mercedes 1:27.560
## 2 McLaren-Mercedes 1:27.805
## 3          Mercedes 1:28.235
## 4          Ferrari 1:28.360
## 5 Red Bull Racing-Renault 1:28.467
```

The practice session results include the name of each driver, their classification within that session, their team, the number of laps they completed, their best laptime as a natural time (using the format *minutes:seconds.milliseconds*) and as a time in seconds and milliseconds, and the natural gap (*natgap*)/*gap* (the *natgap* as seconds/milliseconds) to the best time in the session.

The pNSectors tables contain the best sector times recorded by each driver in each practice session (N=1,2,3), and the qualiSectors table the best sector times from qualifying:

```
dbGetQuery(f1, ("SELECT * FROM p1Sectors LIMIT 5"))
```

```
##   sector      race      driverName year sectortime driverNum pos
## 1     1 AUSTRALIA Jenson Button 2012 29.184      3  1
## 2     1 AUSTRALIA Lewis Hamilton 2012 29.190      4  2
## 3     1 AUSTRALIA Nico Rosberg 2012 29.514      8  3
## 4     1 AUSTRALIA Michael Schumacher 2012 29.583      7  4
## 5     1 AUSTRALIA Mark Webber 2012 29.645      2  5
```

The pNSpeeds table records the fastest speed recorded by each driver in a given practice session (N=1,2,3), with the qualiSpeeds table given the best speeds achieved during qualifying:

```
dbGetQuery(f1, ("SELECT * FROM p1Speeds LIMIT 5"))
```

	timeOfDay	race	driverName	year	driverNum	qspeed	pos
## 1	13:58:43	AUSTRALIA	Kimi Räikkönen	2012	9	313.9	1
## 2	13:32:04	AUSTRALIA	Michael Schumacher	2012	7	312.8	2
## 3	13:59:30	AUSTRALIA	Romain Grosjean	2012	10	312.8	3
## 4	13:52:10	AUSTRALIA	Daniel Ricciardo	2012	16	312.5	4
## 5	13:34:45	AUSTRALIA	Nico Rosberg	2012	8	312.0	5

We can combine the sector times from each sector by binding the rows from queries onto separate session tables together, as well as creating appropriately named dataframes in the global scope. To distinguish in which session the best sector times were set, we add a new column that specifies the session in which the time was recorded; to generalise the underlying function, we pass in the partial name of the data table according to the session data we want to return (*Sectors* or *Speeds*):

```
sessionData=function(race,year/sessionType='Sectors',sessions=c('p1','p2','p3','quali')){
  df=data.frame()
  if (length(sessions)>=1)
    for (session in sessions) {
      sessionName=paste(session,sessionType,sep=' ')
      q=paste("SELECT * FROM ", sessionName, " WHERE race=UPPER('",race,"') AND year=",
      year,"'", sep="")
      #print(q)
      #The following line creates appropriately named dataframes in the global scope
      #containing the results of each separate query
      assign(sessionName,dbGetQuery(conn=f1, statement=q), envir = .GlobalEnv)
      df.tmp=get(sessionName)
      df.tmp['session']=session
      df=rbind(df,df.tmp)
    }
  df
}

sectorTimes=function(race,year,sessions=c('p1','p2','p3','quali')){
  sessionData(race,year, 'Sectors',sessions)
}

sessionSpeeds=function(race,year,sessions=c('p1','p2','p3','quali')){
```

```

sessionData(race,year,'Speeds',sessions)
}

#Usage:
#Get all the practice and qualifying session sector times for a specific race
#df=sectorTimes('AUSTRALIA','2012')

#Get P3 and Quali sector times
#df=sectorTimes('AUSTRALIA','2012',c('p3','quali'))

#Get the speeds from the quali session.
#df=sessionSpeeds('Australia','2012','quali')

#This function can be found in the file f1comdataR-core.R

head(sessionSpeeds('Australia','2012','quali'),n=5)

##   timeOfDay      race    driverName year driverNum qspeed pos session
## 1 17:16:37 AUSTRALIA Sebastian Vettel 2012         1 303.7 19  quali
## 2 17:16:17 AUSTRALIA Romain Grosjean 2012        10 310.2  7  quali
## 3 17:16:08 AUSTRALIA Paul di Resta 2012        11 308.1 12  quali
## 4 17:04:53 AUSTRALIA Nico Hulkenberg 2012       12 308.0 14  quali
## 5 17:06:58 AUSTRALIA Kamui Kobayashi 2012      14 312.5  5  quali

```

The `qualiResults` table is more elaborate than the results tables for the practice sessions, because it includes the best lap time recorded in each qualifying session as well as the number of laps completed across qualifying.

```
dbGetQuery(f1, ("SELECT * FROM qualiResults LIMIT 5"))
```

```

##   q1time driverNum pos q1natTime q2time      race q3time year q2natTime
## 1 86.800          4  1 1:26.800 85.626 AUSTRALIA 84.922 2012 1:25.626
## 2 86.832          3  2 1:26.832 85.663 AUSTRALIA 85.074 2012 1:25.663
## 3 86.498          10 3 1:26.498 85.845 AUSTRALIA 85.302 2012 1:25.845
## 4 86.586          7  4 1:26.586 85.571 AUSTRALIA 85.336 2012 1:25.571
## 5 87.117          2  5 1:27.117 86.297 AUSTRALIA 85.651 2012 1:26.297
##   q3natTime              team qlaps      driverName
## 1 1:24.922    McLaren-Mercedes 14    Lewis Hamilton
## 2 1:25.074    McLaren-Mercedes 15    Jenson Button
## 3 1:25.302    Lotus-Renault 21    Romain Grosjean
## 4 1:25.336        Mercedes 18 Michael Schumacher
## 5 1:25.651 Red Bull Racing-Renault 17    Mark Webber

```

The race results include the race time for the winner and the total gap to each of the following drivers (or the number of laps they were behind). For drivers that did not finish, the status is returned. The `laps` column gives the number of race laps completed by each driver:

```
dbGetQuery(f1, ("SELECT * FROM raceResults LIMIT 5"))
```

```

##      race laps driverNum pos points grid      driverName raceNum year
## 1 AUSTRALIA  58          3  1     25  2    Jenson Button 1 2012
## 2 AUSTRALIA  58          1  2     18  6 Sebastian Vettel 1 2012
## 3 AUSTRALIA  58          4  3     15  1    Lewis Hamilton 1 2012
## 4 AUSTRALIA  58          2  4     12  5    Mark Webber 1 2012
## 5 AUSTRALIA  58          5  5     10 12 Fernando Alonso 1 2012
##              team timeOrRetired
## 1    McLaren-Mercedes 1:34:09.565
## 2 Red Bull Racing-Renault    +2.1 secs
## 3    McLaren-Mercedes    +4.0 secs
## 4 Red Bull Racing-Renault    +4.5 secs
## 5        Ferrari    +21.5 secs

```

The `racePits` table summarises pit stop activity, with one line for each pit stop including the lap number the stop was taken on and the time of day. The pit loss time for each stop is given along with the cumulative pit loss time.

```
dbGetQuery(f1, ("SELECT * FROM racePits LIMIT 5"))
```

```

##   natPitTime totalPitTime      race natTotalPitTime driverNum stops
## 1    24.599     24.599 AUSTRALIA      24.599       19     1
## 2    32.319     32.319 AUSTRALIA      32.319       16     1
## 3    22.313     22.313 AUSTRALIA      22.313        6     1
## 4    23.203     23.203 AUSTRALIA      23.203        8     1
## 5    22.035     22.035 AUSTRALIA      22.035        5     1
##   pitTime      driverName raceNum year           team lap timeOfDay
## 1 24.599      Bruno Senna      1 2012 Williams-Renault  1 17:05:23
## 2 32.319      Daniel Ricciardo  1 2012 STR-Ferrari    1 17:05:35
## 3 22.313      Felipe Massa      1 2012 Ferrari       11 17:21:08
## 4 23.203      Nico Rosberg      1 2012 Mercedes      12 17:22:31
## 5 22.035      Fernando Alonso    1 2012 Ferrari       13 17:24:04

```

The `raceFastLaps` table records the race lap on which each driver recorded their fastest laptime, along with that laptime and the average speed round the lap.

```
dbGetQuery(f1, ("SELECT * FROM raceFastlaps LIMIT 5"))
```

```

##   timeOfDay lap driverNum pos      race stime raceNum year
## 1 18:34:37  56         3   1 AUSTRALIA 89.187      1 2012
## 2 18:36:10  57         1   2 AUSTRALIA 89.417      1 2012
## 3 18:36:12  57         2   3 AUSTRALIA 89.438      1 2012
## 4 18:36:11  57         4   4 AUSTRALIA 89.538      1 2012
## 5 18:30:22  53        18   5 AUSTRALIA 90.254      1 2012
##           team   speed  natTime      driverName
## 1 McLaren-Mercedes 214.053 1:29.187 Jenson Button
## 2 Red Bull Racing-Renault 213.503 1:29.417 Sebastian Vettel
## 3 Red Bull Racing-Renault 213.452 1:29.438 Mark Webber
## 4 McLaren-Mercedes 213.214 1:29.538 Lewis Hamilton
## 5 Williams-Renault 211.523 1:30.254 Pastor Maldonado

```

Format of the `f1com_results_archive.sqlite` Database

The revised `f1com_results_archive.sqlite` database contains the following tables.

```
f1archive = dbConnect(RSQLite::SQLite(), './f1com_results_archive.sqlite')

dbListTables(f1archive)

## [1] "QualiResultsto2005" "Sectors"           "Speeds"
## [4] "pResults"          "qualiResults"      "raceFastlaps"
## [7] "racePits"          "raceResults"       "
```

One noticeable difference compared with the original scraped database is that the practice results are combined into a single table.

```
str(dbGetQuery(f1archive, ("SELECT * FROM pResults")))
```

```
## 'data.frame':      15552 obs. of  12 variables:
##   $ driverNum : chr  "14" "22" "77" "19" ...
##   $ time      : num  91.8 92.4 92.4 92.4 92.6 ...
##   $ laps      : int  20 23 27 19 26 17 10 28 19 30 ...
##   $ year      : int  2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 ...
##   $ natGap    : num  0 0.517 0.563 0.591 0.759 ...
##   $ gap       : num  0 0.517 0.563 0.591 0.759 ...
##   $ session   : chr  "PRACTICE 1" "PRACTICE 1" "PRACTICE 1" "PRACTICE 1" ...
##   $ pos       : int  1 2 3 4 5 6 7 8 9 10 ...
##   $ driverName: chr  "Fernando Alonso" "Jenson Button" "Valtteri Bottas" "Felipe Massa" ...
##   $ race      : chr  "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" ...
##   $ team      : chr  "Ferrari" "McLaren-Mercedes" "Williams-Mercedes" "Williams-Mercedes" ...
##   $ natTime   : chr  "1:31.840" "1:32.357" "1:32.403" "1:32.431" ...
```

The qualification results scrape has been improved to take into account the different approach to qualifying prior to 2005.

```
str(dbGetQuery(f1archive, ("SELECT * FROM QualiResultsto2005")))
```

```
## 'data.frame':      143 obs. of  9 variables:
## $ race      : chr  "SUNDAY QUALIFYING" "SATURDAY QUALIFYING" "SATURDAY QUALIFYING\
" "SUNDAY QUALIFYING" ...
## $ year       : int  2005 2005 2005 2005 2005 2005 2005 2005 2005 ...
## $ pos        : chr  "15" "5" "10" "3" ...
## $ driverNum : chr  "0" "3" "4" "3" ...
## $ session    : chr  "SUNDAY QUALIFYING" "SATURDAY QUALIFYING" "SATURDAY QUALIFYING\
" "SUNDAY QUALIFYING" ...
## $ driverName: chr  "Anthony Davidson" "Jenson Button" "Takuma Sato" "Jenson Butto\
n" ...
## $ time       : num  191.9 80.5 80.9 164.1 164.7 ...
## $ team       : chr  "BAR-Honda" "BAR-Honda" "BAR-Honda" "BAR-Honda" ...
## $ natTime    : chr  "3:11.890" "1:20.464" "1:20.851" "2:44.105" ...
```

```
str(dbGetQuery(f1archive, ("SELECT * FROM qualiResults")))
```

```
## 'data.frame':      3695 obs. of  14 variables:
## $ session   : chr  "QUALIFYING" "QUALIFYING" "QUALIFYING" "QUALIFYING" ...
## $ q1time    : num  91.7 90.8 92.6 90.9 91.4 ...
## $ driverNum: chr  "44" "3" "6" "20" ...
## $ pos       : chr  "1" "2" "3" "4" ...
## $ q2time    : num  103 102 102 103 103 ...
## $ race      : chr  "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" ...
## $ q3natTime: chr  "1:44.231" "1:44.548" "1:44.595" "1:45.745" ...
## $ q3time   : num  104 105 105 106 106 ...
## $ year      : int  2014 2014 2014 2014 2014 2014 2014 2014 2014 ...
## $ q2natTime: chr  "1:42.890" "1:42.295" "1:42.264" "1:43.247" ...
## $ qlaps     : chr  "22" "20" "21" "19" ...
## $ team      : chr  "Mercedes" "Red Bull Racing-Renault" "Mercedes" "McLaren-Merce\
des" ...
## $ q1natTime : chr  "1:31.699" "1:30.775" "1:32.564" "1:30.949" ...
## $ driverName: chr  "Lewis Hamilton" "Daniel Ricciardo" "Nico Rosberg" "Kevin Magn\
ussen" ...
```

```
str(dbGetQuery(f1archive, ("SELECT * FROM raceResults")))
```

```

## 'data.frame':      22148 obs. of  12 variables:
##   $ race        : chr  "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" ...
##   $ laps         : chr  "57" "57" "57" "57" ...
##   $ driverNum   : chr  "6" "20" "22" "14" ...
##   $ pos          : chr  "1" "2" "3" "4" ...
##   $ session     : chr  "RACE" "RACE" "RACE" "RACE" ...
##   $ grid         : chr  "3" "4" "10" "5" ...
##   $ driverName   : chr  "Nico Rosberg" "Kevin Magnussen" "Jenson Button" "Fernando \
Alonso" ...
##   $ raceNum     : int  1 1 1 1 1 1 1 1 1 ...
##   $ year         : int  2014 2014 2014 2014 2014 2014 2014 2014 2014 ...
##   $ team         : chr  "Mercedes" "McLaren-Mercedes" "McLaren-Mercedes" "Ferrari" \
...
##   $ timeOrRetired: chr  "1:32:58.710" "+26.7 secs" "+30.0 secs" "+35.2 secs" ...
##   $ points       : chr  "25" "18" "15" "12" ...

```

Speed trap data is aggregated from across sessions into a single table.

```
str(dbGetQuery(f1archive, ("SELECT * FROM Speeds")))
```

```

## 'data.frame':      18900 obs. of  8 variables:
##   $ race        : chr  "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" ...
##   $ driverNum   : chr  "77" "11" "20" "6" ...
##   $ qspeed      : chr  "317.5" "315.4" "314.8" "313.8" ...
##   $ pos          : chr  "1" "2" "3" "4" ...
##   $ session     : chr  "QUALIFYING" "QUALIFYING" "QUALIFYING" "QUALIFYING" ...
##   $ driverName   : chr  "Valtteri Bottas" "Sergio Perez" "Kevin Magnussen" "Nico Rosb\
erg" ...
##   $ year         : int  2014 2014 2014 2014 2014 2014 2014 2014 2014 ...
##   $ timeOfDay    : chr  "17:03:47" "17:03:36" "17:04:23" "17:05:23" ...

```

Sector times recorded in different sessions are also now contained within the same table.

```
str(dbGetQuery(f1archive, ("SELECT * FROM Sectors")))
```

```

## 'data.frame':      56457 obs. of  8 variables:
## $ sector      : int  1 1 1 1 1 1 1 1 1 ...
## $ race        : chr  "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" ...
## $ driverNum   : chr  "44" "19" "3" "20" ...
## $ pos         : chr  "1" "2" "3" "4" ...
## $ session     : chr  "QUALIFYING" "QUALIFYING" "QUALIFYING" "QUALIFYING" ...
## $ driverName  : chr  "Lewis Hamilton" "Felipe Massa" "Daniel Ricciardo" "Kevin Magnussen" ...
## $ year        : int  2014 2014 2014 2014 2014 2014 2014 2014 2014 ...
## $ sectortime  : chr  "29.863" "29.940" "30.045" "30.069" ...

str(dbGetQuery(f1archive, ("SELECT * FROM raceFastlaps")))

## 'data.frame':      4234 obs. of  12 variables:
## $ timeOfDay   : chr  "17:41:08" "18:40:12" "18:39:59" "18:12:04" ...
## $ lap          : chr  "19" "56" "56" "38" ...
## $ driverNum   : chr  "6" "77" "14" "26" ...
## $ pos         : chr  "1" "2" "3" "4" ...
## $ race        : chr  "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" ...
## $ stime       : num  92.5 92.6 92.6 92.6 92.9 ...
## $ raceNum    : int  1 1 1 1 1 1 1 1 1 ...
## $ year        : int  2014 2014 2014 2014 2014 2014 2014 2014 2014 ...
## $ team        : chr  "Mercedes" "Williams-Mercedes" "Ferrari" "STR-Renault" ...
## $ speed       : chr  "206.436" "206.235" "206.128" "206.088" ...
## $ natTime     : chr  "1:32.478" "1:32.568" "1:32.616" "1:32.634" ...
## $ driverName  : chr  "Nico Rosberg" "Valtteri Bottas" "Fernando Alonso" "Daniil Kvyat" ...

```



```

str(dbGetQuery(f1archive, ("SELECT * FROM racePits")))

```

```

## 'data.frame':      8599 obs. of  13 variables:
## $ natPitTime     : chr  "17.255" "32.657" "25.541" "34.921" ...
## $ totalPitTime   : num  17.3 32.7 25.5 34.9 22.4 ...
## $ race           : chr  "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" ...
## $ natTotalPitTime: chr  "17.255" "32.657" "25.541" "34.921" ...
## $ driverNum      : chr  "8" "21" "11" "77" ...
## $ stops          : chr  "1" "1" "1" "1" ...
## $ pitTime        : num  17.3 32.7 25.5 34.9 22.4 ...
## $ driverName     : chr  "Romain Grosjean" "Esteban Gutierrez" "Sergio Perez" "Val\ 
tteri Bottas" ...
## $ raceNum        : int  1 1 1 1 1 1 1 1 1 ...
## $ year           : int  2014 2014 2014 2014 2014 2014 2014 2014 2014 ...
## $ team           : chr  "Lotus-Renault" "Sauber-Ferrari" "Force India-Mercedes" "\ 
Williams-Mercedes" ...
## $ lap             : chr  "1" "1" "1" "10" ...
## $ timeOfDay      : chr  "17:09:56" "17:10:12" "17:10:14" "17:24:46" ...

```

Problems with the Formula One Data

Until the 2014 season, driver numbers were allocated to drivers based on the team they drove for and the classification of the team in the previous year's Constructors' Championship. This makes them impossible to use as a consistent identifier across years (driver number 3 this year may not be the same person as driver number 3 last year), something that the introduction of personal driver numbers should help to address. That said, who takes driver number 1 may still change year on year, that number being reserved for the previous year's Driver's World Champion.

How to use the Formula1.com Data alongside the *ergast* data

If we compare certain key elements of the data scraped from the Formula1.com website and the *ergast* data, there are several differences in the presentation of what we might term “naturally occurring” keys, such as the name of a race, or driver.

To be able to jointly work on *ergast* data and data from the Formula1.com website, we need to define mapping or lookup operations that allows us to associate unique elements from one dataset with corresponding unique elements from the other.

Reviewing the Practice Sessions

In which we learn how to look at charts and go back to basics in terms of how to read them, as well as getting started with constructing some charts of our own.

The Weekend Starts Here

Whilst the focus of a race weekend are rightly the Saturday qualifying session and the Sunday race, the race weekend on track action currently begins with P1 and P2 (also referred to as FP1 and FP2), the two, ninety minute Friday free practice sessions (Thursday at Monaco), and the hour long third free practice (P3/FP3) session before qualifying, as specified in article 32 of the FIA Formula One Sporting Regulations⁴⁰. The practice sessions provide the teams with the essential track time they need to set up the cars, and in doing so provide a source of data that may help us identify which cars are likely to be competitive on any particular race weekend. In particular, second practice provides the teams with an opportunity to try some long run race simulations. (As the *intelligentf1* blog showed in previous years, analysis of the laptimes from these second practice long runs can often provide a good insight in the underlying race pace and performance of each car.) Third practice, occurring as it does a couple of hours before qualifying (article 32 of the 2014 regulations set third practice (P3) time between 11.00 and 12.00, with qualifying scheduled between 14.00 and 15.00) provides teams with a final chance to set the car up for qualifying.

The *ergast* database does not contain any data relating to the practice sessions, but we can get a certain amount of information from the FIA website (previously, the formula1.com results pages, which I archived, as described previously), as well as more detailed timing information from the FIA media centre. Prior to the 2014 season, the free (though login require) live timing application on the Formula One website also used to stream live sector times; using third party applications, this data could be scraped and archived from the live timing feed. However, the 2014 season has seen functionality on the free live timing app reduced, including the withdrawal of sector times. At the time of writing, third party developers have not publicly released reverse engineered applications capable of streaming (and logging) data from the paid for official Formula One live timing application. And if they did, I suspect that FOM would probably not be very happy about it!

⁴⁰http://www.fia.com/sport/regulations?f%5B0%5D=field_regulation_category%3A82

Using Practice Laptime Data to Build Race Simulations

During 2012-13, the *intelligentf1.com* website provided a great example of how to make use of lap time data from the race simulations that are typically carried out by each team in second practice. On that site, James Beck used laptime data to calibrate a model that explored the relative competitiveness of each driver and give a feel for the relative tyre degradation rates experienced by each team. More recently, applied mathematician Dr Andrew Phillips has described his approach to building a race simulation on his *F1Metrics* blog: Building a Race Simulator^a. Associated Matlab code can be found in the *syllogismos/F1Sim*^b Github repository.

^a<https://f1metrics.wordpress.com/2014/10/03/building-a-race-simulator/>

^b<https://github.com/syllogismos/F1Sim/>

Throughout this chapter, we will focus solely on data scraped originally from the Formula One website and, from the start of the 2015 season, from the FIA website. Slightly more comprehensive data can also be scraped from the official FIA timing sheets, although that will not be covered here.

Practice Session Data from the Official Formula One Website Prior up to 2014

To start with, we need to establish a connection to the database containing the scraped data.

```
#If you haven't already installed the RSQLite package, you will need to do so.
#install.packages("RSQLite")
library(DBI)
f1 =dbConnect(RSQLite::SQLite(), './scraperwiki.sqlite')
```

The structure of each of the practice tables are the same. The tables containing results from a practice session take the following form:

```
dbGetQuery(f1, ("SELECT * FROM p1Results LIMIT 5"))
```

```

##   driverNum   time laps year natGap   gap      race pos      driverName
## 1          3 87.560  11 2012  0.000 0.000 AUSTRALIA  1      Jenson Button
## 2          4 87.805  14 2012  0.245 0.245 AUSTRALIA  2      Lewis Hamilton
## 3          7 88.235  17 2012  0.675 0.675 AUSTRALIA  3 Michael Schumacher
## 4          5 88.360  21 2012  0.800 0.800 AUSTRALIA  4 Fernando Alonso
## 5          2 88.467  21 2012  0.907 0.907 AUSTRALIA  5      Mark Webber
##               team   natTime
## 1      McLaren-Mercedes 1:27.560
## 2      McLaren-Mercedes 1:27.805
## 3           Mercedes 1:28.235
## 4           Ferrari 1:28.360
## 5 Red Bull Racing-Renault 1:28.467

```

The practice session results include the name of each driver; their classification within that session; their team; the number of laps they completed; their best laptime as a natural time (using the format *minutes:seconds.milliseconds*) and as a time in seconds and milliseconds; and the natural gap (*natgap*)/*gap* (the *natgap* as seconds/milliseconds) to the best time in the session.



What can we do with the practice session results data? Spend a few minutes sketching out what charts or analyses you think you might be able generate from this data. For example, which data columns can usefully be plotted against each other, and what would such visualisations show? How might data be grouped, either as the basis of point colouring or for within group comparisons or analysis? How might you combine - and visualise - data from all the practice sessions in a race weekend? What *derived data columns* can you generate from the data?

To start with, we will consider charts that correspond to data from a single session in a single race. Then we will have a look at what additional views we may get from the data by comparing and/or aggregating data from two or three practice sessions from the same race weekend.

Trivially, in the case of a single practice session from a single weekend, we might begin with:

- the best laptime by driver versus driver classification in a session;
- the number of laps completed by each driver;
- a single dimensional chart show the position of each driver's laptime along a timeline.

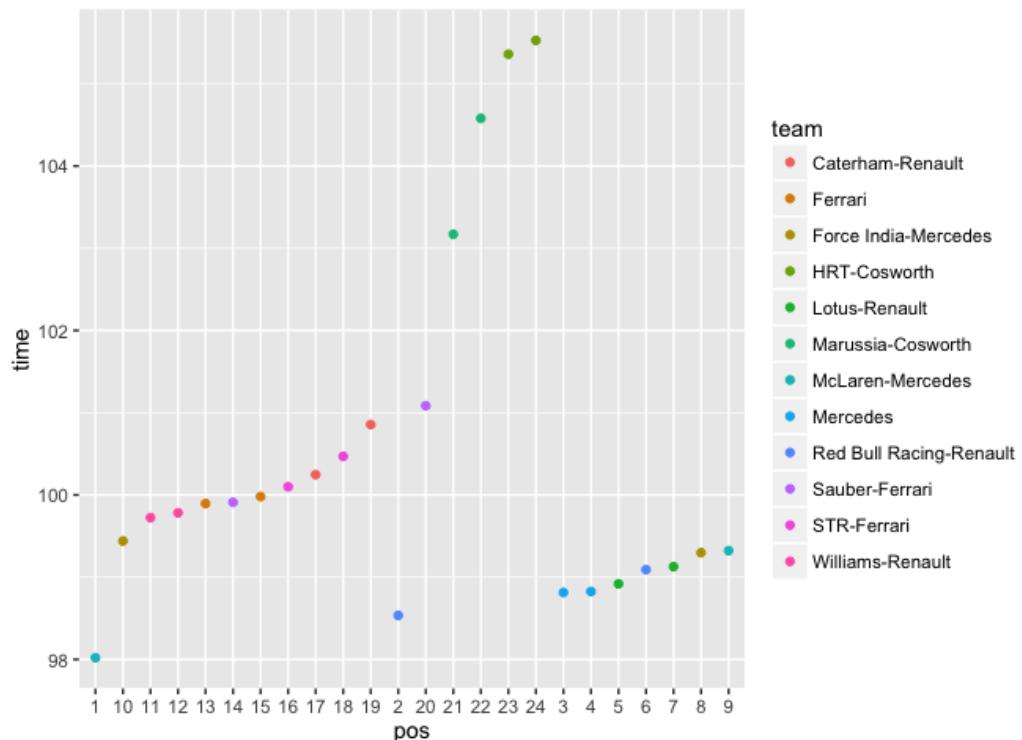
We can quickly sketch these as follows:

```
#Get some data - P1 from Malaysia in 2012
dd=dbGetQuery(f1, ("SELECT * FROM p1Results WHERE year=2012 AND race='MALAYSIA'"))

#Check the column types
#str(dd)

#It may be the case that not all the columns that should be integers actually are...
#If necessary, here's a temporary fix...
#dd$pos=as.integer(dd$pos)
#dd$laps=as.integer(dd$laps)

library(ggplot2)
ggplot(dd) + geom_point(aes(x=pos,y=time,col=team))
```

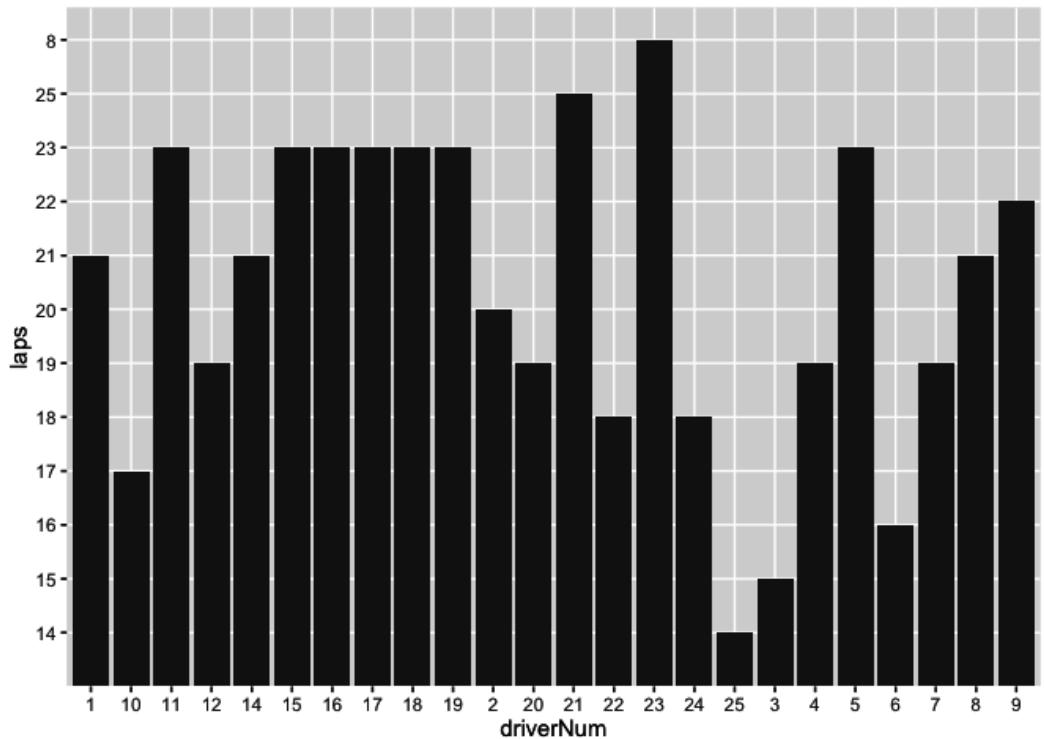


Raw, unordered scatterplot of laptime against rank position

This chart shows how the times differ by rank. We use the colour attribute to help us spot whether there appears to be any sense of grouping around the times recorded by drivers from any particular team.

The simplest view over the lap count is a bar chart. The biggest question this initial sketch raises for me is how we should order the horizontal x-axis, which identifies each driver.

```
ggplot(dd)+geom_bar(aes(x=driverNum,y=laps),stat='identity')
```





What, if anything, is wrong with these charts?

In both charts, the values on the x-axes are arranged in “alphabetical” rather than numerical order. In the case of driver number, this may make sense (the driver number is essentially an arbitrary, unique categorical label or identifier), but it is a nonsense as far as the position ordering goes.

In the laptime chart, the intention behind the x-axis ordering was to order times by position, which does not give us any information within the chart about which driver the time was associated with. (We may, however, get information about the team from the point colour.)

It is also worth noting in the laptime chart that the x-axis describes a *discrete* scale, with markers equally spaced horizontally, whereas the y-axis is a *continuous* scale, and where the vertical spacing between points is related to the actual difference in y-value represented by each point.

In the lap count chart, the y-axis values are also “alphabetically” ordered. This really is a nonsense and shows how important it is to check your axis tick mark labels. To make this chart in any way meaningful, we need to ensure that the lap count is treated as a numerical quantity.

Insofar as a chart represents a particular quantity associated with a particular car or driver, what might the most appropriate ranking be? The first decision to take is whether the x-axis values are ordered based on the x-axis values *or the y-axis values*. Here are some possibilities:

- for the laptime chart, the most sensible classification is probably in rank order of position, which is also the rank order of the fastest laptime (the y-axis value);
- for the lap count chart, we could rank by driver number. This arguably made more sense using the numbering scheme that existed prior to 2014, when personal numbers were introduced, because drivers were essentially ranked based on performance of the team in the Constructors’ Championship from the previous season and the nominal first driver/second driver ranking within each team). With personal driver numbers the numerical ordering is not really meaningful - the personal driver numbers are qualitative, categorical labels rather than numbers *per se*. A more useful ordering would probably be to order the x-axis values so that the y-values are in height order (that is, by increasing or decreasing lap count), relabeling the x-axis *driverNum* with the corresponding driver’s name or identifier.
- we might arrange the bars into groups for each team so we see at a glance how much track time each team had. But if we do that, how do we order the groups, and the bars within each team? Should we dodge them, allowing us to compare drivers within a team? Or stack them, allowing us to more easily compare aggregated team results?

Whenever you have a categorical axis, such as driver identifiers in the case of the lap count chart, it's worth remembering that the ordering is essentially arbitrary, although some orderings may be more meaningful than others in terms of the semantic relationships between the different categorical values. For example, how might we group, or order drivers? Previous world champions vs rookies vs others is another possible grouping; and so on. Team members group well, but how should we order the teams? Or the drivers within a team? Different ordering choices may also help you see different patterns or structures within the data.

If we inspect the structure of the data, we see that the pos column is indeed identified as a character type, as are the laps and driverNum columns:

```
str(dbGetQuery(f1, ("SELECT * FROM p1Results LIMIT 5"))[c('pos', 'laps', 'driverNum')])
```

```
## 'data.frame':      5 obs. of  3 variables:
## $ pos     : chr  "1" "2" "3" "4" ...
## $ laps    : chr  "11" "14" "17" "21" ...
## $ driverNum: chr  "3" "4" "7" "5" ...
```

Constructing Charts With *ggplot*

The *ggplot2* package, written by Hadley Wickham, is an implementation of Leland Wilkinson's *The Grammar of Graphics*. The grammar allows charts to be constructed as a series of mappings from data variables onto visual variables. By overlaying different layers onto a chart, data from one - or more - datasets can be used in the construction of a single chart. For a good introduction to The Grammar of Graphics, see Hadley Wickam's 2010 paper *A layered grammar of graphics*, Journal of Computational and Graphical Statistics 19(1) pp.3-28^a

^a<http://vita.had.co.nz/papers/layered-grammar.html>

If we recast the position to an integer, the laptime chart can be represented more intuitively (we can also take the opportunity to tidy up the chart labels and add a title to the chart itself):

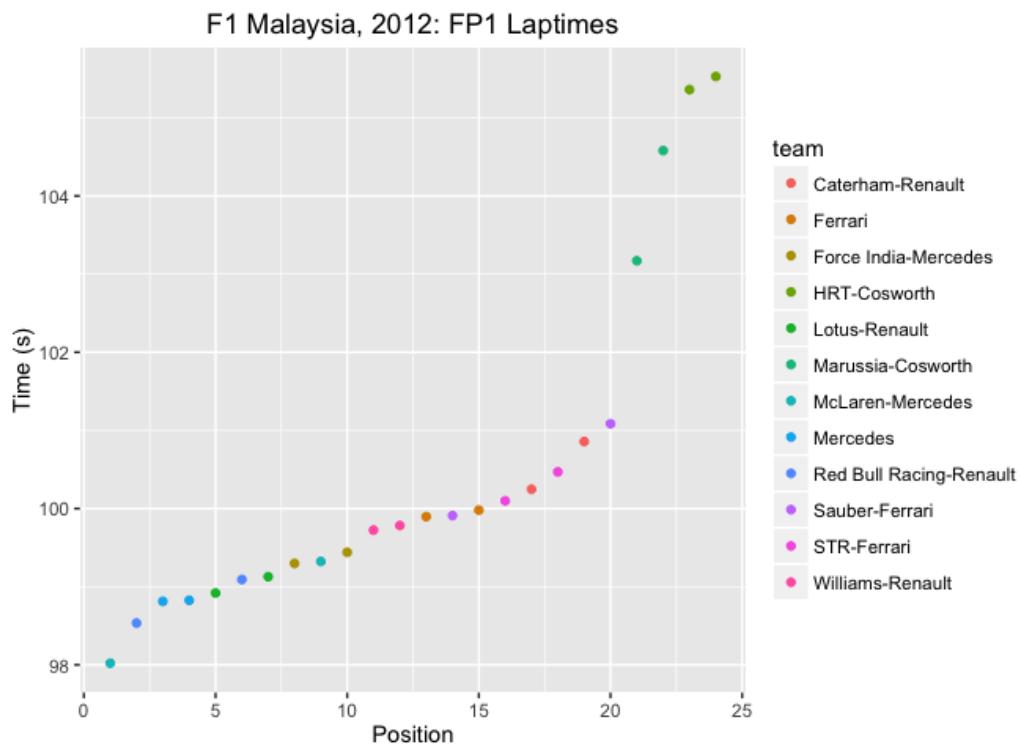
```

dd$pos=as.integer(dd$pos)

g=ggplot(dd)+geom_point(aes(x=pos,y=time,col=team))
g=g+xlab('Position')+ylab('Time (s)')

#Add in a chart title
g=g+ggtitle('F1 Malaysia, 2012: FP1 Laptimes')
# We can set the title of the legend, or remove it, as in this case.
# Note the default ordering in the legend is alphabetical and a default
# colour palette is used.
g=g+ guides(fill=guide_legend(title=NULL))
g

```

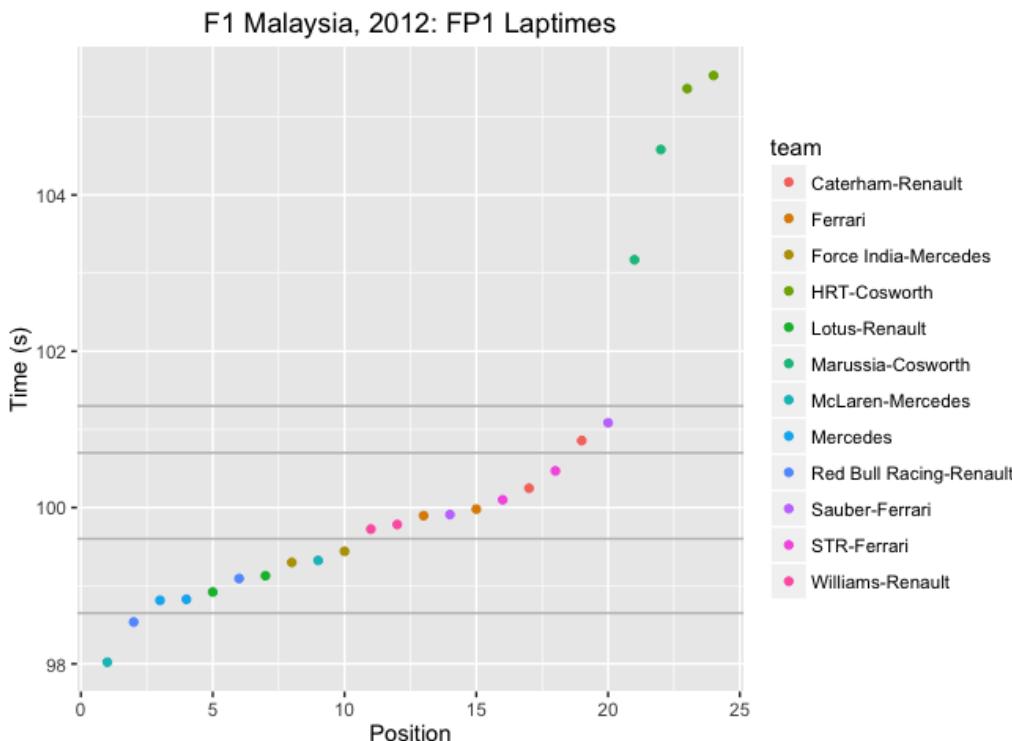


Labelled chart showing laptimes against ranked position

By eye, we can identify several different groupings in the data. Let's add some guide lines to the chart to separate out the groups: the first two cars are clear of cars in positions 3 to 10; cars 11 to 18, or perhaps 11 to 20 are in a group of their own, and then the four backmarkers

are way off the pace.

```
g2=g+geom_hline(yintercept=98.65,col='grey')
g2=g2+geom_hline(yintercept=99.6,col='grey')
g2=g2+geom_hline(yintercept=100.7,col='grey')
g2=g2+geom_hline(yintercept=101.3,col='grey')
g2
```



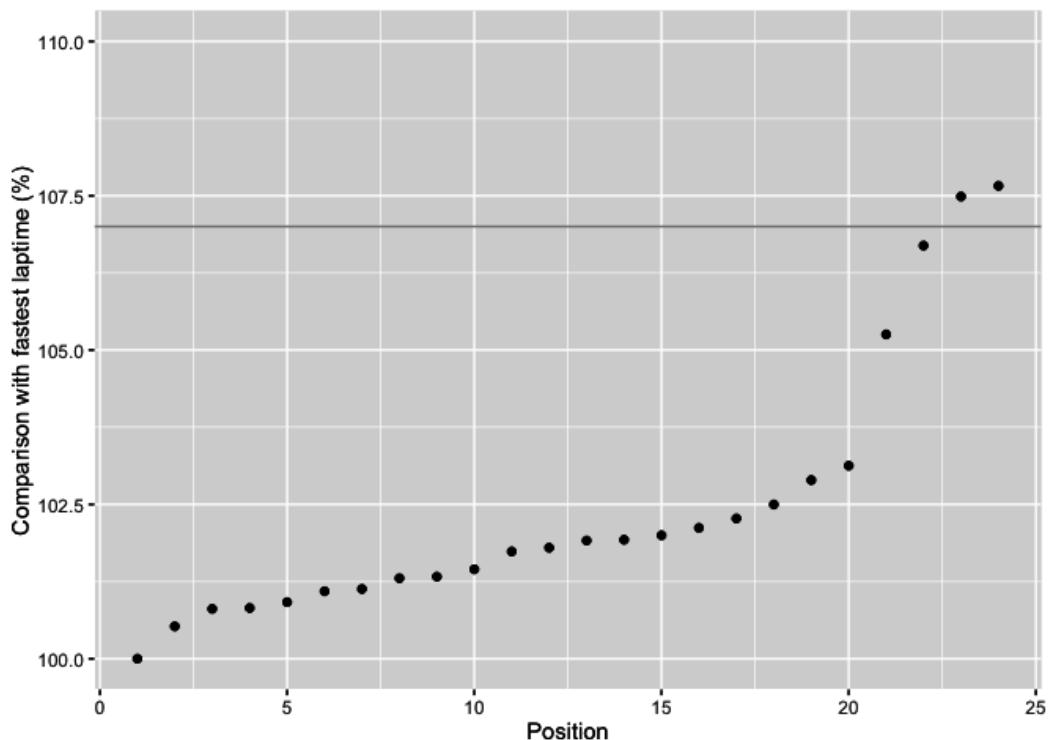
Annotated chart showing laptime against ranked position, with horizontal “separation bars” added manually

One thing we might want to think about it how to identify clusters more reliably/algorithimically; but for now, we'll rely on making manual annotations.

Another way of viewing the same data is to look at the lap times in terms of the percentage they are over the fastest laptime. This allows us to make comparisons across sessions within a particular race weekend, where conditions may change day to day, or across races, where individual laptimes may differ significantly depending on the length and character of the

circuit. The 107% value is meaningful in the sense of F1 regulations, because if a car does not record a time within 107% of the first placed car in qualification, it does not automatically qualify for the actual race.

```
dd$percent=100*dd$time/min(dd$time)
g = ggplot(dd) + geom_point(aes(x=pos, y=percent),stat='identity')
g = g + geom_hline(yintercept=107, col='grey') + ylim(100, 110)
g + xlab('Position') + ylab('Comparison with fastest laptime (%)')
```



Laptime shown as a percentage of the fastest lap time, with 107% line added

To get a clearer view of the time differences between cars, we can look at the difference in time between each car and the car classified one position ahead of it by generating a `delta` column between their lap or gap to leader times. To do this, sort the data by position and then use the `diff()` function to find the lagged difference between consecutive times in the time/position ordered dataframe.

```

library(plyr)
ddx=dd
ddx=arrange(ddx, pos)
ddx$delta=c(0, diff(ddx$natGap))

head(ddx, n=3)

##   driverNum   time laps year natGap   gap      race pos      driverName
## 1          4 98.021  19 2012  0.000 0.000 MALAYSIA  1  Lewis Hamilton
## 2          1 98.535  21 2012  0.514 0.514 MALAYSIA  2 Sebastian Vettel
## 3          8 98.813  21 2012  0.792 0.792 MALAYSIA  3  Nico Rosberg
##                               team natTime percent delta
## 1      McLaren-Mercedes 1:38.021 100.0000 0.000
## 2 Red Bull Racing-Renault 1:38.535 100.5244 0.514
## 3      Mercedes 1:38.813 100.8080 0.278

```

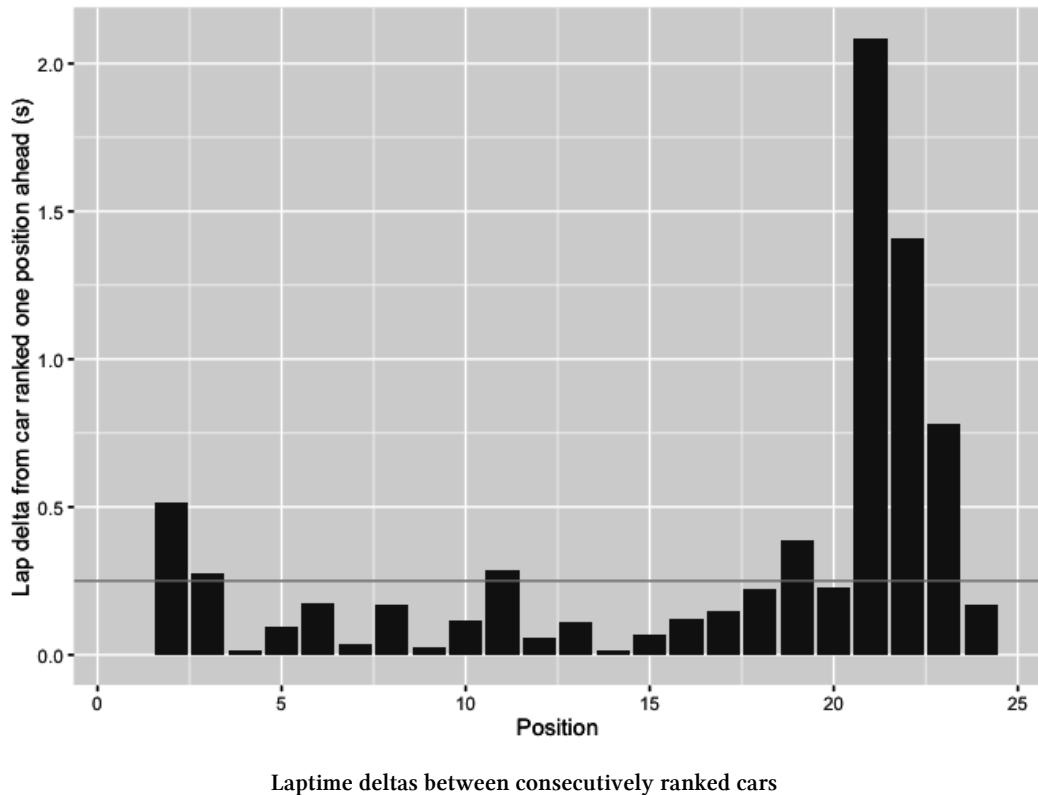
The `diff()` function returns a list of differences between consecutive items in a list, which is to say it will return a list of $N-1$ difference values when presented with a list containing N items. We return a list of difference values containing N items by prepending the actual difference list with a conventional first value of 0. That is, we assume that a zero difference time is associated with the first item of the list. Another convention would be set it as NA.

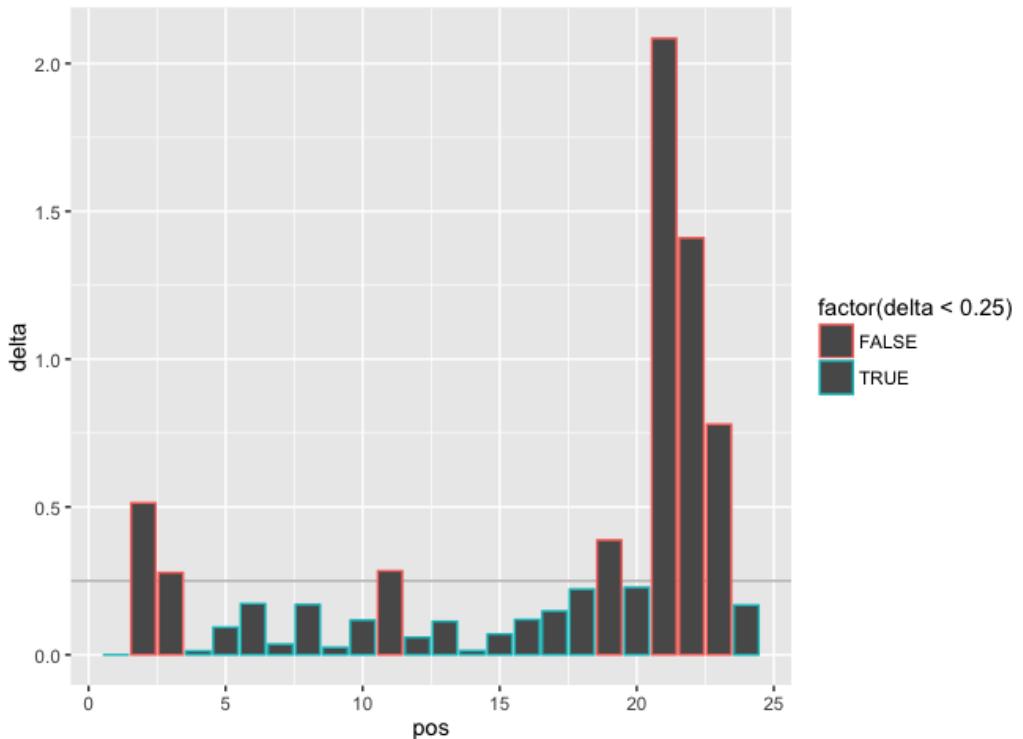
We can now plot these values as a bar chart, adding a line that helps us identify deltas of more than a quarter of a second, for example, between cars.

```

g = ggplot(ddx) + geom_bar(aes(x=pos, y=delta), stat='identity')
g = g + geom_hline(yintercept=0.25, col='grey')
g + xlab('Position') + ylab('Lap delta from car ranked one position ahead (s)')

```

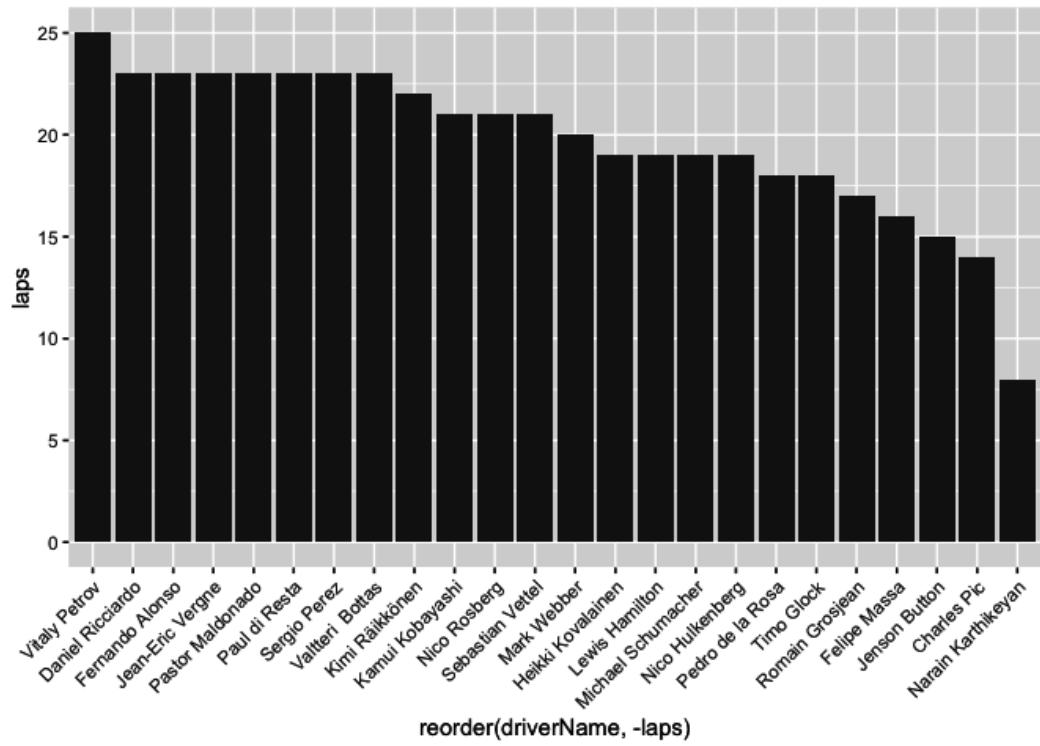




Lap delta bar chart with coloured bar edges

Returning to the laps count, how might we more correctly render that? Recall, the `laps` count wasn't being correctly rendered as a number, and we also needed to find a way to sort the x-axis values on the basis of the corresponding y-axis value. In the following example, we sort in terms of decreasing lap count.

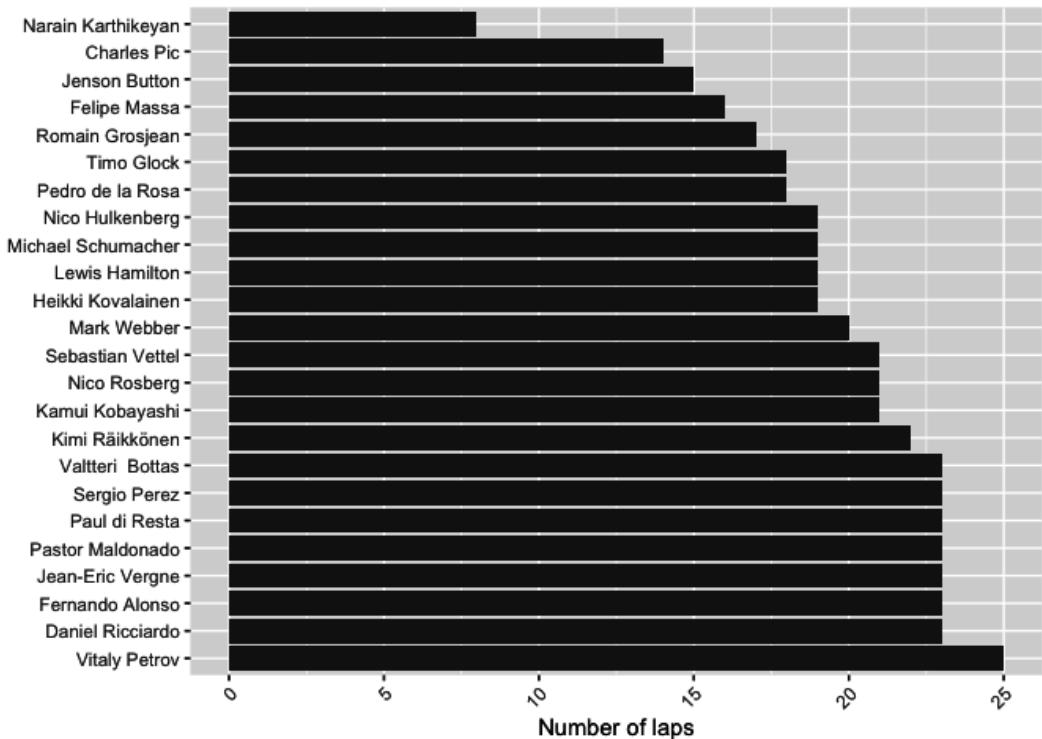
```
dd$laps = as.integer(dd$laps)
g = ggplot(dd) + geom_bar(aes(x=reorder(driverName,-laps), y=laps), stat='identity')
g = g + theme(axis.text.x = element_text(angle = 45, hjust = 1))
g
```



Lap count bar chart ordered by lap count

We can also rotate the chart to provide a horizontal bar chart view, which is arguably easier to read in this case.

```
g + coord_flip() + xlab(NULL) + ylab('Number of laps')
```

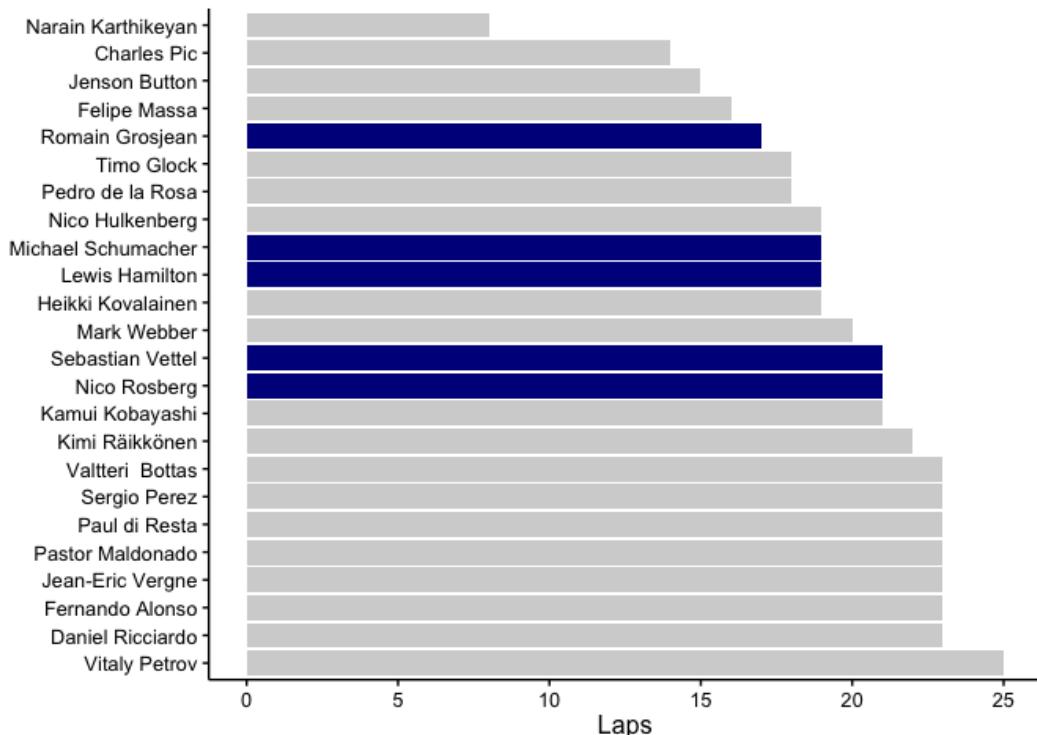


Horizontal bar chart, with the original x/y axis labels flipped too

Note how we can add additional transformations to the chart object without having to rebuild the chart from scratch.

In terms of looking for meaning in this chart, we might look to see how many laps the top 5 cars by laptime completed. We can highlight the drivers classified in the top 5, adopt a clean, minimal theme and then suppress the legend.

```
#In part via http://stackoverflow.com/a/12910865/454773
dd[["top5"]] = ifelse(dd$pos <= 5, "top5", "notTop5")
g = ggplot(dd)
g = g + geom_bar(aes(x=reorder(driverName,-laps), y=laps, fill=top5), stat='identity')
g = g + coord_flip() + xlab(NULL) + ylab('Laps')
g = g + scale_fill_manual(values = c("top5" = "darkblue", "notTop5" = "lightgrey"))
g + theme_classic() + theme(legend.position="none")
```

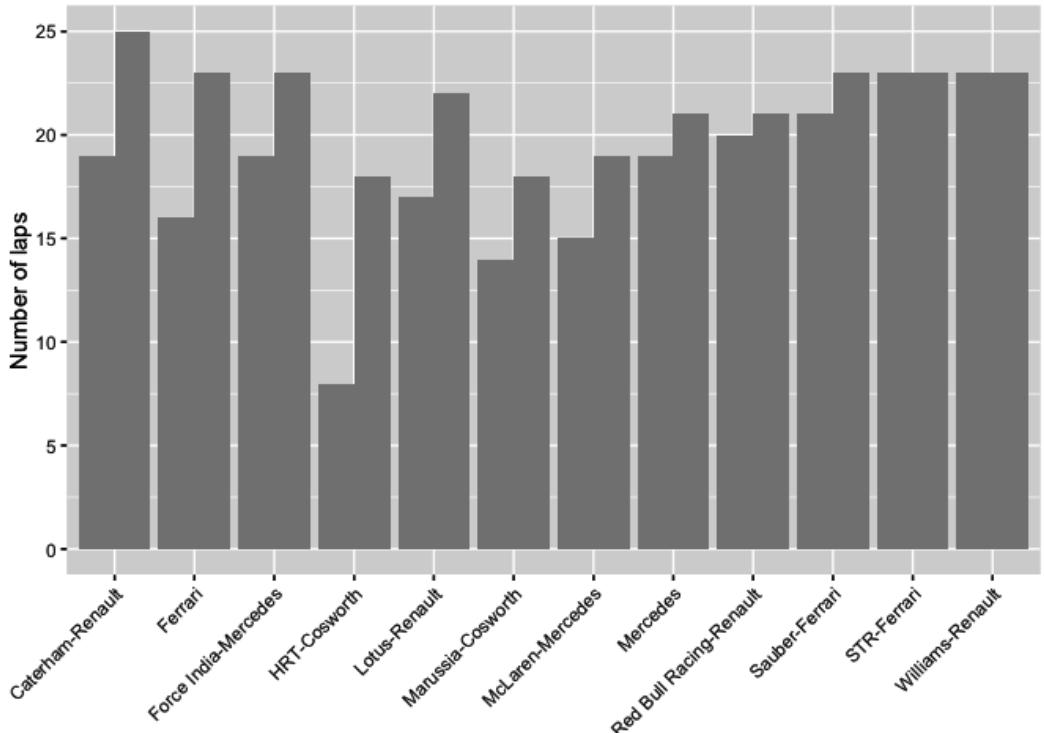


Styled horizontal bar chart with top 5 ranked drivers highlighted

If we want to compare the number of laps by team, we can group the bars on that basis. Prior to 2014, we could use driver number to identify which driver was the “first” driver in a team and which driver was the second ranked driver and then order drivers within teams on that basis. One way to do this for the current season might be to nominate the driver who was classified highest in the previous year’s Drivers’ Championship as the “first driver” within a team (with rookies being ranked lowest and a strategy for numbering drivers in teams with two rookies). In the chart below, it is ambiguous which driver corresponds to which bar within each team. For visual clarity, it might make sense to order the bars within each group in ascending order:

```
#Order the driverName used for grouping within each team by number of laps
dd$driverName=reorder(dd$driverName,dd$laps)

g = ggplot(dd)
g = g + geom_bar(aes(x=team, y=laps, group=factor(driverName)),
                  stat='identity', fill='grey', position='dodge')
g = g + theme(axis.text.x = element_text(angle = 45, hjust = 1))
g + xlab(NULL) + ylab('Number of laps')
```



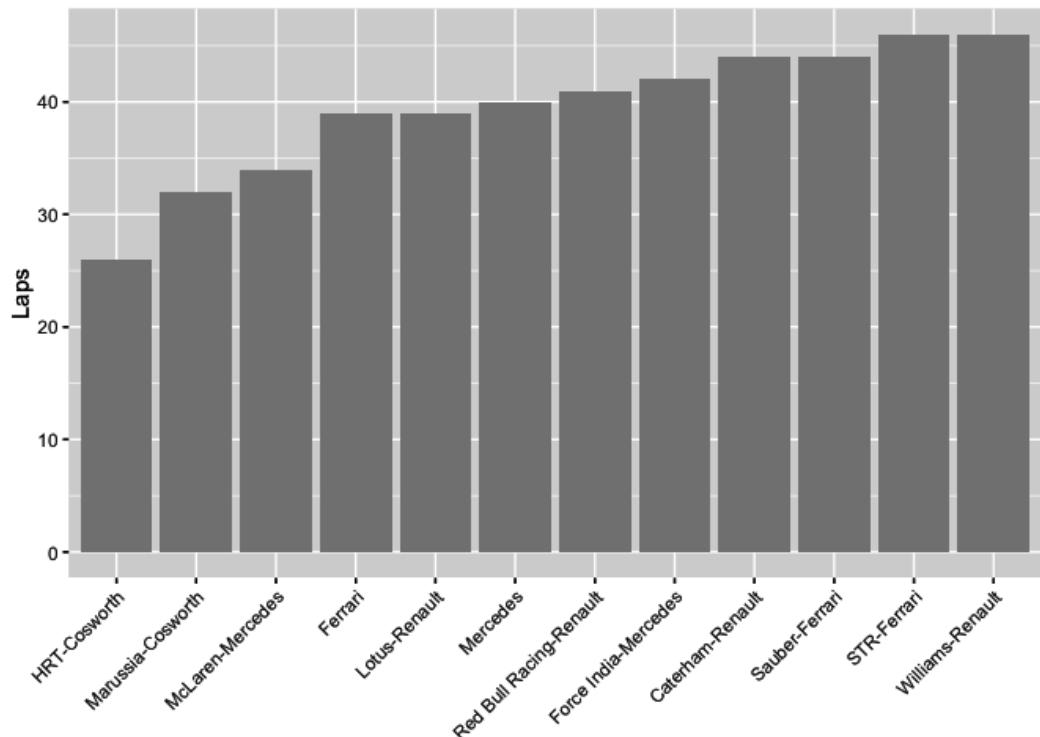
Dodged bar chart showing lap count for drivers grouped by team

To view the total number of laps by team, we can stack the bars, perhaps ordering the chart by the total number of laps completed by the team. *To order the bars by descending total lapcount, rather than increasing total lapcount, simply reorder by -laps rather than laps.*

```

g = ggplot(dd)
g = g + geom_bar(aes(x=reorder(team, laps), y=laps),
                  stat='identity', fill='grey', position='stack')
g = g + theme(axis.text.x = element_text(angle = 45, hjust = 1))
g + xlab(NULL) + ylab('Laps')

```

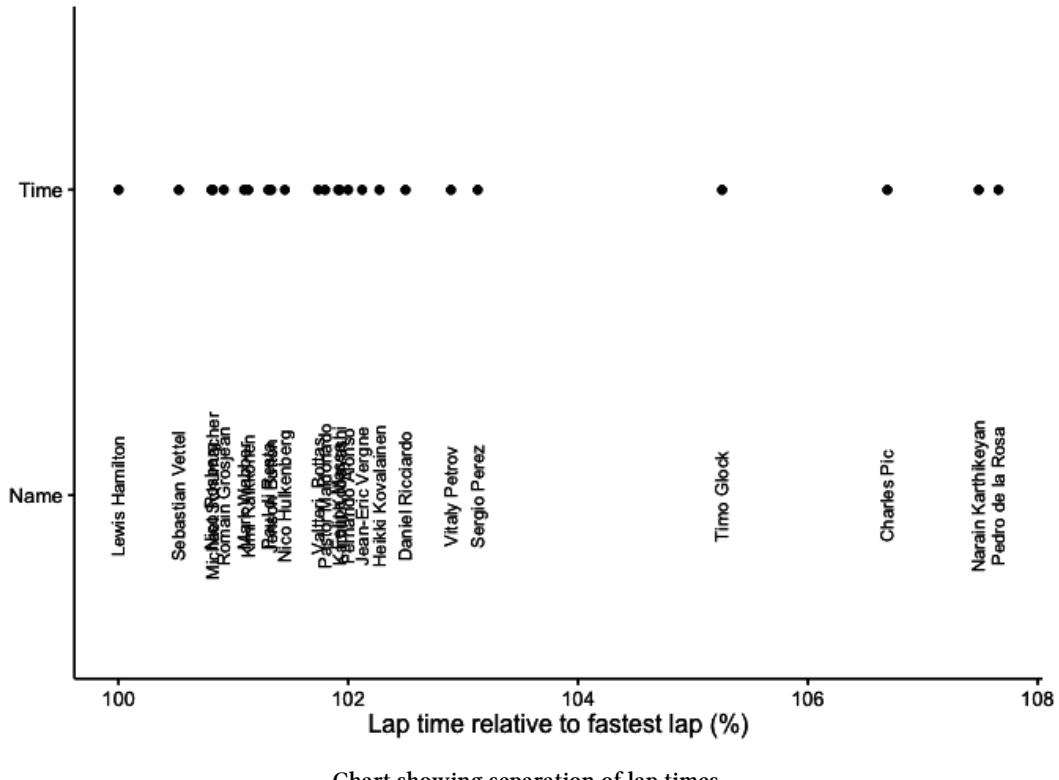


Stacked bar chart showing total lap count by team

As you might imagine, if we had a single dataframe containing the number of laps completed by drivers from each team from across several practice sessions, we could use this sort of chart to display the total number of laps of running achieved by each team throughout practice.

Let's flip back again to the laptimes chart: how else might we try to identify not only the gap between drivers, but also their positions, by driver name? One way is to use a horizontal x-axis that identifies the laptime, and then place a marker for each driver, identifying them by name. This essentially reduces the data to a single dimension data set. In the chart below, we space driver names and markers according to lap time. Note how we rotate the driver names to make the chart more readable as well explicitly indicating the relative laptimes.

```
g = ggplot(dd) + geom_point(aes(x=percent, y="Time"))
g = g + geom_text(aes(x=percent, y="Name", label=driverName), angle=90, size=3)
g + theme_classic() + xlab('Lap time relative to fastest lap (%)') + ylab(NULL)
```



Whilst there are some problems with this chart (for example, some occlusion of overlapping names) the linear dot plot highlights just how close the drivers are to each other in terms of lap time.

Comparing Team Performances

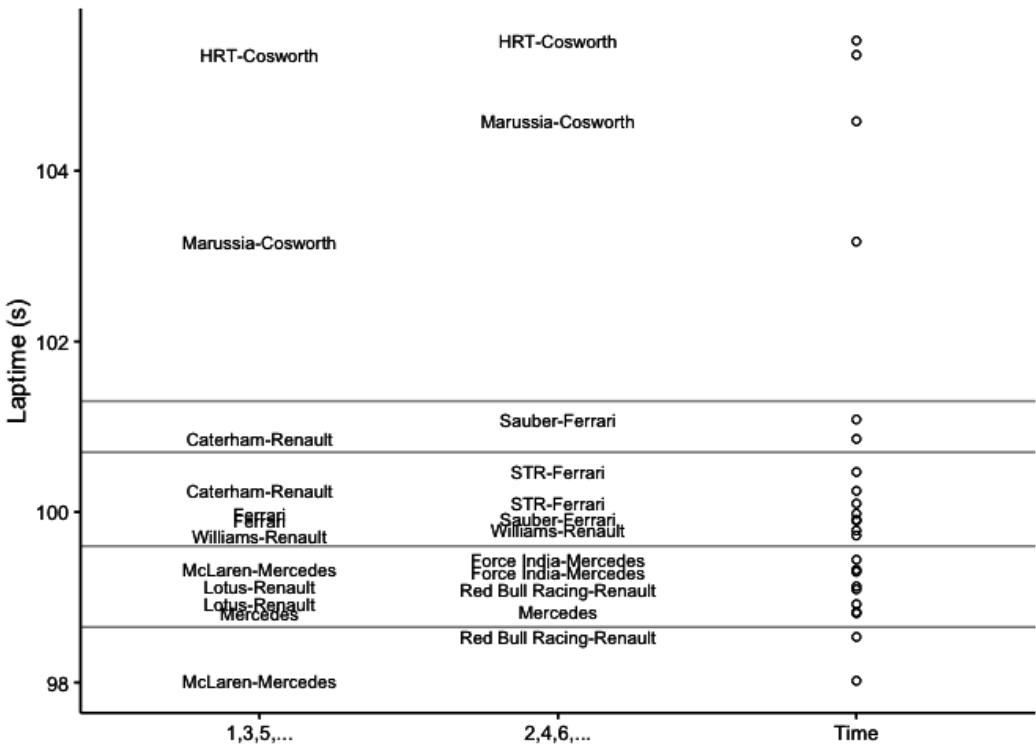


How would you modify the above chart to display the team names rather than driver names, organising the names using a vertical rather than horizontal profile? Can you think of a way of making the names more readable? Where laptimes are so close to each other that they merge into each other, how might you more clearly distinguish between marks that overlap each other?

We can use team names rather than driver names by setting the text label to the `team` column. To rotate the chart, we can use `coord_flip()`, and also remove the label rotation within the `geom_text()` element.

One way of making the names more readable is to split the data into separate columns, for example putting cars placed in odd numbered positions in one column and those placed in even numbered columns in a second column. To cope with overlapping symbols that mark out the time points, we might use a different symbol type, such as an unfilled or empty circle.

```
#Basic dot plot - shape=1 is an empty circle
#See for example: http://www.cookbook-r.com/Graphs/Shapes_and_line_types/
g=ggplot(dd) + geom_point(aes(x=time,y="Time"),shape=1)
#Split the drivers into two groups - odd position number and even position number
#Use each group as a separate y-axis categorical value
g = g + geom_text(data=subset(dd, subset=(pos %% 2!=0)),
                   aes(x=time, y="1,3,5,...", label=team), size=3)
g = g + geom_text(data=subset(dd, subset=(pos %% 2==0)),
                   aes(x=time, y="2,4,6,...", label=team), size=3)
#Tweak the theme
g = g + theme_classic() + ylab(NULL) + xlab('Laptimes (s)')
#Add in some intercept lines using the values we used before
g = g + geom_vline(xintercept=98.65, col='grey')
g = g + geom_vline(xintercept=99.6, col='grey')
g = g + geom_vline(xintercept=100.7, col='grey')
g = g + geom_vline(xintercept=101.3, col='grey')
#Flip the co-ordinates
g=g+coord_flip()
g
```



Flipped and dodged chart showing separation of lap times

If team names are close to each other on a horizontal line or vertically, then those cars are close in terms of time, even if not directly next to each other in terms of position. By using an empty circle rather than a filled circle to identify the lap times, we can more clearly see very close laptimes. We might even further distinguish between times associated with the dodged left-hand and right-hand labels using different symbol types.

Once again, the grouping elements are also emphasised by visual proximity, something we can additionally reinforce using the grey splitter lines. Although the grouping is to some extent arbitrary, it may be useful as a storytelling or journalistic device.

Note that another way of dodging the names might be to assign drivers in the same team to different columns, although we would have to decide on what basis to put a particular driver into a given column.

Something else that jumps out from this particular example is that the team names are actually a combination of the manufacturer and engine supplier. (Where the name is not hyphenated the manufacturer *is* the engine supplier, as in the case of Ferrari and Mercedes).

This suggests we can generate two additional data columns: *car manufacturer*, which is what we typically think of as the team name; and *engine supplier*.

```
#For each row, split the team name on a '-'
#and take the last item in the resulting list
dd=ddply(dd,
  .(driverName),
  mutate,
  engine = tail(strsplit(team,'-')[[1]], n=1) )

head(dd,n=3)

##   driverNum      time laps year natGap   gap      race pos      driverName
## 1        23 105.360     8 2012  7.339 7.339 MALAYSIA 23 Narain Karthikeyan
## 2        25 104.580    14 2012  6.559 6.559 MALAYSIA 22 Charles Pic
## 3        3  99.323    15 2012  1.302 1.302 MALAYSIA  9 Jenson Button
##             team natTime percent top5   engine
## 1 HRT-Cosworth 1:45.360 107.4872 notTop5 Cosworth
## 2 Marussia-Cosworth 1:44.580 106.6914 notTop5 Cosworth
## 3 McLaren-Mercedes 1:39.323 101.3283 notTop5 Mercedes
```

We can quickly count how many teams run each engine. First, identify the unique engine/team combinations:

```
engineTeams = unique(dd[, c('engine','team')])
```

```
##      engine              team
## 1  Cosworth          HRT-Cosworth
## 2  Cosworth          Marussia-Cosworth
## 3  Mercedes          McLaren-Mercedes
## 4  Ferrari            Ferrari
## 5  Renault            Lotus-Renault
## 8  Renault            Caterham-Renault
## 10 Mercedes          Mercedes
## 11 Mercedes          Force India-Mercedes
## 12 Renault Red Bull Racing-Renault
## 13 Ferrari           Sauber-Ferrari
## 17 Ferrari           STR-Ferrari
## 20 Renault            Williams-Renault
```

Then count the number of occurrences of each engine:

```
#Use the plyr count() function
count(engineTeams, 'engine')
```

```
##      engine freq
## 1 Cosworth     2
## 2 Ferrari     3
## 3 Mercedes     3
## 4 Renault      4
```

We can also generate a summary report of the total, mean (average) and median number of laps completed by each car running each engine type:

```
dplyr(dd, .(engine), summarise,
      totLaps = sum(laps), meanLaps = mean(laps), medianLaps = median(laps))

##      engine totLaps meanLaps medianLaps
## 1 Cosworth     58 14.50000    16.0
## 2 Ferrari     129 21.50000    23.0
## 3 Mercedes    116 19.33333    19.0
## 4 Renault      170 21.25000    21.5
```

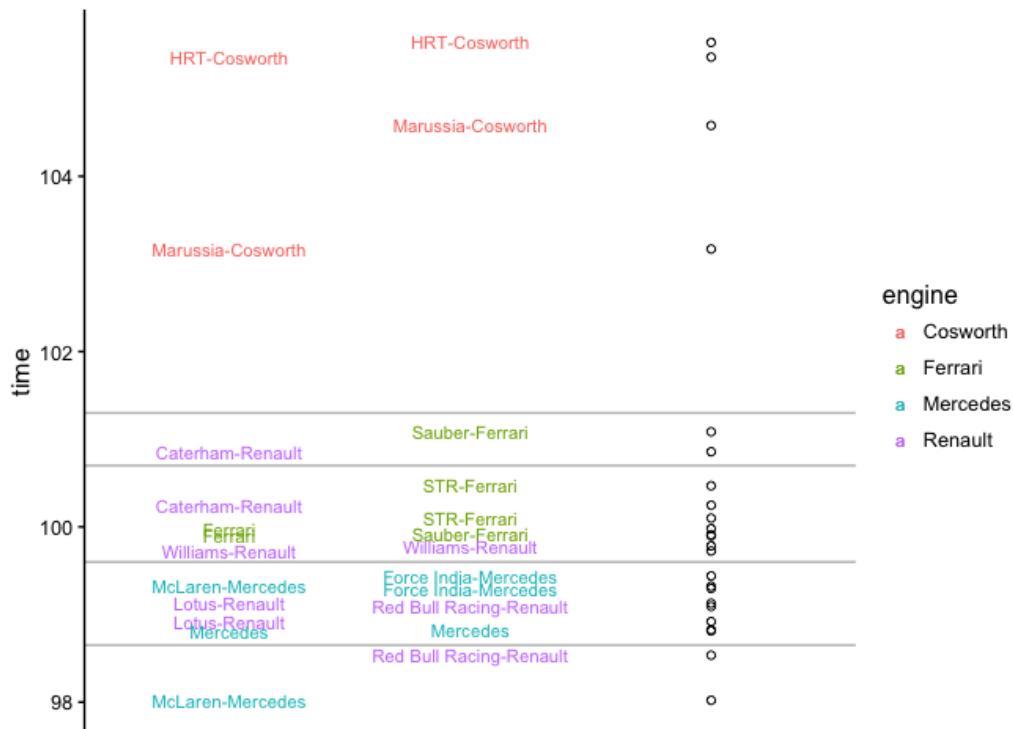
Let's also make use of the engine type to colour our laptime ordering chart. In addition, tidy up the x-axis by removing the ticks and the tick labels.

```
#Basic dot plot
g=ggplot(dd) + geom_point(aes(x=time,y="Time"),shape=1)
#Split the drivers into two groups - odd position number and even position number
#Use each group as a separate y-axis categorical value
#Add in an additional colour aesthetic tied to engine type
g = g + geom_text(data=subset(dd, subset=(pos %% 2!=0)),
                   aes(x=time,y="1,3,5,...", col=engine, label=team), size=3)
g = g + geom_text(data=subset(dd, subset=(pos %% 2==0)),
                   aes(x=time, y="2,4,6,...", col=engine, label=team), size=3)
#Tweak the theme
g = g + theme_classic() + ylab(NULL)
#Add in some intercept lines using the values we used before
g = g + geom_vline(xintercept=98.65, col='grey')
g = g + geom_vline(xintercept=99.6, col='grey')
```

```

g = g + geom_vline(xintercept=100.7, col='grey')
g = g + geom_vline(xintercept=101.3, col='grey')
#Flip the co-ordinates
g = g + coord_flip()
#Remove the x-axis ticks and tick labels
g = g + theme(axis.ticks.x=element_blank(), axis.text.x=element_blank())
g

```



Highlighting engine manufacturer using the colour dimension

Although only an informal sort of analysis, our eye suggests that in this case Mercedes and Renault powered cars were competitive with each other, Ferrari powered cars were slightly behind, and the Cosworth powered teams were in a league of their own, though in the wrong direction!

The black and white print edition of this chart is unfortunately not so informative. Different typographic styles could perhaps help to distinguish between the different engine suppliers.



Finding Gaps

One of the things the above chart requires is the explicit *manual* placement of the lines that separate identifiable-to-the-eye groupings. It would be nice if we could come up with some sort of heuristic to automatically place these. For example, we might identify gaps between consecutive positions above a particular threshold (using something like a *second* difference value, `ddx$deltaDelta = c(0, diff(ddx$delta))` perhaps?) and place a line mid-way through the gap. To prevent grouping singletons, we would also need to count how many cars are in each grouping and perhaps *not* draw a line if there is less than particular number of cars in a group, or the difference between consecutive gaps (a *third difference*, calculated against gap values) falls below some threshold value. Automated class interval identifying clustering techniques such as the *Jenks Natural Breaks* algorithm for partitioning one dimensional datasets.

Hopefully these examples show how even the simplest data tables can be mined to produce a wide variety of different summary reports in both tabular and graphical form.

Sector Times (Prior to 2015)

As has already been mentioned, whilst the F1 website did provide sector time information from the practice sessions, when the results moved to the FIA website in 2015 the sector times for the practice sessions stopped being published. However, sector times for qualifying and race sessions associated with the current season continue to be made available from the FIA website as well as via the official PDF timing sheets. Consequently, analysis of F1 sector times for the practice sessions is now only possible for previous years using archived data grabbed originally from the F1 website prior to its redesign.

The data in the archived, historical practice session sectors tables contains the sector number, the driver number, their position in the session classification, their name, and their best recorded sector time in that session.

```
dbGetQuery(f1, ("SELECT * FROM p1Sectors LIMIT 5"))
```

sector	race	driverName	year	sectortime	driverNum	pos
1	AUSTRALIA	Jenson Button	2012	29.184	3	1
1	AUSTRALIA	Lewis Hamilton	2012	29.190	4	2
1	AUSTRALIA	Nico Rosberg	2012	29.514	8	3
1	AUSTRALIA	Michael Schumacher	2012	29.583	7	4
1	AUSTRALIA	Mark Webber	2012	29.645	2	5

As Ricardo Divila writes in *Going with your gut*, his RaceCar Engineering column from June 2014 (p5), “[s]ectors that repeat are ones without problems for drivers - the ones that vary are the difficult ones and show where car setup problems lie.” Unfortunately, the FIA don’t publish the full history of sector times, so this form of insight is not available to us. (Prior to 2014, the free FIA timing app had been a source of scrapeable sector times via third party timing apps that have reverse engineered the live timing data feed. As of the start of the 2014 season, this data seems no longer to be streamed through the free app.)

We work with what we do have, however. Inspecting the structure of the sector time data, we see that not all the columns are typed as naturally as we might like:

```
str(dbGetQuery(f1, ("SELECT * FROM p1Sectors LIMIT 5")))

## 'data.frame':      5 obs. of  7 variables:
## $ sector     : int  1 1 1 1 1
## $ race       : chr  "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" ...
## $ driverName: chr  "Jenson Button" "Lewis Hamilton" "Nico Rosberg" "Michael Schum\ 
acher" ...
## $ year       : chr  "2012" "2012" "2012" "2012" ...
## $ sectortime: chr  "29.184" "29.190" "29.514" "29.583" ...
## $ driverNum : chr  "3" "4" "8" "7" ...
## $ pos        : chr  "1" "2" "3" "4" ...
```

To correct this we need to cast the column types explicitly:

```
p1sectors=dbGetQuery(f1, ("SELECT * FROM p1Sectors"))
p1sectors$sectortime=as.double(p1sectors$sectortime)
p1sectors$pos=as.integer(p1sectors$year)
p1sectors$pos=as.integer(p1sectors$pos)
str(p1sectors)
```

```
## 'data.frame':      2911 obs. of  7 variables:
## $ sector      : int  1 1 1 1 1 1 1 1 1 ...
## $ race        : chr  "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" ...
## $ driverName: chr  "Jenson Button" "Lewis Hamilton" "Nico Rosberg" "Michael Schum\ 
acher" ...
## $ year         : chr  "2012" "2012" "2012" "2012" ...
## $ sectortime: num  29.2 29.2 29.5 29.6 29.6 ...
## $ driverNum   : chr  "3" "4" "8" "7" ...
## $ pos          : int  2012 2012 2012 2012 2012 2012 2012 2012 2012 ...
```

Using the individual sector times, we can calculate the *ultimate lap* for each driver as the sum of their best sector times.

To do this we need to generate the sum of the sector times recorded for each driver in each race of each year.

In pseudo-code, we might imagine a recipe for achieving this sort of operation taking the form:

```
for each year:
  for each race:
    for each driver:
      calculate the sum of the driver's sector times
```

An alternative way to approach the same calculation is to adopt a *split-apply-combine* strategy, as described in Hadley Wickham, “The split-apply-combine strategy for data analysis”, *Journal of Statistical Software* 40, no. 1 (2011): 1-29. Using the `plyr` library, we can call on a particular function, `ddply`, that allows us to split a data frame into groups based on the values of one or more columns, and then perform a summarising operation across the members of each grouping.

In this case, we need to split the data into groups corresponding to the data rows associated with each particular driver in each particular race of each year. This should result in three rows for each group, one row for each of the three sectors. The summarising operation we then need to perform is to summarise the data in each group by calculating the sum of the sector times within that grouping.

```
library(plyr)
ultimate=function(d) {
  ddply(d, c("year", "race", "driverName"),
    summarise,
    ultimate = sum(sectortime, na.rm=T))
}
ult = ultimate(p1sectors)
```

year	race	driverName	ultimate
2012	ABU DHABI	Daniel Ricciardo	106.490
2012	ABU DHABI	Felipe Massa	105.179
2012	ABU DHABI	Fernando Alonso	104.313
2012	ABU DHABI	Giedo van der Garde	124.504
2012	ABU DHABI	Heikki Kovalainen	107.317
2012	ABU DHABI	Jean-Eric Vergne	106.673

We can then compare the ultimate laptimes for each driver in a session to the best laptime they recorded in the session. (Note that it may not be possible to drive such an ultimate lap - the best sector time achieved for a particular sector may require taking a line that jeopardises another.)

Let's just check the data we can pull in from the session results tables:

```
p1results = dbGetQuery(f1, ("SELECT * FROM p1Results"))
p1results$laps = as.integer(p1results$laps)
str(p1results)

## 'data.frame':      986 obs. of  11 variables:
## $ driverNum : chr  "3" "4" "7" "5" ...
## $ time      : num  87.6 87.8 88.2 88.4 88.5 ...
## $ laps       : int  11 14 17 21 21 22 23 16 8 26 ...
## $ year       : chr  "2012" "2012" "2012" "2012" ...
## $ natGap     : num  0 0.245 0.675 0.8 0.907 ...
## $ gap        : num  0 0.245 0.675 0.8 0.907 ...
## $ race       : chr  "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" "AUSTRALIA" ...
## $ pos        : chr  "1" "2" "3" "4" ...
## $ driverName: chr  "Jenson Button" "Lewis Hamilton" "Michael Schumacher" "Fernand\o Alonso" ...
## $ team       : chr  "McLaren-Mercedes" "McLaren-Mercedes" "Mercedes" "Ferrari" ...
## $ natTime    : chr  "1:27.560" "1:27.805" "1:28.235" "1:28.360" ...
```

Graphically Comparing Ultimate and Actual Laptimes

One way of comparing the ultimate lap time with actual laptime for each driver in a particular race is to plot the result on to a two dimensional scatterplot. The easiest way to plot this is from a dataframe that contains the data corresponding to the ultimate lap time in one column and the actual laptime in another.

At the moment, we have the data in two separate tables, *ult* and *p1results*. We can merge the data into a single dataframe using the *year*, *race* and *driverName* columns as merge keys:

```
p1results_merge = merge(p1results, ult, by=c("year", "race", "driverName"))
head(p1results_merge, n=5)
```

```
##   year      race      driverName driverNum     time laps natGap   gap
## 1 2012 ABU DHABI Daniel Ricciardo      16 106.649  24  3.364 3.364
## 2 2012 ABU DHABI Felipe Massa          6 105.567  24  2.282 2.282
## 3 2012 ABU DHABI Fernando Alonso        5 104.366  21  1.081 1.081
## 4 2012 ABU DHABI Giedo van der Garde    21  0.000   3  0.000 0.000
## 5 2012 ABU DHABI Heikki Kovalainen     20 107.418  23  4.133 4.133
##   pos      team natTime ultimate
## 1 17 STR-Ferrari 1:46.649 106.490
## 2 11 Ferrari 1:45.567 105.179
## 3  4 Ferrari 1:44.366 104.313
## 4 24 Caterham-Renault No time 124.504
## 5 19 Caterham-Renault 1:47.418 107.317
```

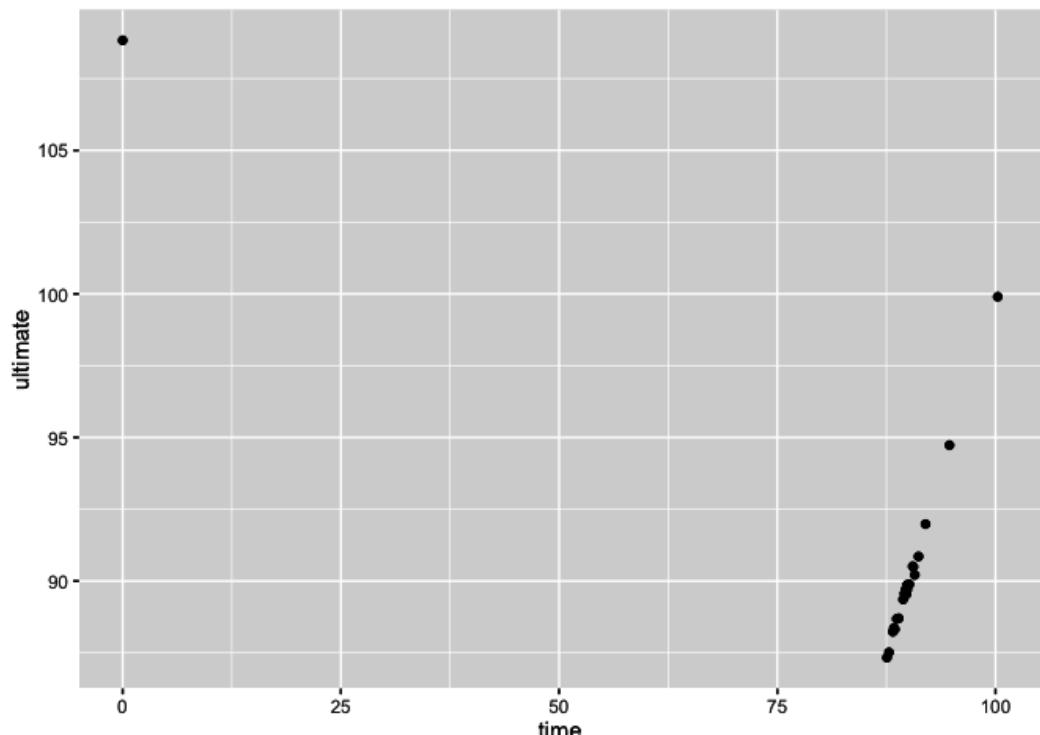
We can then plot directly from the merged dataset. Let's take a subset of the data, focusing on the 2012 Australian Grand Prix:

```
gp_2012_aus_p1_results = subset(p1results_merge, year=='2012' & race=="AUSTRALIA")
#Remove the year and race columns from the display
```

	driverName	driverNum	time	laps	natGap	gap	pos	team	natTime	ultimate
25	Bruno Senna	19	89.953	21	2.393	2.393	14	Williams-Renault	1:29.953	89.832
26	Charles Pic	25	100.256	11	12.696	12.696	22	Marussia-Cosworth	1:40.256	99.901
27	Daniel Ricciardo	16	88.908	23	1.348	1.348	7	STR-Ferrari	1:28.908	88.701

	driverName	driverNum	time	laps	natGap	gap	pos	team	natTime	ultimate
28	Felipe Massa	6	90.743	11	3.183	3.183	18	Ferrari	1:30.743	90.212
29	Fernando Alonso	5	88.360	21	0.800	0.800	4	Ferrari	1:28.360	88.360
30	Heikki Kovalainen	20	90.586	16	3.026	3.026	17	Caterham Renault	1:30.586	90.494

```
library(ggplot2)
ggplot(gp_2012_australia_p1_results) + geom_point(aes(x=time, y=ultimate))
```



Raw scatterplot showing ultimate versus actual laptime for each driver

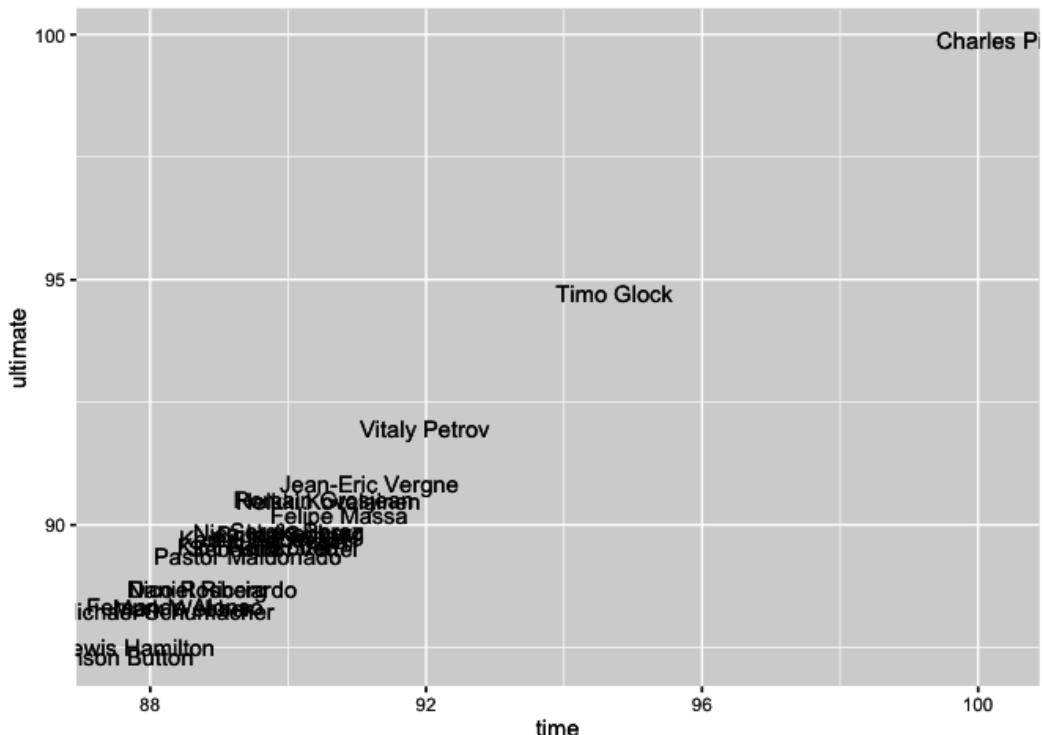
So what's wrong with this chart? Two things immediately come to mind. Firstly, there is an outlier that skews the lower limit of the horizontal *x-axis*: one of the drivers appears not to have a session time recorded. Unfortunately, which don't know which driver this time occurs for, which leads to the second problem: which point corresponds to which driver?

One way of addressing the outlier problem is to filter out drivers for whom no time is recorded in the session (if they do record a time, they will also necessarily have separate sector times, and hence an ultimate laptime, recorded). We can also check that the time is not recorded as absent, that is, as NA.

```
gp_2012_aus_p1_results = subset(gp_2012_aus_p1_results, time>0 & !is.na(time) )
```

To address the other issue, that of not knowing which driver each point refers to, we can instead use a text plot. This requires using an extra aesthetic parameter, `label`, that identifies which column's values should be displayed as the text label for each plotted marker.

```
g = ggplot(gp_2012_aus_p1_results, aes(x=time, y=ultimate, label=driverName))
g + geom_text()
```



Raw text plot showing ultimate vs best laptime for each driver by name

As a finished product, this chart obviously has some issues: the labels are large, and overlap, making them, and their registration point (that is, the co-ordinate value they are located at) unclear; the labels also overflow the chart's plotting area; and so on. However, as a quick, preliminary sketch, it does provide us with something we can start to work with and gain information from.

So for example, the chart suggests that the ultimate times broadly follow the session times, which makes sense. However, it's virtually impossible to tell whether a driver's session time matched their ultimate time, or whether it was some way away from it. Several factors contribute to this lack of clarity:

- we can't tell what is being used as the registration point for each label - that is, which part of the label marks the `(time, ultimate)` co-ordinates.
- the length of the labels covers a wide range. If the registration point is the mid-point of the label, where is that exactly?
- the font size used for the labels is quite large, meaning that labels obscure each other;
- the name labels have overflowed the plotting area;
- some of the labels appear to fall outside the area displayed by the chart, making them difficult to read;
- it's hard to tell where the line corresponding to equal ultimate and session laptimes lies. The grid is probably too coarse grained to be able to take accurate measurements for each marker, even if we could tell where the registration point is.

Let's work through the problems one at a time. `ggplot` supports layering in plots, with the layer order determined by the order in which layers are added to the plot. By default, the first layer is the lowest layer, the last layer the highest. We can mark the registration point using a `geom_point()` layer.

Let's add the point *underneath* the corresponding label. This means adding the `geom_point()` to a lower level than the `geom_text()` by adding it to the `ggplot` chart *before* the `geom_text()`. To make things easier to read, we might also reduce the size of the text labels.

Whilst we could identify the values to be used as `x` and `y` aesthetics in each layer, we can also declare them in the base plot and allow their values to be inherited by the chart layers

We can also assign the plot to a variable, and build it up a layer at a time, before plotting the final compound chart.

```

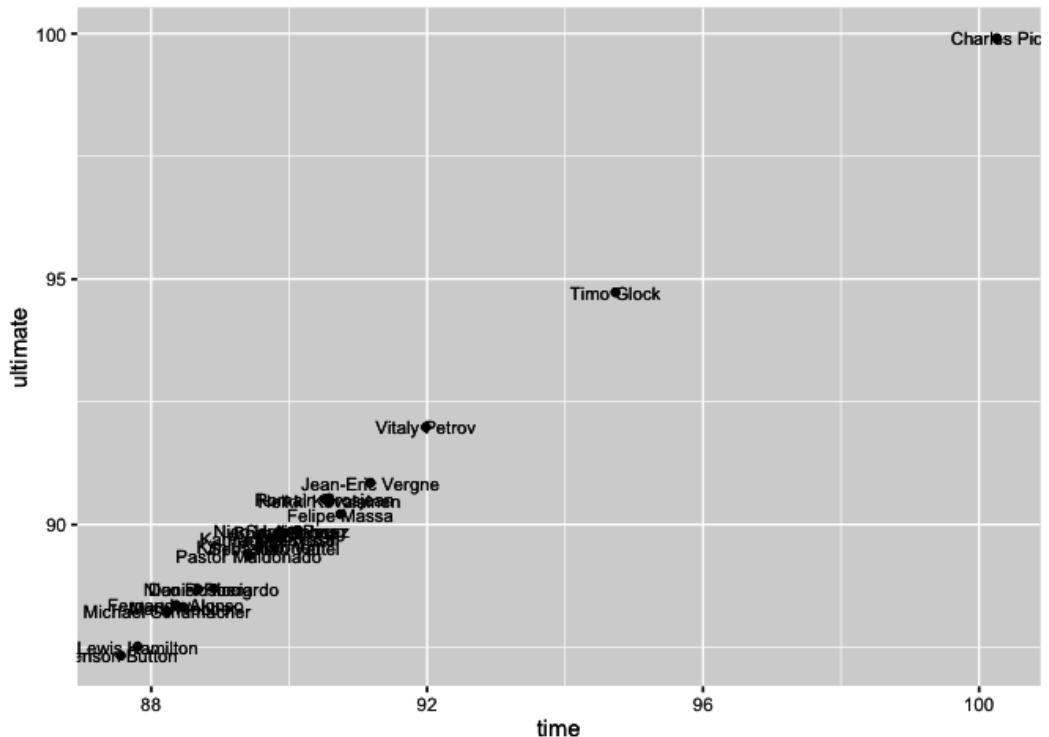
g = ggplot(gp_2012_aus_p1_results, aes(x=time, y=ultimate))

#Add the points layer
g = g + geom_point()

#Add the text layer on top of the chart
g = g + geom_text(aes(label=driverName), size=3)

#Plot the chart
g

```



Ultimate vs actual laptime with point and text label layers

Interactive Charts

Interactive charts provide one way of getting around certain layout issues by allowing

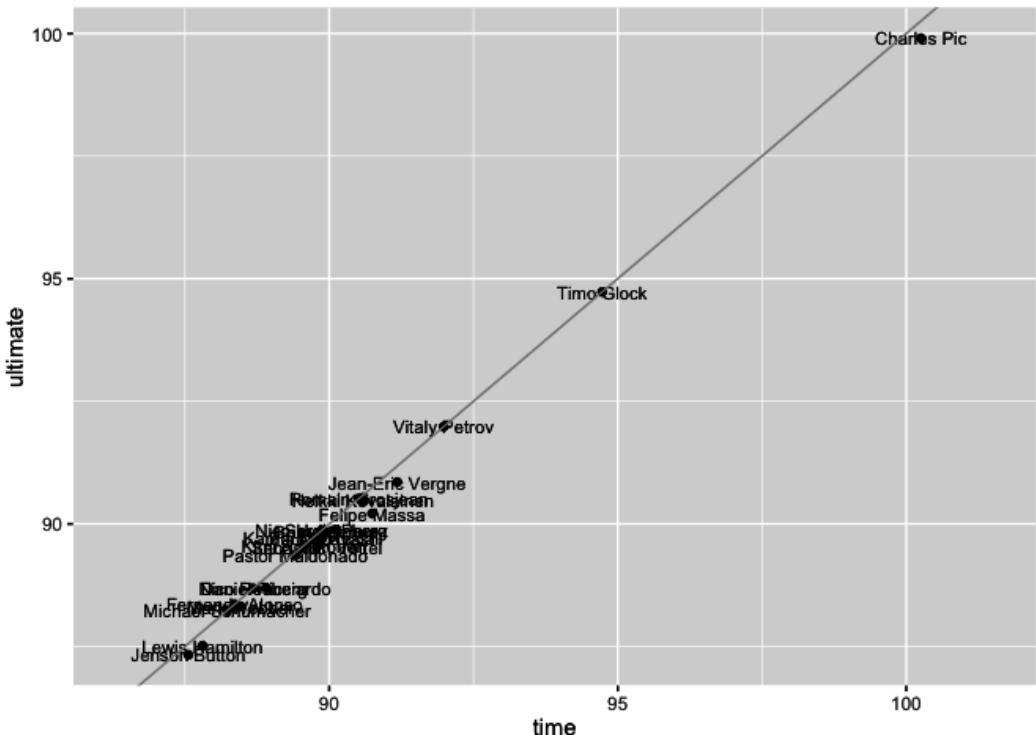
the user to select or highlight particular areas of the chart or elements within it in order to display additional information, such as x/y co-ordinate values for a selected point. Interactive Javascript charts can be produced from a variety of R packages such as rCharts^a or HTMLWidgets^b, but they will not be described further within this book...

^a<http://rcharts.io/>

^b<http://www.htmlwidgets.org/>

To find the line on which the best laptime *equals* the ultimate laptime, we need to draw a line x=y. In ggplot, we can use a `geom_abline()` to draw such a line. By default, the colour of the line is black, but that will dominate the chart somewhat so I am going to give it a grey colour. The `expand()` argument to `scale_x_continuous()` provides multiplicative and additive expansion constants that allow the text labels to be placed some distance away from the chart's axes.

```
g + geom_abline(col='grey') + scale_x_continuous(expand=c(0,2))
```



Addition of line showing equivalent ultimate and actual laptimes

Note that this line is added in the layer *above* the previous chart elements, so the line will be placed *on top of* the marked points. If we want the `geom_point()` elements to sit on top of the equivalence line, we need to add the `geom_point()` layer to the chart *after* the `geom_abline()` component.

It may seem picky, but it's important to take care when reading a chart so that you read it as you intended. This is one of the great challenges of data visualisation - producing charts that have what we might term a "natural reading" that makes sense 'at a glance' and doesn't catch out the unwary reader.

In the above example, it may make sense to think of the best laptime as equaling or *falling short* of the ultimate laptime. If the point lies on the equivalence line, the laptime equals the ultimate time. So how do we read this chart to find how far off the ultimate time a particular laptime was?

Consider this example - laptime = 100, ultimate time = 95.

Would the chart be any easier to interpret if we were to swap the axes, so that the ultimate

time was reported along the horizontal *x-axis* and recorded times on the vertical *y-axis*? Which provides the most “natural” reading?

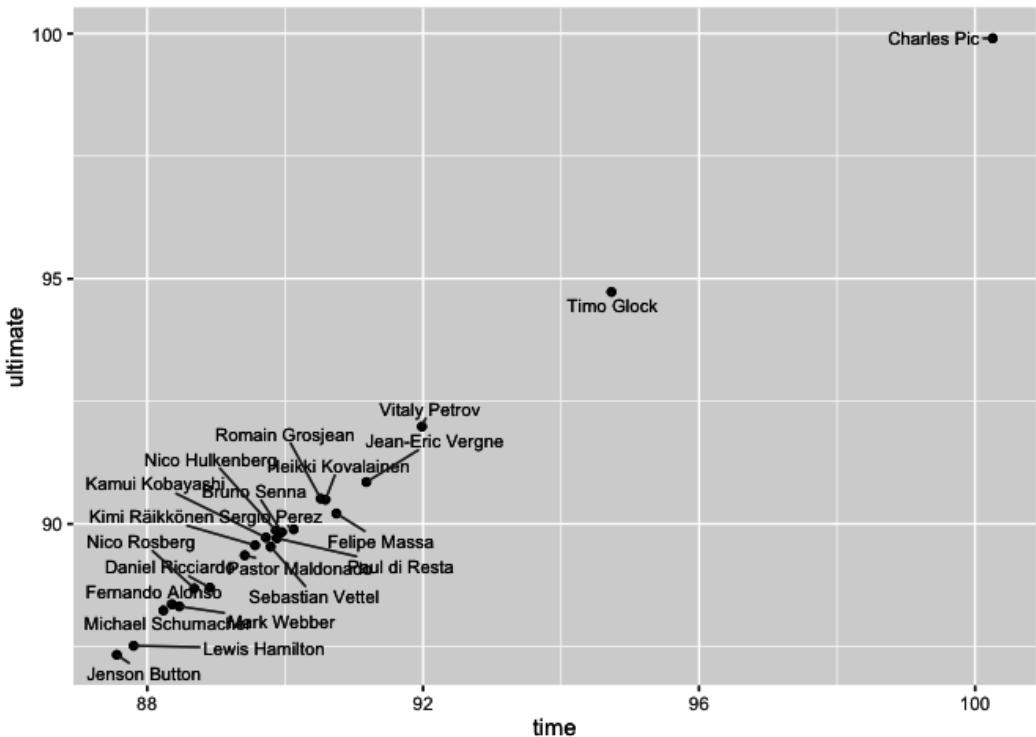
Improving the Readability of the Ultimate vs. Actual Laptime Chart

The close times, both ultimate and actual, that are recorded for each of the drivers makes the chart difficult to read as the driver name labels overlap and occlude each other. The `ggrepel` add-on to `ggplot2` provides a new function, `geom_text_repel()` that allows labels to *repel* each other so that they don’t overlap. Short line segments connect the each label to its *registration point* (that is, the x/y co-ordinate it would ordinarily be located at).

```
#At the time of writing, the package needs to be installed from github
#install.packages("devtools")
#devtools::install_github("slowkow/ggrepel")
library(ggrepel)

g = ggplot(gp_2012_aus_p1_results, aes(x=time, y=ultimate))
#Add the points layer
g = g + geom_point()

#Add a dodged text layer on top of the chart
g + geom_text_repel(aes(label=driverName), size=3)
```



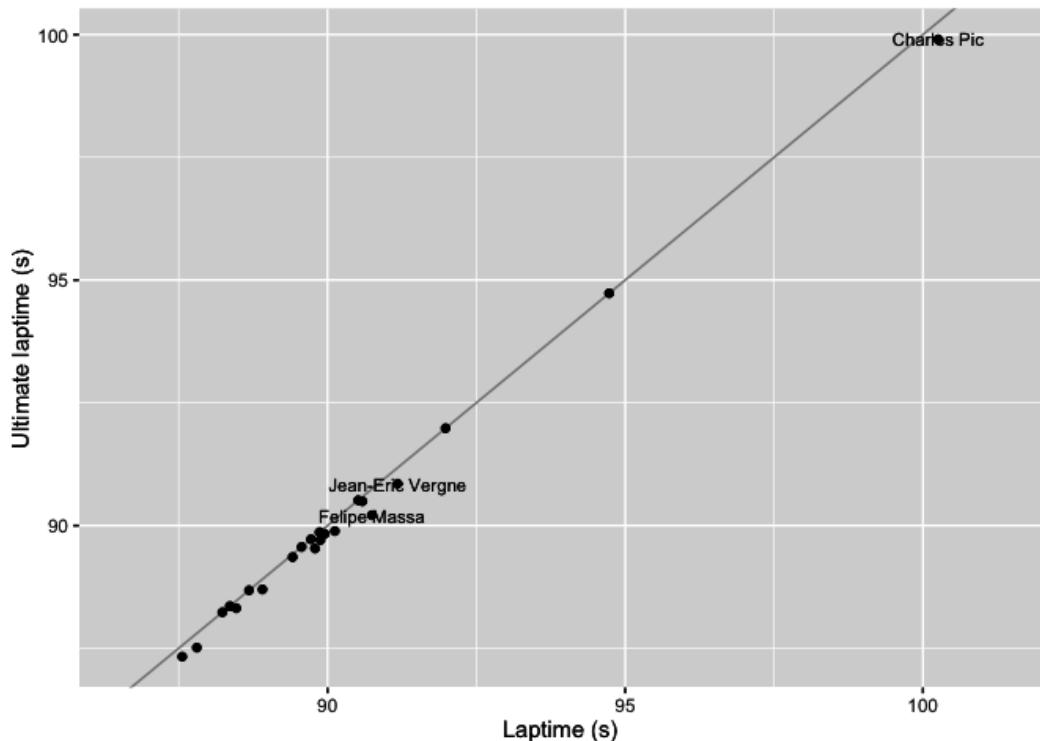
Using the `ggrepel` library to avoid text label overlaps

Whilst it is easy to label every point in a chart, and tools such as `geom_text_repel()` allow us to avoid the problem of overlapping labels, text plot charts can soon become cluttered, particularly if there is a large number of points or labelled points are close to each other. In some cases, it may make sense to only label points that we are interested in, or that have an interesting property or feature associated with them. For example, if we are looking for drivers that have missed their ultimate laptime by a considerable margin, having labels for *each* driver displayed confuses the view. How about if instead we just label the drivers who miss the ultimate laptime by more than a specified amount, such as a two or three tenths or more?

```

g = ggplot(gp_2012_aus_p1_results, aes(x=time, y=ultimate))
#Add a line to show where the actual and ultimate times are equivalent
g = g + geom_abline(col='grey')
#Mark each laptime on the layer above that equivalence line
g = g + geom_point()
#Add the text layer on top of the chart for drivers that miss their ultimate
#laptime by at least 0.3s
#We can also reduce the size of the text label a little
g = g + geom_text(data=subset(gp_2012_aus_p1_results, (time-ultimate)>=0.3),
                   aes(label=driverName), size=3)
g = g + xlab('Laptime (s)') + ylab('Ultimate laptime (s)')
#Extend the x axis limits to prevent labels at the extremes overflowing
tmp.x = gp_2012_aus_p1_results$time
g + scale_x_continuous(limits=c(min(tmp.x)-1, max(tmp.x)+1))

```



Highlighting drivers more than 0.3s away from their ultimate laptime

In addition to using times for the x and y axes, we could also plot rankings, where the axis

marks are discretely spaced rather than plotted against a continuous number line. Within a particular class of laptime, such as the ultimate time, we might even plot the ultimate rank on one axis against the ultimate laptime on the other, providing yet another view on to how drivers compared in their ultimate lap performances.

ggplot2 is a very flexible charting library, supporting a range of co-ordinate transformations that can be used to create a wide variety of chart types. For example, if you are feeling adventurous, you could try plotting a radar chart showing team ranking for each sector, perhaps inspired by this post on creating a radar chart⁴¹ (a.k.a. star plot; spider plot) using ggplot2 in R⁴¹ on Stack Overflow.

As well as exploring alternative chart types, we can also generate a range of other reference laptimes against which we can compare performances. For example, setting:

```
r=gp_2012_aus_p1_results
```

we can generate:

- the best overall session laptime, `min(r$time)`: 87.56;
- the best overall session laptime in a team, `min(r[r$team=='Williams-Renault',]$time)`: 89.415;
- the best ultimate lap in a team: `min(r[r$team=='Williams-Renault',]$ultimate)`: 89.358;

If we go back to the original textualisation data, we can calculate yet more reference times, although this may require joining one data set with another, either by merging datasets in R, or by using more complex queries onto the original database.

For example, we might look for the ultimate lap for a team (based on the best recorded sector times across a team) or the ultimate session lap, based on the best sector times recorded across all drivers.

Summary

In this chapter, we have started to explore a wide range of summary reports on the practice sessions. For example, we have generated scatterplots showing laptimes against classification

⁴¹<http://stackoverflow.com/questions/9614433/creating-radar-chart-a-k-a-star-plot-spider-plot-using-ggplot2-in-r>

rank and vertical and horizontal bar charts to show laptime differences between consecutively classified cars. We have also started to look at the generation of novel custom charts, such as the charts that depict laptime differences relative to the fastest lap in a session, or the charts that start to explore team performance.

An analysis of best sector times allows to ask whether each driver managed to drive their ultimate lap during a session, or whether their performance fell short of it and they left time on the track, to mix metaphors! Unfortunately, as of the start of the 2015 season, sector times for F1 practice sessions are no longer available, so we must limit our sector time based analyses to just the qualifying and race sessions.

Stylistically, many of the charts still appear to have low production values - labels overlap or even overflow a chart's borders. In the following chapters, you will see several strategies for overcoming these failings and improving not only the visual attractiveness of similar charts, but also their readability and usefulness.

Practice Session Utilisation

For the teams, the three practice sessions that open up a race weekend provide an opportunity to set the car up for the weekend. For the data junkie, timing sheets published by the FIA provide details of all the laptimes recorded by each driver. Using this data, we can chart the extent to which each driver made use of the session, perhaps start to generate some sort of performance model from their long run laptimes, and look at how well each driver's laptimes compare to their ideal lap for the session.

Unfortunately, practice session laptimes are not published in a convenient form as data. Instead, the data must be scraped from the FIA timing sheets, either using a custom written scraper, or a PDF data scraping tool such as PDFTables (online service)⁴² or Tabula (free/open source desktop tool)⁴³.

On the practice session timing sheets, the first “laptime” is actually the time of day the driver completed their first lap of the session.

⁴²<https://pdftables.com/>

⁴³<http://tabula.technology/>



Third Practice Session Lap Times

First recorded time is time of day

3 D. RICCIARDO

NO	TIME	NO	TIME
1	14:06.24	11	1:45.480
2	1:42.045	12	1:45.431
3	2:01.914	13	1:45.157
4 P	1:50.852	14	1:45.673
5	12:57.457	15	1:47.444
6	1:40.590	16 P	1:51.737
7 P	1:55.674	17	2:39.160
8	1:48.651	18	1:45.242
9	1:44.552	19 P	1:50.482
10	1:43.158		

5 S. VETTEL

NO	TIME	NO	TIME
1 P	14:02.25	8 P	2:46.508
2	25:36.133	9	15:23.683
3	1:40.532	10	1:40.266
4	1:45.096	11	1:59.519
5	1:41.290	12	1:40.785
6	1:41.451	13 P	2:00.237
7 P	1:56.522		

6 N. ROSBERG

NO	TIME	NO	TIME
1 P	14:04.53	8 P	1:54.416
2	19:24.182	9	14:41.831
3	1:40.392	10	1:39.690
4	2:14.987	11	2:16.609
5	1:40.252	12 P	1:48.316
6	2:15.126	13 P	2:40.467
7	1:40.902		

Pit at end of lap

7 K. RAIKKONEN

NO	TIME	NO	TIME
1 P	14:02.36	6	1:40.245
2	9:55.307	7	2:02.275

8 R. GROSJEAN

NO	TIME	NO	TIME
1 P	14:02.43	9	1:49.326
2	9:42.674	10	1:42.468

9 M. ERICSSON

NO	TIME	NO	TIME
1	14:06.15	11	1:43.980
2	1:43.588	12	1:43.757

Example FIA F1 Practice session lap time timing sheet, Copyright FIA, 2015

If we find the earliest recorded time during the session, we can use it as a zero basetime against which to record the time into the session at which each laptime was recorded. That is, we take the time of day recorded by the driver who first recorded a laptime during the session (even if that was just a sighting lap from which they returned straight to the pits) as a baseline session time of 0 and then calculate all other start times relative to that. When talking about *time into the session*, this is relative to the baseline time, *rather than the time the lights went green on the session* (that data does not appear in the timing sheets).

A PDF scraper can be used to extract timing data from the session laptimes timing sheet (example data file⁴⁴). We can then present the data in the following form:

⁴⁴<https://gist.githubusercontent.com/psychemedia/11187809/raw/f12015test.csv>

```
#Data available at:
#https://gist.githubusercontent.com/psychomedia/11187809/raw/f12015test.csv
ptimes = read.csv("f12015test.csv")
```

lapNumber	laptme	name	number	pit	stime
1	00:05:59	D. RICCIARDO	3	False	359.000
2	1:41.799	D. RICCIARDO	3	False	101.799
3	1:55.667	D. RICCIARDO	3	True	115.667
4	17:26.076	D. RICCIARDO	3	True	1046.076
5	12:43.618	D. RICCIARDO	3	True	763.618
6	23:23.912	D. RICCIARDO	3	True	1403.912

The `lapNumber` is the lap count within the session for each driver; the `laptme` is the laptme in minutes, seconds and milliseconds *apart from the first time recorded for each driver*, which is the time (in hours, minutes, seconds) relative to the time of day first recorded in the session; the `name` and `number` are the driver's name and racing number; the `pit` flag says whether the driver pitted on that lap; and the `stime` is the laptme (or, for the first lap, the time since the first recorded time in the session) measured in seconds and milliseconds.

We can augment the scraped data with three letter driver codes using a recipe such as the following:

```
ptimes$name=factor(ptimes$name)
#http://en.wikipedia.org/wiki/Template:F1stat
driverCodes=c("L. HAMILTON"= "HAM", "S. VETTEL"= "VET", "N. ROSBERG"= "ROS",
            "D. RICCIARDO"= "RIC", "D. KVYAT"= "KVV", "M. VERSTAPPEN"= "VES",
            "F. MASSA" = "MAS", "R. GROSJEAN"= "GRO", "V. BOTTAS"= "BOT",
            "M. ERICSSON"= "ERI", "K. RAIKKONEN"= "RAI",
            "P. MALDONADO" = "MAL", "N. HULKENBERG"= "HUL",
            "S. PEREZ"= "PER", "C. SAINZ"= "SAI", "F. NASR"= "NAS",
            "J. BUTTON" = "BUT", "F. ALONSO"= "ALO", "R. MERHI"= "MER",
            "W. STEVENS"="STE")
driverCode=function(name) unname(driverCodes[name])
ptimes['code']=apply(ptimes['name'],2,function(x) driverCode(x))
```

Whilst the scraped data may appear to offer slim pickings at first glance, we can actually derive several additional columns from it, including a reckoning of the stints completed by each driver, the lap number in each stint, the best laptme recorded so far by a particular driver ("green" times), and the best laptme recorded overall so far in to a session ("purple" lap times).

Note - for the purposes of charts the session utilisation charts, there are several different approaches we may take towards displaying green laps. For example, we might display each driver's best time in the session overall as their only green lap (unless they also get the session best overall time, which we'd display as purple), which helps us identify a driver's best time in the session; or we might display their best time in each stint as green; or we might display their best time so far in the session as green, which would allow us to see progression throughout the session.



Exercise

At first glance, the practice laptime data may appear to offer slim pickings as a basis for producing a wide range of informative charts. But there is actually quite a lot of additional data we can derive from it, such as green or purple laptimes, howsoever defined.

What other information do you think you might be able to derive just from the raw laptime data?

Session Utilisation Charts

Session utilisation charts use a chart design that is intended to provide an *at a glance* overview of how each driver used the track during a particular session. The chart plots when drivers complete a lap against elapsed session time, clearly showing run durations, as well as pit, green and purple laptime information.

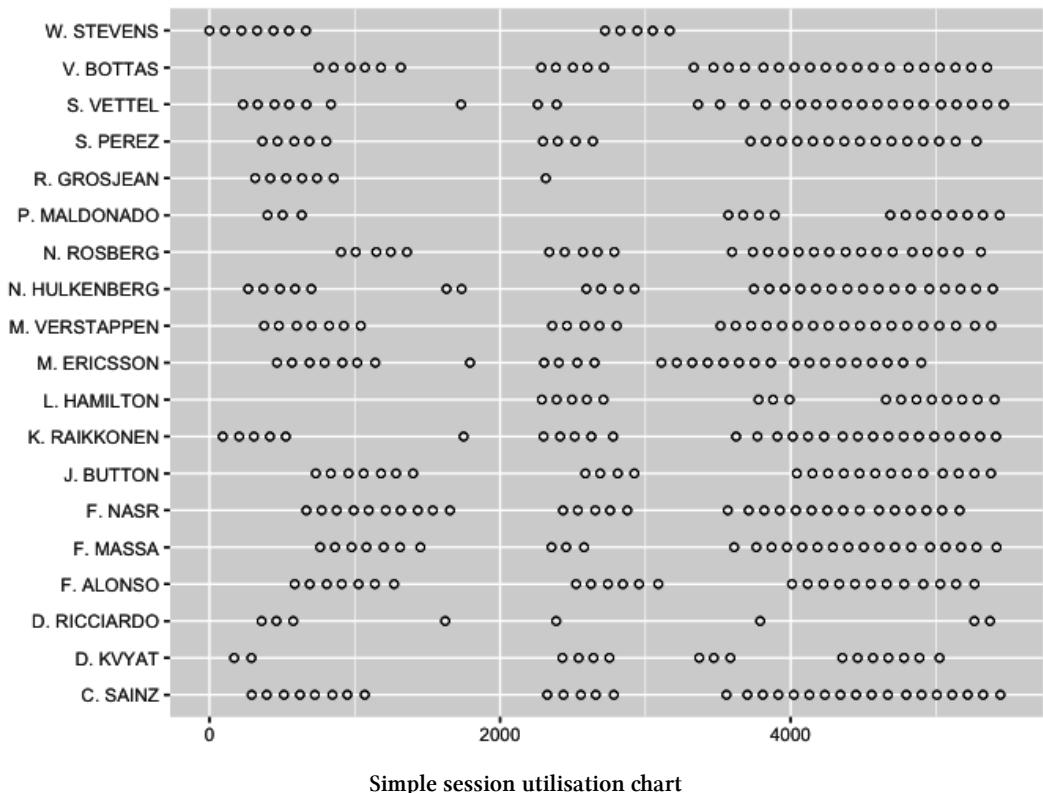
To begin with, we can easily generate an accumulated session time for each driver to record the time into the session they recorded each laptime. This is simply a cumulative sum of laptimes (`stime`) taken over the laptimes, sorted by lap number, recorded by each driver.

```
library(plyr)
ptimes=arrange(ptimes,name,lapNumber)
ptimes=ddply(ptimes,.(name),transform,cuml=cumsum(stime))
```

```
##   lapNumber    laptime      name number    pit    stime code    cuml
## 1           1 00:04:51 C. SAINZ      55 False 291.000  SAI 291.000
## 2           2 1:42.671 C. SAINZ      55 False 102.671  SAI 393.671
## 3           3 1:58.929 C. SAINZ      55 False 118.929  SAI 512.600
## 4           4 1:50.393 C. SAINZ      55 False 110.393  SAI 622.993
## 5           5 1:42.517 C. SAINZ      55 False 102.517  SAI 725.510
## 6           6 2:00.413 C. SAINZ      55 False 120.413  SAI 845.923
```

Using the accumulated session time, we can plot a basic *session utilisation chart* that shows when each driver completed a lap time in the session.

```
library(ggplot2)
g=ggplot(ptimes) + geom_point(aes(x=cuml, y=name), pch=1)
g = g + xlab(NULL)+ylab(NULL)
g
```



The spacing *between* marks along a row indicates the time between the completion of one lap and the next; that is, the lap time, or lap + pit time.

We can clearly see how the laps completed by each driver are grouped together in separate stints. In a later section, we'll see how we might automatically label the laps in each stint with a corresponding stint number.

We can further annotate the session utilisation chart by explicitly recognising those laps on which a particular driver pitted.

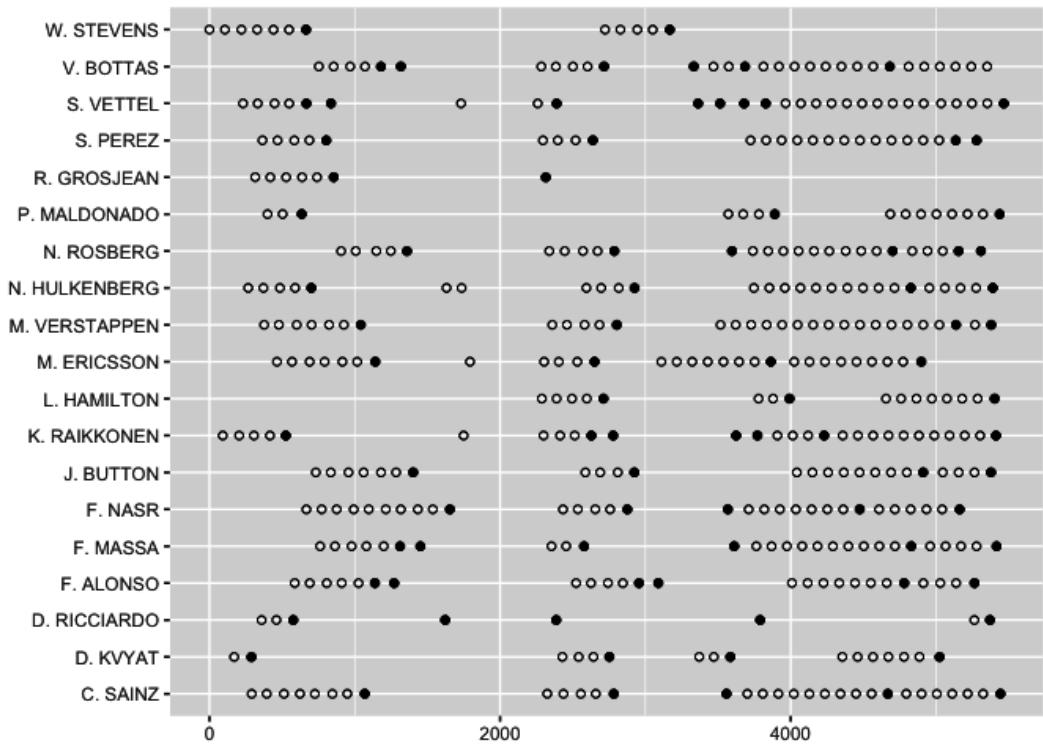
For convenience, convert the pit *True/False* flag to a proper R boolean value:

```
ptimes['pit'] = ptimes['pit']=='True'
```

Although the above formulation may look a little clumsy, it is just a simple assignment of a (calculated) Boolean value. The value that is assigned is a True or False value based on a test of the ptimes['pit'] value is set to the string value 'True'.

We can now annotate the previous session utilisation chart with another layer that indicates whether a driver pitted at the end of that particular lap by means of a solid, black filled, circle symbol:

```
g+geom_point(data=ptimes[ptimes['pit']==TRUE,],aes(x=cum1,y=name))
```



Add highlights to identify those laps on which a driver pitted

Looking at the chart, there appear to be a couple of anomalies, such as in the case of Raikkonen and Ericsson where we see a couple of apparently long lap times (wide spacings along the x-axis) associated with isolated laps (one lap stints) that do not appear to finish with a pit stop (that is, a solid, black circle). (We might reasonably expect a long lap time to follow a pit stop, because this out lap time will include the time spent in the pits. If we have a stint that appears following a stint that did not appear to end with a pit stop, it suggests that the car spent a long time out on track, perhaps as a result of one or more practice starts; or was perhaps recovered and did not cross a timing line that identified a pit event; or maybe the session was red flagged and timing of laps suspended as cars are forced to return to the pits.) We can check back with the original timesheet to check that the visualised view does, in fact, represent a fair depiction of the original data:

7 K. RAIKKONEN

NO	TIME	NO	TIME
1	14:03:51	15	1:45.245
2	1:53.853	16	1:45.348
3	1:41.002	17 P	1:49.990
4	1:50.194	18	2:08.374
5 P	1:49.139	19	1:44.921
6	20:23.524	20	1:44.597
7	9:10.261	21	1:45.086
8	1:54.600	22	1:45.253
9	1:40.163	23	1:44.956
10 P	1:54.613	24	1:45.048
11 P	2:29.078	25	1:45.152
12 P	14:06.953	26	1:45.282
13 P	2:26.733	27	1:45.424
14	2:17.803	28 P	1:49.384

8 R. GROSJEAN

NO	TIME	NO	TIME
1	14:07:35	5	1:42.948
2	1:43.663	6 P	1:54.813
3	1:49.174	7 P	24:19.594
4	1:49.424		

9 M. ERICSSON

NO	TIME	NO	TIME
1	14:10:05	16	1:46.076
2	1:42.877	17	1:48.046
3	2:01.984	18	1:47.275
4	1:42.483	19	1:47.431
5	2:02.298	20 P	1:51.892
6	1:42.478	21	2:39.978
7 P	2:03.356	22	1:45.726
8	10:52.470	23	1:46.800
9	8:30.389	24	1:50.854
10	1:41.261	25	1:46.190
11	2:07.648	26	1:46.452
12 P	1:57.997	27	1:47.017
13	7:39.907	28	1:47.401
14	1:46.232	29 P	2:04.493
15	1:46.496		

Checking odd looking times in the original timing sheet data

Indeed, we see Ericsson's isolated lap 8 is longlived without apparently ending in a pit stop, as is Raikkonen's lap 6. To find out what actually happened at the end of those laps, we'd need to check other sources of information. One thing we might note from the session utilisation chart in this particular case is that all these "incomplete stints" appear at about the same time in the session. Checking back against session live reports for the particular session depicted above, we see that indeed the session was red flagged around that time.

Colour Codes in Motorsport Timing Screens

Over the years, conventional colours have come to be used on motorsport timing screens. The following description is taken from an old version of the f1.com website and describes the conventions used for the timing screens at that time:

Yellow Default display colour. Red Indicates cars exiting and entering the pits. When a car leaves the pits the word OUT is displayed in the time column. The car number and driver name appears in red until that car goes through the first sector when it will revert to its normal colour. When a car enters the pits the words IN PIT are displayed in the time column. The car number and driver name is shown in red and reverts to its normal colour after a short period of time. White Indicates the most recent information available for that driver (e.g. a white sector time is the information for the sector just completed). Green Indicates a personal best for that driver and may relate to individual sector times as well as a lap time. Magenta Indicates the overall best in that session and may relate to individual sector times as well as a lap time. Grey Used in qualifying only. Indicates drivers who have been knocked out in the first or second qualifying phase and

are therefore no longer eligible to participate in the session. _STOP_ Appears in place of the sector information in the event of a car not completing that sector, indicating that in all probability the car has stopped on the circuit. _OUT_ Appears in place of the sector information to indicate that a car has just exited the pits. _IN PIT_ Appears in place of the sector information to indicate a car has just entered the pits. A line showing the personal best time in each sector for that car in the current session is displayed in yellow after the car has been in the pits for 15 seconds.

The screenshot below shows an example timing screen, taken from the Tata F1 Innovation Connectivity Prize Challenge Brief, 2014

				Position	Car number	Driver name	Fastest lap time	Gap to the leader's fastest lap	Sector 1 time for the current lap	Sector 2 time for the current lap	Sector 3 time for the current lap	Number of laps
							BEST LAP	GAP				
1	6	N. ROSBERG	1:25.887						23.5	32.4	29.8	16
2	44	L. HAMILTON	1:26.756	0.869					23.5	32.5	30.4	9
3	14	F. ALONSO	1:27.188	1.301					25.7			14
4	19	F. MASSA	1:27.223	1.336					23.6	32.9	30.5	10
5	8	R. GROSJEAN	1:27.682	1.795					28.6			15
6	20	K. MAGNUSEN	1:27.806	1.919					26.2	36.0		14
7	3	D. RICCIARDO	1:27.808	1.921					24.0	32.8	30.9	10
8	22	J. BUTTON	1:28.006	2.119					24.2			10
9	13	P. MALDONADO	1:28.076	2.189					28.2			17
10	1	S. VETTEL	1:28.085	2.198					25.3	37.4	31.9	18
11	25	J. VERGNE	1:28.242	2.355					29.1			14
12	26	D. KVYAT	1:28.298	2.411					23.7	33.3	31.1	14
13	7	K. RAIKKONEN	1:28.419	2.532					24.3	33.9		13
14	99	A. SUTIL	1:28.715	2.828					24.1	33.5	30.9	14
15	11	S. PEREZ	1:28.720	2.833					24.1	33.2		11
16	77	V. BOTTAS	1:28.733	2.846					24.1	33.1		10
17	27	N. HULKENBERG	1:28.751	2.864					27.4			11
18	21	E. GUTIERREZ	1:29.804	3.917					24.2			15
19	4	M. CHILTON	1:30.169	4.282					24.6		STOP	15
20	17	J. BIANCHI	1:30.670	4.783					24.6	34.2	31.6	12
21	10	K. KOBAYASHI	1:31.195	5.308					24.7	34.1		16
22	9	M. ERICSSON	1:31.800	5.913					24.7	34.5		16

F1 Timing screen - image taken from Tata F1 Innovation Connectivity Prize Challenge Brief, 2014

Finding Purple and Green Times

Another way of annotating the original laptime data is to find the “cumulative” (rolling) best (quickest) time for each driver in the session (that is, the smallest laptime so far (rolling minimum laptime) for each driver in that session) and highlight it as such. Noting that the first laptime will be an outlap rather than a recorded flying lap time, and further that this value will be set to zero for the driver who was first to collect a time in the session, we omit the first laptime for each driver and instead set it to a high dummy value.

```
#Sort by driver and driver's lap number
ptimes=arrange(ptimes, name, lapNumber)
#Find best time so far in each session for each driver
ptimes=ddply(ptimes,
             .(name),
             transform,
             driverbest=cummin(c(9999, stime[2:length(stime)])))
```

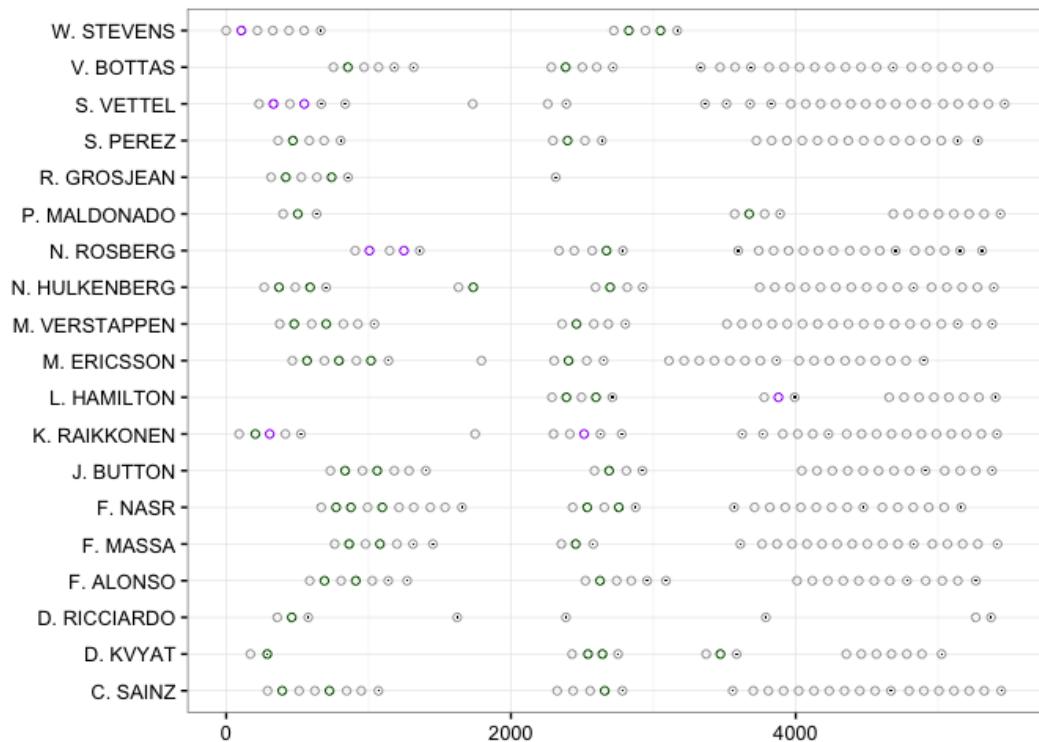
If we now sort by accumulated session time in which the rows are ordered according to the order in which the laps were completed across all drivers, we can identify “purple” times (that is, the best of all the drivers’ best laps so far, which is to say, the fastest lap time set so far in the session) and “green” laptimes (a driver’s personal best laptime in the session that isn’t a purple time).

```
ptimes=arrange(ptimes, cuml)
ptimes['purple']=sapply(ptimes['driverbest'], cummin)
#TO CHECK - do we need to trap green with & !df['pit'] & !df['outlap'] ?
ptimes['colourx']=ifelse(ptimes['stime']==ptimes['purple'],
                         'purple',
                         ifelse(ptimes['stime']==ptimes['driverbest'],
                                'green',
                                'black'))
ptimes=arrange(ptimes, name, 1apNumber)
```

Once again, we reset the order of the rows once we have done.

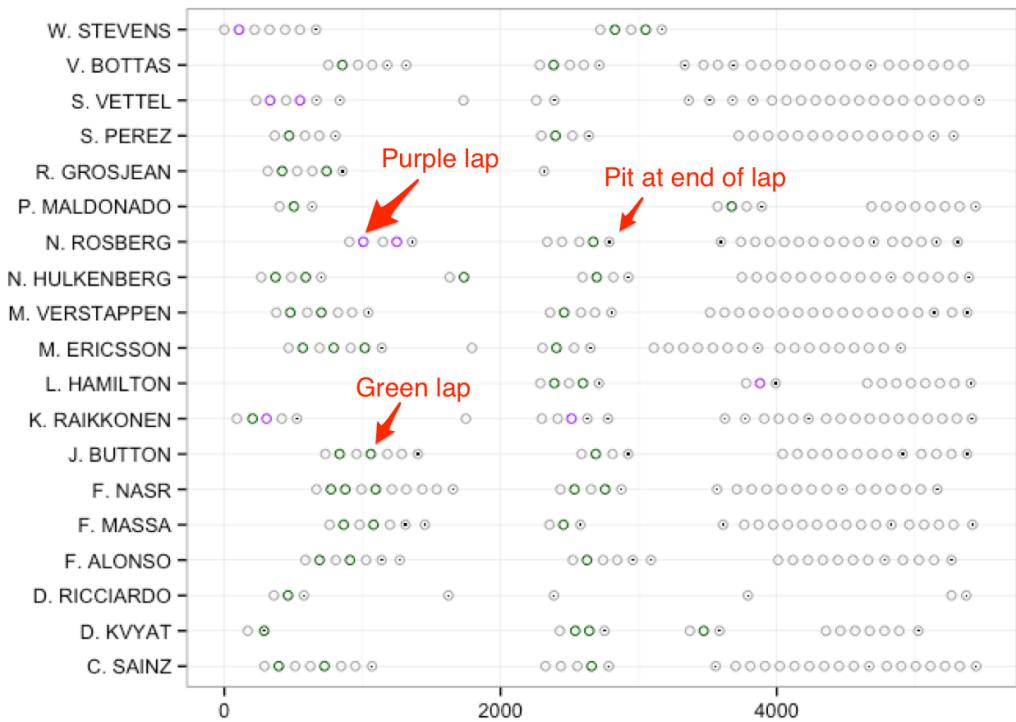
We can now replot the session utilisation chart to show green and purple laptimes, as well as identifying those laps on which the car also came into the pits.

```
g = ggplot(ptimes)
#Layer showing green/purple laps
g = g + geom_point(aes(x=cuml, y=name, color=factor(colourx)), pch=1)
g = g + scale_colour_manual(values=c('darkgrey', 'darkgreen', 'purple'))
#Overplot with a layer showing pit laps
g = g + geom_point(data=ptimes[ptimes['pit']==TRUE, ],
                    aes(x=cuml, y=name), pch='.')
g = g + xlab(NULL) + ylab(NULL) + guides(colour=FALSE) + theme_bw()
g
```



Adding colour to the session utilisation chart: green and purple laps

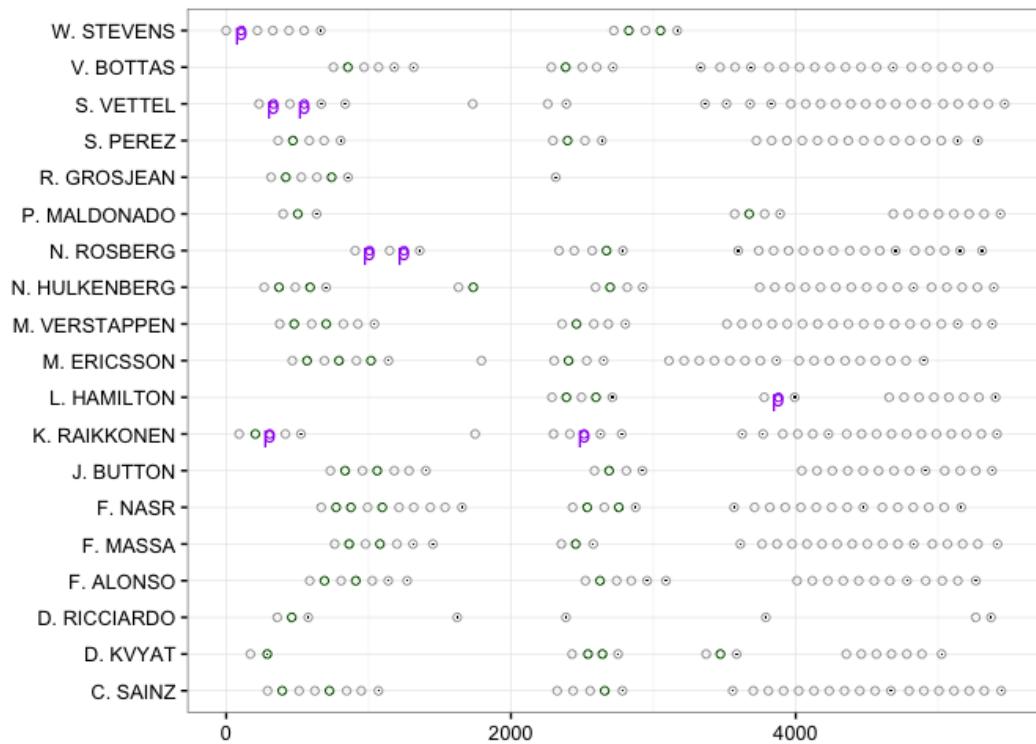
To clarify the chart, three key features of it are the colour coded display of green and purple laps, and identification of the pit stop laps:



Some key features of a session utilisation chart

For black and white (that is, monochrome greyscale) plots, we might choose to use different symbols to represent the green and purple times recorded across the session, or overplot the circles with small labels representing those distinguished times.

```
g + geom_text(data=ptimes[ptimes['colourx']=='purple',],
               aes(x=cuml, y=name),
               label='p',size=4, color=factor('purple'))
```



Annotating the chart for use as a monochrome greyscale figure.

Stint Detection

During a practice session, cars are likely to engage in several rounds of running, or *stints*, interrupted by pit events so that work can be carried out on the car.

By defining a stint as a contiguous sets of laps uninterrupted by a pit event and finishing with a pit event, we can use the pit flags as a marker that allows us to generate a set of stint numbers for each driver and then label each lap with the corresponding stint number.

One method for generating stint number is to sort each driver's laps by *decreasing* lap number (that is, *reverse* the order of each driver's laps), then subtract an accumulated count of pit events for that driver across laps from that driver's total number of pit events. (When summing over TRUE and FALSE values, TRUE counts as 1 and FALSE counts as 0.) Adding one to this value then gives a stint count indexed on an initial stint number of 1.

```
#Reverse order by lapnumber for each driver
ptimes=arrange(ptimes, name, -lapNumber)
#Calculate stint number based on a pitflag counts
ptimes=ddply(ptimes,
            .(name),
            transform,
            stint=1+sum(pit)-cumsum(pit))
```

Ordering the rows by driver and increasing lap number (the normal sort order for each driver), we can then number the laps in each stint:

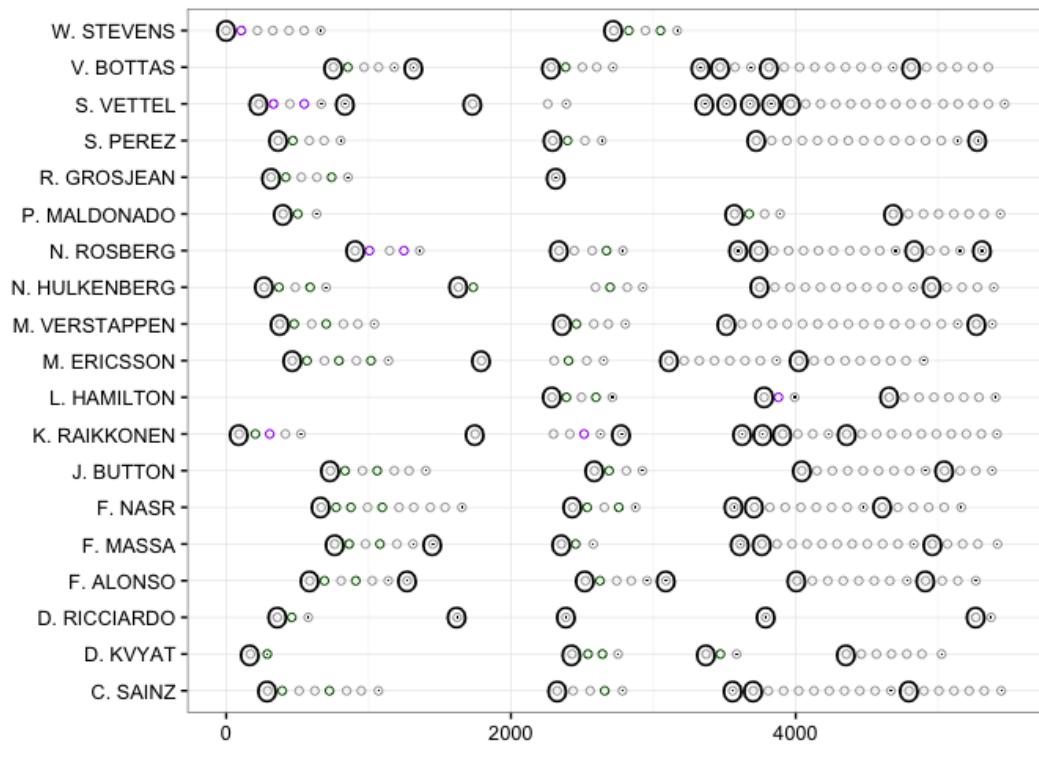
```
ptimes=arrange(ptimes, name, lapNumber)
#Group the rows by driver and stint and number the lap in each stint accordingly
ptimes=ddply(ptimes,
            .(name, stint),
            transform,
            lapInStint=1:length(stint))
```

We can then use the pit flag identifier and stint lapcount to identify outlaps (that is, laps that a driver started from the pit):

```
ptimes=arrange(ptimes, name, lapNumber)
ptimes= ddply(ptimes,
              .(name),
              transform,
              outlap=c(FALSE, head(pit,-1)))
#We also need to capture as an outlap the first lap of the session
ptimes['outlap']= ptimes['outlap'] | ptimes['lapInStint']==1
```

A graphical view allows us to quickly check the identification of outlaps.

```
g + geom_text(data=ptimes[ptimes['outlap']==TRUE,],
               aes(x=cuml, y=name),
               label='0', size=5)
```



Annotating the chart to display outlaps.

Note that an *inlap* is a lap on which the pit flag is set to TRUE.

We can find the number of laps in each stint in a variety of ways. For example, we can group on driver and stint and report the maximum `lapInStint` value for each driver stint:

```
head( ddply( ptimes,
             .(name, stint),
             summarise,
             lapsInStint=max(lapInStint) ) )
```

```
##      name stint lapsInStint
## 1 C. SAINZ     1        8
## 2 C. SAINZ     2        5
## 3 C. SAINZ     3        1
## 4 C. SAINZ     4       10
## 5 C. SAINZ     5        7
## 6 D. KVYAT     1        2
```

Alternatively, we might group on name and stint and count the number of rows (laps) in each group:

```
head( ddply(ptimes,
            .(name,stint),
            summarise,
            lapsInStint=length(stint)) )
```

```
##      name stint lapsInStint
## 1 C. SAINZ     1        8
## 2 C. SAINZ     2        5
## 3 C. SAINZ     3        1
## 4 C. SAINZ     4       10
## 5 C. SAINZ     5        7
## 6 D. KVYAT     1        2
```

We could even co-opt the code developed in the chapter on *Streakiness* used to identify streaks by finding “streaks” in stint number (though I am not sure *why* we would want to adopt this more complex approach!)

```
source('streakiness.R')
sn=ptimes[ptimes$name=='C. SAINZ',]
streaks(sn$stint)
```

```
##   start end 1
## 1     1   8 8
## 2     9  13 5
## 3    14  14 1
## 4    15  24 10
## 5    25  31 7
```

If we look at the stints detected for Ericsson, which as we saw previously looked as if it contained an anomalous singleton lap on his eighth lap, we see that the lack of the pit flag means that single lap stint was not recognised as such.

```
sn=ptimes[ptimes$code=='ERI',]
streaks(sn$stint)
```

```
##   start end 1
## 1     1   7 7
## 2     8  12 5
## 3    13  20 8
## 4    21  29 9
```

Perhaps we need to further elaborate on the stint detection decision by identify a stint as ending *either* when it finishes with a pit event, *or* it is followed by a lap that exceeds a particular duration that does not end in a pit event? (That is, if we can “see” a gap separating stints in the session utilisation chart, perhaps we should also recognise a significant gap in terms of session time between two consecutive laps as indicative of a separation of the laps into two stints?) One obvious question that arises when taking this approach is: *what period of time indicates a significant gap?* In the following example, I use the heuristic of *twice the purple lap time*.

```
ptimes['outlap']= ptimes['outlap'] | ptimes['lapInStint']==1 |
  ( ptimes['stime'] > 2.0 * min(ptimes['purple']) ) & (!ptimes['pit'] ) )
```

We can now *recalculate* the stints based on the change we made to the outlap definition (that is, that an outlap is also classed as a lap more than twice the duration of the session best laptime; note that if a car completes a lap particularly slowly and doesn’t pit, it will be incorrectly classified as an outlap):

```

ptimes=arrange(ptimes, name, lapNumber)
ptimes=ddply(ptimes,
             .(name),
             transform,
             stint=cumsum(outlap))
ptimes=ddply(ptimes,
             .(name, stint),
             transform,
             lapInStint=1:length(stint))
sn=ptimes[ptimes$code=='ERI',]
streaks(sn$stint)

##   start end 1
## 1     1    7 7
## 2     8    8 1
## 3     9   12 4
## 4    13   20 8
## 5    21   29 9

```

Contextualising the Streak Detection Code - Stint Summary Reports

The streak detection code identifies streaks, but otherwise returns data that is free of useful context, such as driver identifier. We can recover some of the context by denormalising the data further by combining it with a driver identifier and stint number:

```

stints=data.frame()
for (name in levels(ptimes$name)){
  dft=ptimes[ptimes$name==name,]
  dft=streaks(dft$stint)
  dft['name']=name
  dft=dft[c('name', 'start', 'end', '1')]
  stints=rbind(stints,dft)
}
#Number the stints for each driver
stints=ddply(stints,.(name),transform,stintNumber=1:length(1))
head( stints )

```

```
##      name start end  l stintNumber
## 1 C. SAINZ     1   8  8       1
## 2 C. SAINZ     9  13  5       2
## 3 C. SAINZ    14  14  1       3
## 4 C. SAINZ    15  24 10       4
## 5 C. SAINZ    25  31  7       5
## 6 D. KVYAT     1   2  2       1
```

We can now generate simple reports that describe just the number of stints completed by each driver:

```
#number of stints
stintcount=ddply(stints,.name),nrow)
stintcount=rename(stintcount, c("V1" = "Stint Count"))

##      name Stint Count
## 1 C. SAINZ      5
## 2 D. KVYAT      4
## 3 D. RICCIARDO  5
## 4 F. ALONSO     6
## 5 F. MASSA      6
## 6 F. NASR       5
```

What these reports don't reveal is those "extended stints" in which a driver was perhaps completing a race simulation that included a practice pit stop. Should these be classed as one stint? Or should we define another "stint group" column that attempts to combine separate stints that appear to be connected by practice pit stop into a corresponding stint group?

Generating Text from a Stint Summary Report

As well as producing tabular data reports, we can generate natural language text reports that summarise various aspects of the data.

For example, suppose we set a variable to the name of a particular driver:

```
name='C. SAINZ'
```

We can then use inline R code within an Rmd file as the basis for a textualisation of several data elements. For example, the sentence:

```
r name completed `r sum(abs(stints[stints['name']==name,][l']))` laps over r nrow(stints[stints['name']==name,]) stints, with a longest run of `r max(abs(stints[stints['name']==name,][l]))` laps.
```

is rendered as: *C. SAINZ completed 31 laps over 5 stints, with a longest run of 10 laps.*

We can also loop through the rows in a dataframe to generate a natural language report from each row. For example:

```
stints['name']=factor(stints$name)
for (name in levels(stints$name)){
  text=**`r name` completed `r sum(abs(stints[stints['name']==name,]['1']))` laps ove\
r `r nrow(stints[stints['name']==name,])` stints, with a longest run of `r max(abs(st\
ints[stints['name']==name,]['1']))` laps.*"
  cat(paste0(knit_child(text=text,quiet=TRUE),'\n'))
}
```

C. SAINZ completed 31 laps over 5 stints, with a longest run of 10 laps.

D. KVYAT completed 16 laps over 4 stints, with a longest run of 7 laps.

D. RICCIARDO completed 8 laps over 5 stints, with a longest run of 3 laps.

F. ALONSO completed 25 laps over 6 stints, with a longest run of 8 laps.

F. MASSA completed 27 laps over 6 stints, with a longest run of 11 laps.

F. NASR completed 30 laps over 5 stints, with a longest run of 10 laps.

J. BUTTON completed 24 laps over 4 stints, with a longest run of 9 laps.

K. RAIKKONEN completed 28 laps over 8 stints, with a longest run of 11 laps.

L. HAMILTON completed 16 laps over 3 stints, with a longest run of 8 laps.

M. ERICSSON completed 29 laps over 5 stints, with a longest run of 9 laps.

M. VERSTAPPEN completed 30 laps over 4 stints, with a longest run of 16 laps.

N. HULKENBERG completed 27 laps over 5 stints, with a longest run of 11 laps.

N. ROSBERG completed 26 laps over 6 stints, with a longest run of 10 laps.

P. MALDONADO completed 15 laps over 3 stints, with a longest run of 8 laps.

R. GROSJEAN completed 7 laps over 2 stints, with a longest run of 6 laps.

S. PEREZ completed 24 laps over 4 stints, with a longest run of 14 laps.

S. VETTEL completed 28 laps over 9 stints, with a longest run of 15 laps.

V. BOTTAS completed 30 laps over 7 stints, with a longest run of 9 laps.

W. STEVENS completed 12 laps over 2 stints, with a longest run of 7 laps.

As well as using a string based template, we could also import a more elaborate template directly from a file, by dropping the `text` parameter to `knit_child()` and simply passing in the name of an Rmd file containing the required sentence, paragraph or document template.

Natural Language Generation (NLG)

Natural Language Generation (NLG) refers the generation of human readable, natural language texts by automated means. One of the easiest ways to get started with NLG is to use templates to create texts directly from data sources, replacing “blanks” in a text with meaningful values extracted from a data set. A slightly more elaborate form of templating involved the use of conditional *if-then* rules to qualify values (for example, transforming a quantity -7 to the phrase *a decrease of 7*). NLG techniques have already been used for several years in the production of automated sports reports and are now becoming mainstream, for example, in baseball reporting^a, creating hype in some communities about the rise of “robot journalists”.

^a<http://www.poynter.org/news/mediawire/344335/resistance-is-futile-ap-to-use-computers-to-cover-baseball-games/>

Finding LapTimes Associated With Long-Run Stints

One of the ways in which teams frequently try to make use of second practice is to complete one or more *long runs* in which a driver stays out for 10 laps or more, collecting data about tyre degradation and presumably helping teams to tune energy management over the course of a lap.

It can be useful to plot the laptimes for just long runs so we can start to see how well the cars are performing, comparing not just one driver with another but also seeing how a particular car behaves over time.

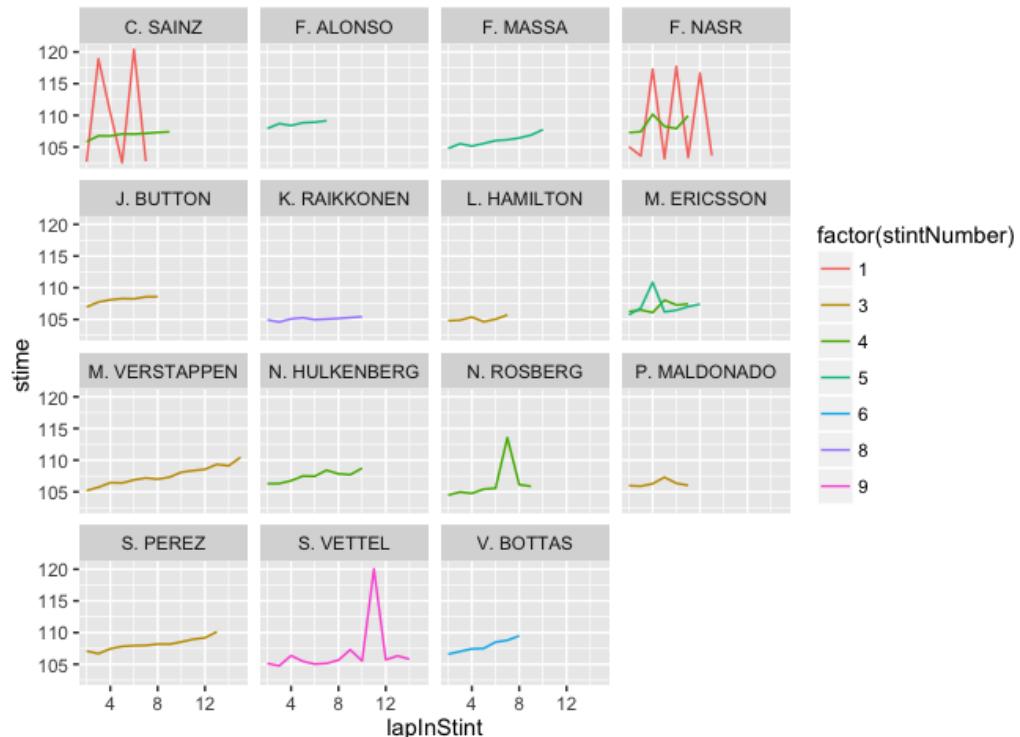
To chart long runs, we need to do a couple of things:

- identify which stints are long run stints (that is, which stints exceed a certain number of laps);
- pull out the lap times for each driver that correspond to their long run laptimes.

```
longruns=merge(stints[abs(stints['l'])>=8,],
               ptimes,by.x=c('name','stintNumber'),
               by.y=c('name','stint'))
longruns=arrange(longruns,lapNumber)
```

We can then plot the run of laptimes associated with each of the long run stints completed by each driver, omitting the outlap and inlap times (and perhaps any other unrepresentative laptimes during the stint) to get a better estimate of how laptimes evolved over the course of each stint.

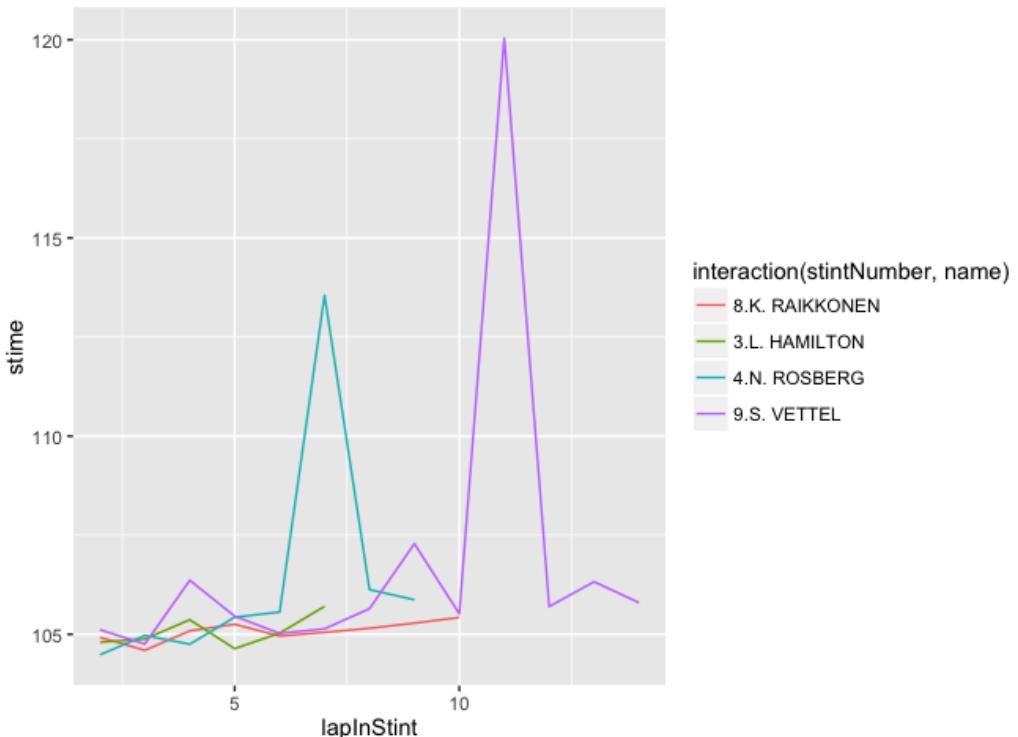
```
#Remove the unrepresentative outlap times
g= ggplot(longruns[!longruns['outlap'] & !longruns['pit'],])
g=g+geom_line(aes(x=lapInStint, y=stime, group=stintNumber,
                   colour=factor(stintNumber)))
g+facet_wrap(~name)
```



Faceted line charts showing long run laptimes for each driver

To compare drivers more directly, we might use a single chart:

```
drivers=c('L. HAMILTON', 'N. ROSBERG', 'K. RAIKKONEN', 'S. VETTEL' )
g= ggplot(longruns[!longruns['outlap'] & !longruns['pit'] & longruns$name %in% driver\$ ,])
g+geom_line(aes(x=lapInStint, y=stime, group=interaction(stintNumber,name),
colour=interaction(stintNumber,name)))
```

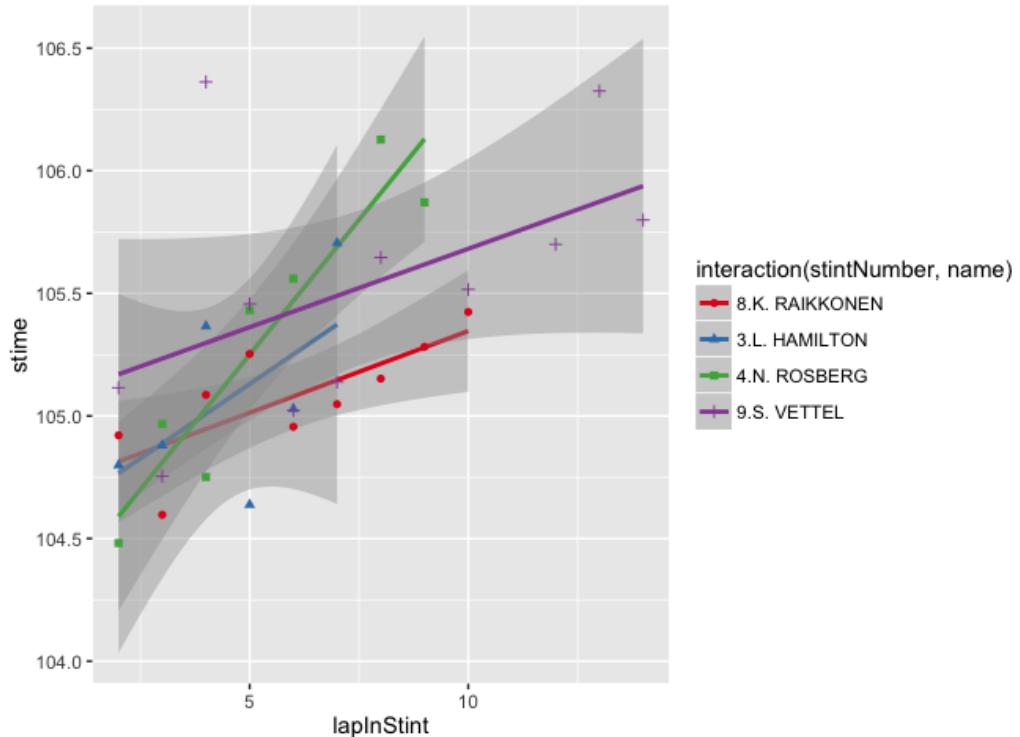


Faceted line charts showing long run laptimes for each driver

With long run stints identified, we can then start to produce a range of simple graphical models for the different drivers. The `geom_smooth()` function generates best fit lines (with associated error limits) according to a specified modeling function.

For example, using a simple linear model, we can get a simple relationship for laptime degradation across a stint.

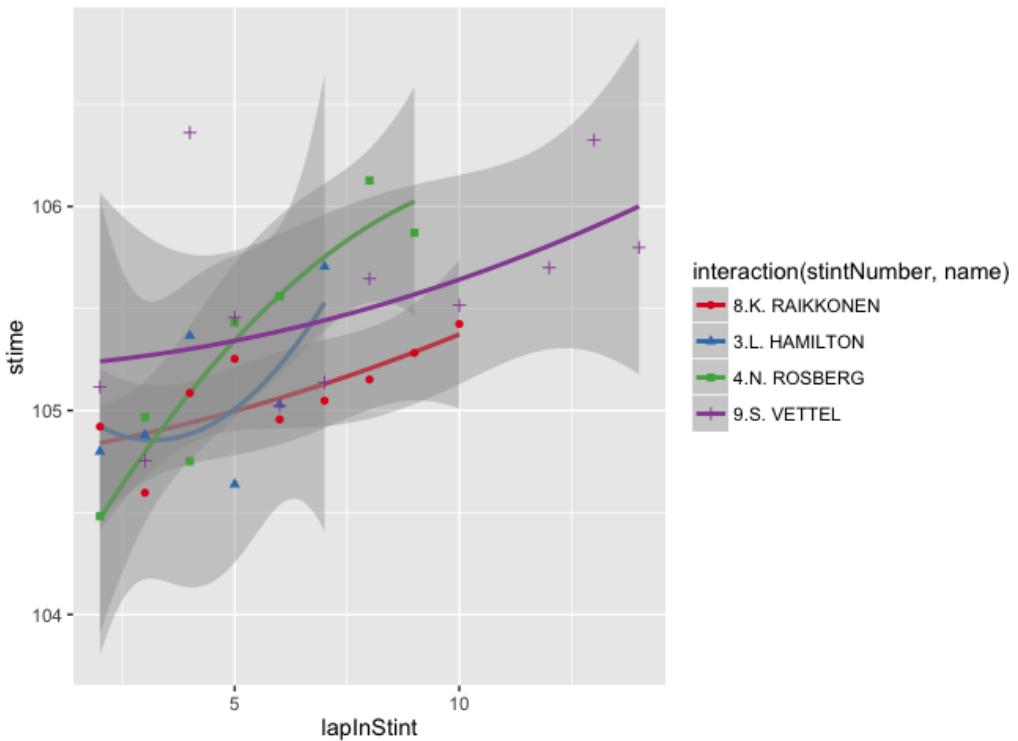
```
drivers=c('L. HAMILTON', 'N. ROSBERG', 'K. RAIKKONEN', 'S. VETTEL' )
g= ggplot(longruns[!longruns['outlap'] & !longruns['pit'] & longruns$name %in% driver\
s & longruns['stime']<1.07*min(longruns['purple'])],,
         aes(x=lapInStint, y=stime, colour=interaction(stintNumber,name)))
g+geom_smooth(method = "lm",
               aes( group=interaction(stintNumber,name)))+ geom_point(aes(shape=inter\action(stintNumber,name)))+ scale_colour_brewer(palette="Set1")
```



Linear model fit for long run laptimes for each driver

Alternatively, we might want to explore higher order models, such as a quadratic model.

```
g= ggplot(longruns[!longruns['outlap'] & !longruns['pit'] & longruns$name %in% driver\ 
s & longruns['stime']<1.07*min(longruns['purple'])],)
  aes(x=lapInStint, y=stime, colour=interaction(stintNumber, name)))
g+geom_smooth(method = "lm", formula = y ~ poly(x, 2),
aes( group=interaction(stintNumber, name)))+ geom_point(aes(shape=inter\ 
action(stintNumber, name)))+ scale_colour_brewer(palette="Set1")
```



Quadratic model fit for long run laptimes for each driver

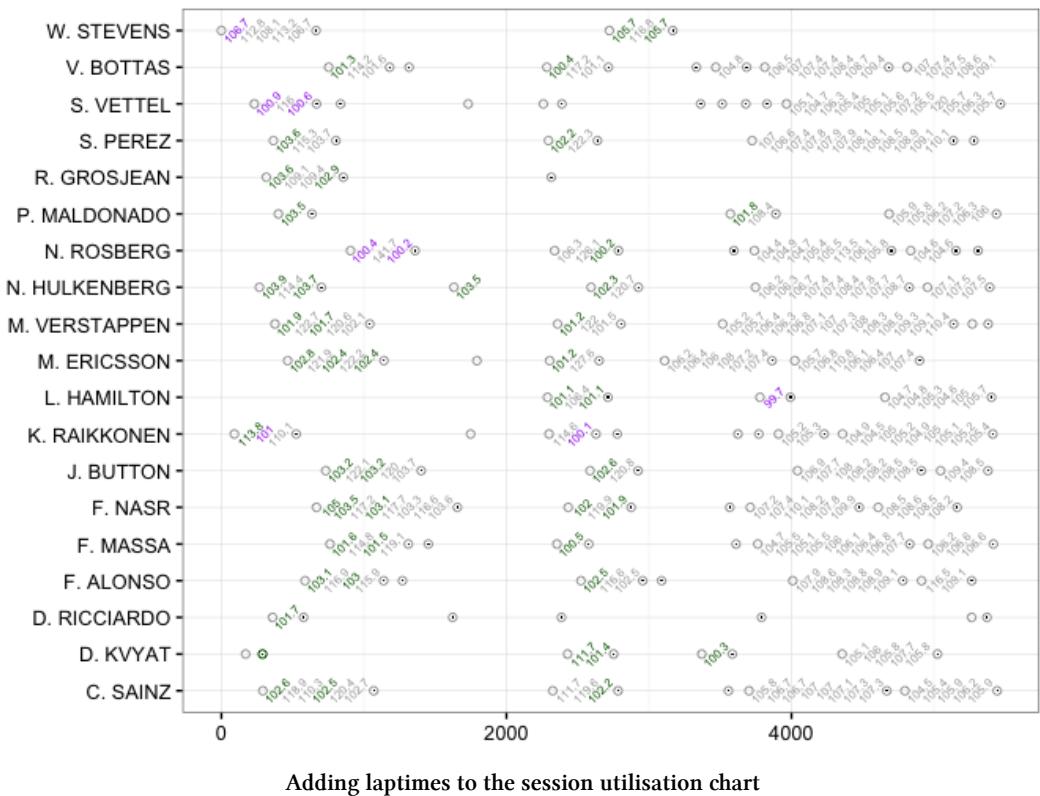
By removing unrepresentative laptimes by detecting and excluding outliers from long stint laptime sets, perhaps even replacing them with representative, interpolated times, we might be able to generate improved models based around these more representative laptimes.

Revisiting the Session Utilisation Chart - Annotations

To add a further layer of information to the session utilisation chart, we might use text labels to show the laptimes, generously rounded down (as is the convention) to the nearest tenth of a second, for laps that are not out-laps or in-laps. (The rounding also reduces clutter on the chart.) The R `round()` function can be used to round to the nearest tenth, but to round *down* we use the `floor` function, which rounds down to the nearest integer, in combination with a *10 multiplier (to bring tenths of seconds into scope as part of an integer value) followed by a /10 divisor (to get back to time in seconds and tenths).

```
g = ggplot(ptimes)
#Layer showing in-laps (laps on which a driver pitted) and out-laps
g = g + geom_point(data=ptimes[ptimes['outlap'] | ptimes['pit'],],
                    aes(x=cuml, y=name, color=factor(colourx)), pch=1)
#Further annotation to explicitly identify pit laps (in-laps)
g = g + geom_point(data=ptimes[ptimes['pit']==TRUE,],
                    aes(x=cuml, y=name), pch='.')
#Layer showing full laps with rounded laptimes and green/purple lap highlights
g = g + geom_text(data=ptimes[!ptimes['outlap'] & !ptimes['pit'],],
                   aes(x=cuml, y=name,
                       label=floor(stime*10)/10,
                       color=factor(colourx)),
                   size=2, angle=45)
g = g + scale_colour_manual(values=c('darkgrey','darkgreen','purple'))

g = g + xlab(NULL) + ylab(NULL) + guides(colour=FALSE) + theme_bw()
g
```



In this case, we use text, colour and symbols to communicate a variety of information about the context of each recorded lap or laptome.

The chart could be further enhanced by using different font styles (such as italics, or bold font) to add even more emphasis to green or purple times.

The open circle markers identify outlaps; the circle markers with a spot in the middle represent inlaps, at the end of which the car pitted.

Whilst this chart is quite dense, and does not fare so well when rendered in black and white (greyscale), it does allow us to see stints and green and purple laps, as well as letting us review laptimes within a stint. On long runs where a car pits and then quickly returns to the track, it might be the case that a driver has done a practice pitstop. In such a case, it might be interesting to try to retrieve the time spent in the pit lane, as well as the times recorded for the in- and out- laps.

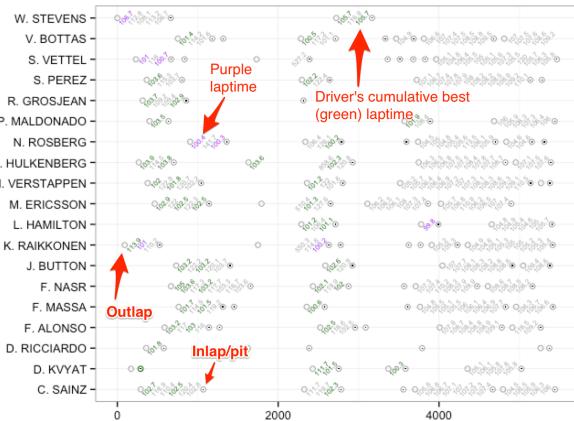
The chart is also missing information about the session classification and the gaps to session

best time and the position ahead. This information is published via the timing sheets, but we can also generate it from the laptime data directly.

Session Summary Annotations

Whilst the scatterplot area of the session utilisation chart is now perhaps at the limits of readability in terms of the amount of detail that is starting to appear *within* the chart, there remains an opportunity for us to annotate the margins of the chart with derived session summary information or information taken from the session classification report.

For example, one of the pieces of information we can derive is the gap between the best laptime of each driver and the driver with the fastest lap in the session or the gap between consecutively ordered driver best laptimes.



Decoding the annotated session utilisation chart

```
#Find the gap to the session best time and the gap between consecutive positions
ptimes=arrange(ptimes,name,lapNumber)
spurple=min(ptimes['purple'])
ptimesClass=ddply(ptimes[ptimes['driverbest']<9999,], .(name), summarise,
                  driverbest=min(driverbest), gap=min(driverbest)-spurple)
ptimesClass=arrange(ptimesClass,driverbest)
ptimesClass['diff']=c(0,diff(ptimesClass$gap))
ptimesClass$pos=1:nrow(ptimesClass)
ptimesClass
```

```

##           name driver best   gap diff pos
## 1     L. HAMILTON    99.790 0.000 0.000  1
## 2     K. RAIKKONEN   100.163 0.373 0.373  2
## 3     N. ROSBERG   100.218 0.428 0.055  3
## 4     D. KVYAT      100.346 0.556 0.128  4
## 5     V. BOTTAS     100.450 0.660 0.104  5
## 6     F. MASSA      100.560 0.770 0.110  6
## 7     S. VETTEL     100.652 0.862 0.092  7
## 8 M. VERSTAPPEN   101.220 1.430 0.568  8
## 9 M. ERICSSON    101.261 1.471 0.041  9
## 10 D. RICCIARDO  101.799 2.009 0.538 10
## 11 P. MALDONADO  101.877 2.087 0.078 11
## 12 F. NASR        101.988 2.198 0.111 12
## 13 S. PEREZ       102.242 2.452 0.254 13
## 14 C. SAINZ       102.291 2.501 0.049 14
## 15 N. HULKENBERG 102.330 2.540 0.039 15
## 16 F. ALONSO      102.506 2.716 0.176 16
## 17 J. BUTTON       102.637 2.847 0.131 17
## 18 R. GROSJEAN    102.948 3.158 0.311 18
## 19 W. STEVENS     105.704 5.914 2.756 19

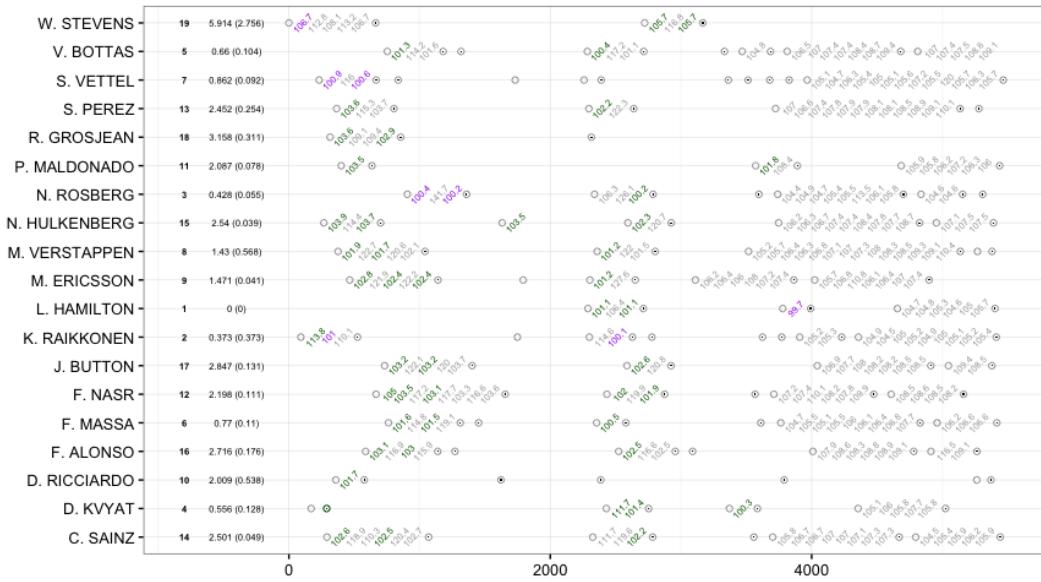
```

This information can then be used to annotate the session utilisation chart to provide an added, session summary, dimension to the margin of the chart.

```

g=g+geom_text(data=ptimesClass,
               aes(x=-800,y=name,label=pos),size=2,fontface='bold')
g=g+geom_text(data=ptimesClass,
               aes(x=-400,y=name,
                   label=paste(round(gap,3)," (",round(diff,3),")",sep=' ')),
               size=2)
g

```

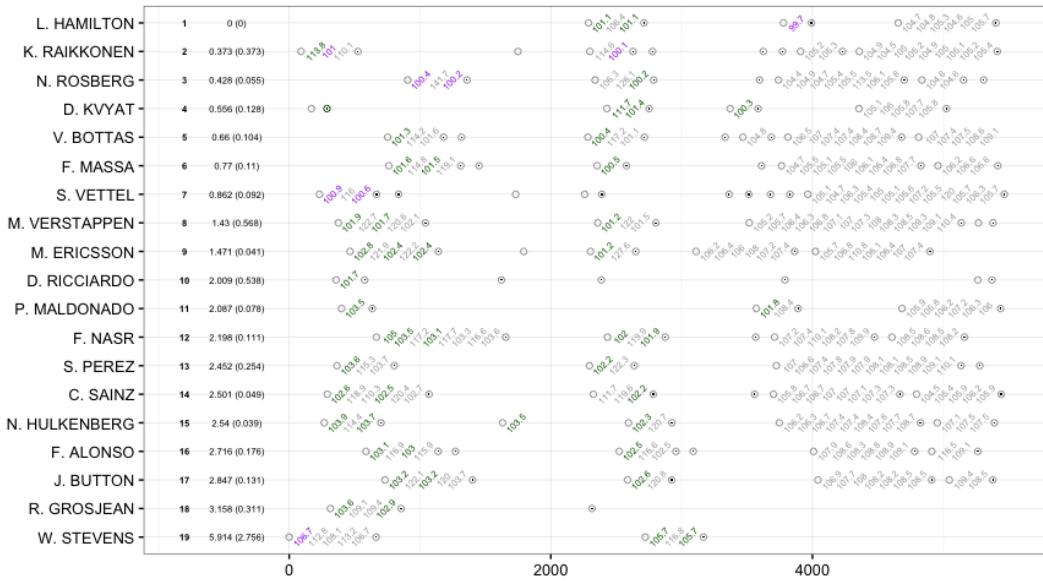


Adding session summary information to the margin of the session utilisation chart

At the moment, the chart is alphabetically ordered on the y-axis, although this is perhaps not the best ordering, as the marginal information highlights. Alternatives include: by number of laps, by number of stints, by fastest lap, by time of day of first lap, by time of day of last lap.

We can manipulate the y-axis order of the chart by refactoring the levels of the driver names according the position rank.

```
#Order the chart by driver session position
levels(ptimes$name) = factor(ptimesClass$name,
                             levels = levels(ptimesClass$name[order(ptimesClass$pos)]))
g+scale_y_discrete(limits=rev(levels(ptimes$name)))
```



Ordering drivers by laptime classification on the session utilisation chart

Further annotations to the chart might then include a count of the laps completed by each driver, and a statement of their fastest time in the session. Or vertical lines (either solid, or dashed) could be overplotted onto the chart might to mark out times when purple times are achieved. It would also be useful if the chart carried information about times when the circuit was under yellow or red flag conditions, but this information cannot reliably be derived from the timing sheets.

Session Utilisation Lap Delta Charts

One of the messages commentators look for in the practice session laptime data is some sort of indication of the rate of drop off in laptime throughout a stint. This sort of information can be obtained from statistical models, as the simple graphical long stint analysis hinted at. But commentators and race engineers are also well versed in extracting this sort of difference information from timing screens. So can we try to capture some of that sort of information on the session utilisation chart?

The basic annotated session utilisation chart we have considered so far just shows the raw (rounded) laptime, which means the reader has to mentally calculate the differences between consecutive laptimes. But perhaps we can make use of alternative “toggle” views of the

chart that use a similar layout but that show complementary information about each point compared to the original?

One possible “toggle view” of the session utilisation chart is a chart in which we show the first full laptime of a stint as such, and then for each consecutive lap display the time difference to the previous lap, rather than just the laptime associated with each lap.

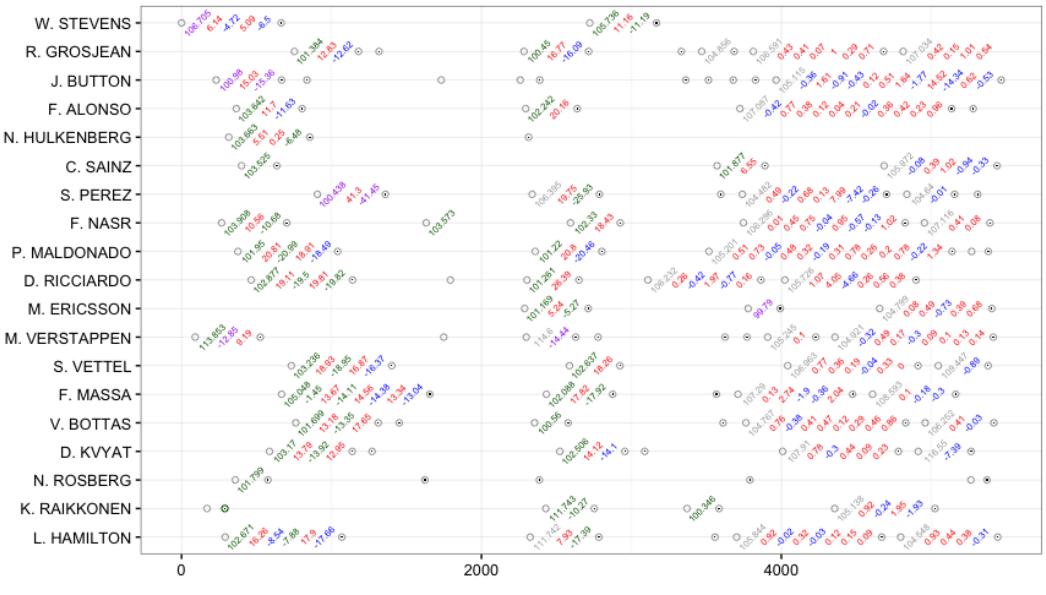
Let’s start by calculating the difference for each driver in turn between consecutive laps on each of their stints, adding a colour channel to identify additional information about the time (e.g. was it a purple time, an decrease on the previous laptime, and so on).

```
ptimes=ddply(ptimes,.(name,stint),transform,diff=c(0,diff(stime)))
ptimes['coloury']=ifelse(ptimes$colourx=='black',
                         ifelse(ptimes$diff>=0.0,'red','yellow'),
                         ptimes$colourx)
```

Charting this data allows us to see how each driver’s lap compared in terms of laptime difference with their previous lap.

```
g = ggplot(ptimes)
#Layer showing in-laps (laps on which a driver pitted) and out-laps
g = g + geom_point(data=ptimes[ptimes['outlap'] | ptimes['pit'],],
                    aes(x=cuml, y=name, color=factor(colourx)), pch=1)
#Further annotation to explicitly identify pit laps (in-laps)
g = g + geom_point(data=ptimes[ptimes['pit']==TRUE,],
                    aes(x=cuml, y=name),pch='.')
#Layer showing start of stint laptimes and green/purple lap highlights
g=g+geom_text(data=ptimes[ptimes['lapInStint']==2 & !ptimes['pit'],],
              aes(x=cuml, y=name,
                  label=stime,#floor(stime*10)/10,
                  color=factor(colourx)),
              size=2, angle=45)
#Layer showing stint laptime deltas and green/purple lap highlights
g = g + geom_text(data=ptimes[ptimes['lapInStint']>2 & !ptimes['pit'],],
                  aes(x=cuml, y=name,
                      label=round(diff,2),
                      color=factor(coloury)),
                  size=2, angle=45)
g = g + scale_colour_manual(values=c('darkgrey','darkgreen','purple','red','blue'))

g + xlab(NULL) + ylab(NULL) + guides(colour=FALSE) + theme_bw()
```



Laptimes delta toggle view of the session utilisation chart

We use colour as an additional channel to show whether or not the difference was a positive or negative one in terms of improved laptimes, although in the above chart the difference is also identified by the presence, or otherwise, of the minus sign: a negative number indicates that that lap was *faster* than the previous laptimes (i.e. the laptimes were *less* than the previous laptimes). It could also be argued that we should negate the differences for this sort of signed displayed so that an negative number indicates that the laptimes was that much time *worse* than the previous lap.

Summary

In this chapter, we have started to explore the idea of *session utilisation charts* that use practice session laptimes to show how each particular driver made use of a particular practice session. These charts can be used to provide an “at a glance” summary of track utilisation for a whole practice session across all drivers involved.

The laptimes data can itself be processed to identify separate stints, consecutive laps separated by a pit stop, as well as outlaps and inlaps at the start and end of each stint accordingly.

The charts can be enriched by using a blend of symbols to show inlaps and outlaps as such, and text labels to show the times of fully completed laps. Colour is used to further highlight

each driver's cumulative best laptime across a session, as well as purple laps as they are recorded through a session.

The chart margins can also be used to display additional *session summary* data, such as the gap to the best laptime recorded in the session, or the gap to the to car classified one position ahead.

This chapter also introduced what I have termed *toggle views* of a chart. These are views in which the spatial layout is preserved but alternative label displays are used. (The phrase is derived from interactive displays where we might naturally toggle (that is, switch) between views of a chart.) In the case of the session utilisation charts, a toggle chart view was created to show consecutive laptime differences for each driver within each of their stints.

Useful Functions Derived From This Chapter

```
#A function to augment raw laptime from practice and qualifying sessions
## with derived data columns
library(plyr)
rawLap_augment_laptimes = function(df){
  df['code']=apply(df['name'],2,function(x) driverCode(x))

  df=ddply(df,.(name),transform,cuml=cumsum(stime))
  df['pit']=df['pit']=='True'
  df=arrange(df,name, -lapNumber)
  df=ddply(df,.(name),transform,stint=1+sum(pit)-cumsum(pit))
  df=arrange(df,name, lapNumber)
  df=ddply(df,(name,stint),transform,lapInStint=1:length(stint))
  df=arrange(df,name, lapNumber)
  df=ddply(df,.(name),transform,driverbest=cummin(c(9999,stime[2:length(stime)])))
  df=arrange(df,cuml)
  df['purple']=sapply(df['driverbest'],cummin)
  df['colours']=ifelse(df['stime']==df['purple'],
    'purple',
    ifelse(df['stime']==df['driverbest'],
      'green',
      'black'))

  df=arrange(df,name, lapNumber)
  df= ddply(df,.(name),transform,outlap=c(FALSE, head(pit,-1)))
  df['outlap']= df['outlap'] | df['lapInStint']==1 | (df['stime'] > 2.0 * min(df['purple'])) & (!df['pit']) )
  df=ddply(df,
    .(name),
```

```

    transform,
    stint=cumsum(outlap),
    lapInStint=1:length(stint))
df=ddply(df,
         .(name, stint),
         transform,
         lapInStint=1:length(stint))
df
}

plot_session_utilisation_chart = function (df){
  g = ggplot(df)
  #Layer showing in-laps (laps on which a driver pitted) and out-laps
  g = g + geom_point(data=df[df['outlap'] | df['pit'],],
                      aes(x=cuml, y=name, color=factor(colourx)), pch=1)
  #Further annotation to explicitly identify pit laps (in-laps)
  g = g + geom_point(data=df[df['pit']==TRUE,],
                      aes(x=cuml, y=name), pch='.')
  #Layer showing full laps with rounded laptimes and green/purple lap highlights
  g = g + geom_text(data=df[!df['outlap'] & !df['pit'],],
                     aes(x=cuml, y=name,
                         label=floor(stime*10)/10,
                         color=factor(colourx)),
                     size=2, angle=45)
  g = g + scale_colour_manual(values=c('darkgrey', 'darkgreen', 'purple'))
  g + xlab(NULL) + ylab(NULL) + guides(colour=FALSE) + theme_bw()
}

```


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 part for the participating drivers. The number of drivers progressing from one part of qualifying to the next is, in part, determined by the number of race entrants. Prior to 2015, 16 cars progressed from the first round of qualifying (“Q1”) to to the second (“Q2”). With a smaller grid size in 2015, the number of cars making it into Q2 was reduced to 15. The top 10 cars from Q2 then progress to the final round of qualifying, Q3.

The final qualifying session classification determines the grid position for the race for each driver (penalties aside) and as such may be a predictor of the race winner, and is determined by the rank position of each driver from the last round of qualifying they competed in. (So the top 10 from Q3 make up the provisional first 10 places on the grid. The 11th to 15th (or 16th) cars from Q2 take up the corresponding positions in the grid. Add the cars that failed to make it out of Q1 are provisionally placed on the grid according to their Q1 classification.)

When trying to predict target cutoff times for each of the qualifying sessions (that is, the times that separate drivers who progress to the next round of qualifying from those that don’t), 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 practice sessions, we can get results data from the *ergast* database and results, speeds and sector times from the F1 (historical data) and FIA (data as of 2015) websites. 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/FIA data - so let’s see which table we need.

```
library(DBI)
f1 =dbConnect(RSQLite::SQLite(), './f1com_results_archive.sqlite')
## list all tables
dbListTables(f1)
```

```
## [1] "QualiResultsTo2005" "Sectors"           "Speeds"
## [4] "pResults"          "qualiResults"      "raceFastlaps"
## [7] "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, omitting the *year* and *race* columns.

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

q1time	driverNum	pos	q2time	q3natTime	q3time	q2natTime	laps	team	q1natTime	driverName
115.926	1	3	114.499	1:54.960	114.96	1:54.499	23	Red Bull Racing	1:55.926	Sebastian Vettel
119.260	10	18	NA	NA	NA	NA	10	Caterham Renault	1:59.260	Kamui Kobayashi
118.362	11	16	118.264	NA	1:58.264	17	Force India-Mercedes	1:58.362	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 part of qualifying, 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. This allows us to see how the drivers were ranked in each part of qualifying, which drivers progressed from one session to the next, and how their classified lap times evolved individually.

Qualifying Progression Charts

The easiest way to generate what we might term a *qualifying progression chart* is to assemble the data in to a *tidy data* form, such 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 session laptimes
qm=melt(qualiResults,
        id=c('driverNum','driverName'),
        measure=c('q1time','q2time','q3time'),
        variable.name='session',
        value.name='laptimes')

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

driverNum	driverName	session	laptimes
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 laptimes within each session.

```
library(ggplot2)

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

<div class="figure">  <p class="caption">Text plot showing relative qualifying session times</p> </div>
```

This chart shows several things directly, 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 within each part of qualifying.

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 15 or 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, by the presence of overlapping driver name labels.

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

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

```
<div class="figure">  <p class="caption">Text plot of qualifying session times with driver connector lines</p> </div>
```

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 such an approach is the clear separation of labels, the downside a loss of information about separation in terms of laptime. Perhaps we could combine the approaches?

Improving the Qualifying Session Progression 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 distinguish 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.

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

With the rank position available within each session, we can highlight the drivers likely to be of interest in each part of qualifying. Using a plain white background increases the contrast and improves the clarity of the graphic.

```

g = ggplot(qm,aes(x=session,y=lapttime))
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)+theme_bw()

```

<div class="figure"> <p class="caption">Session times table, with highlights</p> </div>

(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 reminiscent of, although far more cluttered than, a very clean chart type known as a *slopegraph* originally proposed by Edward Tufte.

It is possibly worth exploring additional toggle views of this chart using separate time measures, such as the gap to leader or the difference to the car classified one place ahead in the session. For example:

- *diff*: for each session, given rank ordered drivers by session classification, find `c(time_of_first_car, diff(lapttime))`; that is, find the gap to the car ahead;
- *gap*: for each session, for rank ordered drivers by session classification, find `c(time_of_first_car, difference_to_first)`; that is, find the gap to the first placed driver in the session.

Whilst these sorts of chart are *always* likely to be plagued by the label overlap problem for drivers recording similar laptimes, we can do something about the overlap of labels and the connecting lines. The trick to doing this is to generate a *slopegraph* in which we split the lines into line segments drawn between the categorical name columns.

However, moving to a ranked view leads to one obvious downside: if a particular driver's best session laptime increases (that is, *gets worse*) moving from one session to another, whilst other drivers improve their best time, we lose that information, along with any insights it might provide as to how much higher placed the driver may have been in a later session if they had maintained their form of an earlier session.

Qualifying Session Rank Position Summary Chart - Towards the Slopegraph

Whilst slopegraphs strictly defined use a continuous vertical y-axis so that differences in value can be visualised across categorical groups arranged along the horizontal x-axis, a scheme similar to the one used in the session table shown above, we have already seen how the vertical arrangement of items can often lead to labels being overlapped.

An alternative approach is to scale the y-axis, for example according to rank. Using session rank to order each driver by session, we can generate a visual summary of how the drivers were classified across each of the qualifying sessions. That is, we can summarise the qualifying session by putting together a session results summary chart that shows the position of each driver at the end of each qualifying session in the form of a *qualifying sessions classification rank chart*. 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))
```

<div class="figure"> <p class="caption">Custom line chart showing rank by qualifying session</p> </div>

Viewing the Qualifying Session Progression Table as a Slopegraph

The multicolumn view used in the qualifying session progression charts is cluttered in part by the way we overlay text labels on top of a continuous line chart. If we introduce breaks or gaps in the line, we can get a much clearer view of the labels, and more clearly see how the labels in one column are connected to the labels in another.

We can generate the ranked times for each session as follows:

```

qualiResults=arrange(qualiResults,q1time)
qualiResults['q1pos']=rank(qualiResults['q1time'],na.last='keep')
qualiResults=arrange(qualiResults,q2time)
qualiResults['q2pos']=rank(qualiResults['q2time'],na.last='keep')
qualiResults=arrange(qualiResults,q3time)
qualiResults['q3pos']=rank(qualiResults['q3time'],na.last='keep')

```

The ranked slopegraph is then constructed by generating a stacked (ranked) list of drivers for each session, and then connecting each driver's time using a line segment. Colour highlights are used to identify the 'names of interest' in each session. For Q1 and Q2 I deem these to be the drivers that *miss* the cut-off.

To make the function more general, we can optionally pass in the cut-off limits for each session; in order to access the cutoff variable, we need to pass in an appropriate environmental scope to the `ggplot aes()` function. Similarly, to account for different label widths, we can pass in the `spacer` parameter value to adjust the line segment widths.

```

slopeblank=theme(panel.border = element_blank(),
                 panel.grid.major = element_blank(),
                 panel.grid.minor = element_blank(),
                 axis.line = element_blank(),
                 axis.ticks = element_blank(),
                 axis.text = element_blank())

core_qualifying_rank_slopegraph= function(qualiResults,qm,
                                             spacer=0.25,cutoff=c(16,10)){
  #http://stackoverflow.com/questions/10659133/local-variables-within-aes
  .e = environment()
  g=ggplot(qualiResults,aes(x=session,y=laptimes), environment = .e)
  g= g+geom_text(data=qm[qm['session']=='q1time'],,
                  aes(x=1,y=qspos,label=driverName,
                      colour=(qspos>cutoff[1] ))
                  ), size=3)
  g= g+geom_text(data=qm[qm['session']=='q2time'],,
                  aes(x=2,y=qspos,label=driverName,
                      colour=(qspos>cutoff[2] ))
                  ), size=3)
  g= g+geom_text(data=qm[qm['session']=='q3time'],,
                  aes(x=3,y=qspos,label=driverName,
                      colour=TRUE
                  ), size=3)
}

```

```

g=g+geom_segment(data=qualiResults[!is.na(qualiResults['q2time'])],,
                  x=1+spacer,xend=2-spacer,
                  aes(y=q1pos,yend=q2pos,group=driverName),
                  colour='slategrey')
g=g+geom_segment(data=qualiResults[!is.na(qualiResults['q3time'])],,
                  x=2+spacer,xend=3-spacer,
                  aes(y=q2pos,yend=q3pos,group=driverName),
                  colour='slategrey')
g=g+scale_colour_manual(values=c('slategrey','blue'))
g=g+guides(colour=FALSE)
g+theme_bw() +xlab(NULL)+ylab(NULL)+xlim(0.5,3.5)+slopeblank
}

```

Defining the chart generator as a function makes the production of charts a one-line affair.

```
core_qualifying_rank_slopegraph(qualiResults,qm)
```



We can also generate this style of chart directly from data pulled from the *ergast* API, although to do so we need to make a few mappings of column names so that we can reuse the chart generating function directly.

```

source('ergastR-core.R')
#quali.ergast=qualiResults.df(2015,2)

qr=qualiResults.df(2015,1)

#Transform the data so that we can use it in the original function

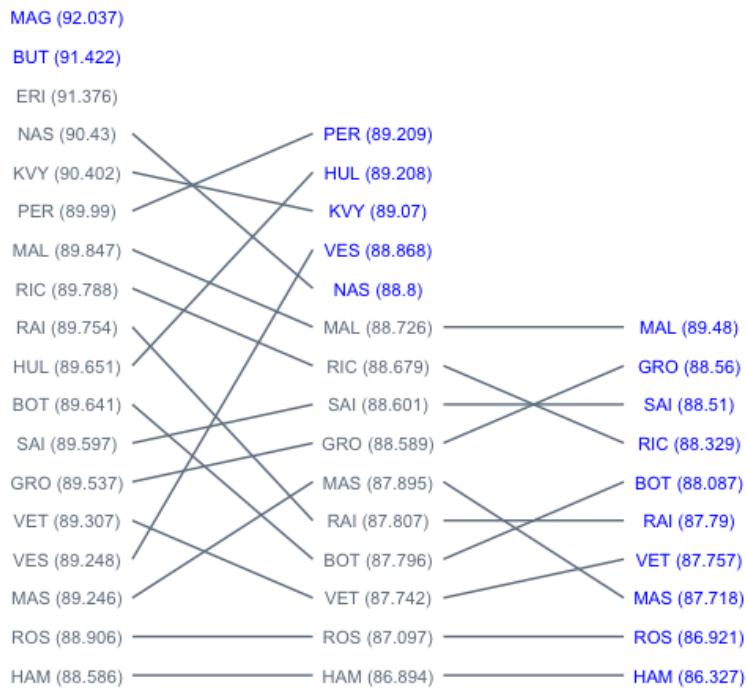
#1. Rename columns so they work with the original charting function
#q1time, q2time, q3time, q1pos, q2pos, q3pos, driverName, qspos
qr=rename(qr, c("Q1_time"="q1time", "Q2_time"="q2time", "Q3_time"="q3time",
               "Q1_rank"="q1pos", "Q2_rank"="q2pos", "Q3_rank"="q3pos",
               "position"="qspos", "code"="driverName"))

#2. Generate qm equivalent by melting elements of qualifying results dataframe
#driverName, qspos, session (q1time, q2time, q3time)
qrm=melt(qr,
          id=c('driverName'),
          measure=c('q1time','q2time','q3time'),
          variable.name='session',
          value.name='laptimes')
qrm$driverName=paste(qrm$driverName, " (",qrm$laptimes,")",sep=' ')
qrm=ddply(qrm, 'session',mutate,qspos=rank(laptimes))

#Make sure that the laptimes values are treated as numeric quantities
qrm$laptimes=as.numeric(qrm$laptimes)
#If a laptimes is recorded as 0 seconds, set it to NA
qrm[qrm == 0] <- NA
#Drop any rows with NA laptimes values
qrm=qrm[with(qrm,!is.na(laptimes)),]

core_qualifying_rank_slopegraph(qr,qrm,spacer=0.21)

```



Rank-Real Plots

One of the problems when using a laptime basis for the vertical y-axis in the qualifying session progression chart is that the labels can often overlap. One way round this might be to use *two* vertical scales to display the information: a base axis ranging 1..N for N drivers, against which we plot each driver name, equally spaced; and a second axis, scaled against the first, that relates to laptime.

The chart can then contain *two* sorts of element - name/identifier columns, containing names/driver codes against an equally spaced categorical (integer) axis, and qualifying session time columns, against a real laptime axis, with the times simply represented as points. Lines can then associate names with time points, and the progression of time points for each driver can also be connected using lines.

Ultimate Laps

As the aim of the qualifying session is to put together the fastest lap, it can be useful to know whether the best laptime achieved 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?

Until 2015, sector times were published in the results area of the official Formula One website. With the redesign of that site at the start of 2015, sector times disappeared from the public areas of the website at least. However, best sector times for each driver are still available (at the time of writing at least) in the form of FIA media release timing sheets.

Summary

In this chapter we have started to look in at some of the data associated with qualifying, focusing in particular on how drivers progressed with the qualifying sessions. Two complementary ways of positioning the ordered driver list for each part of qualifying are possible. The first employs a continuous axis representing the best recorded laptime by each driver within a session; the second utilises a discrete axis representing the driver rank within each part of qualifying.

The continuous axis, laptime based arrangement allows us to see how close drivers were to each other within a session in terms of laptime, as well as allowing us to monitor best laptime evolution across sessions. However, the chart suffered from overlapping labels where drivers' laptimes were close to each other. The discrete, rank based axis ensures that labels are always well spaced and visible, and also allows us to easily recover the overall session classification, but at the expense of losing information relating to laptimes.

A Further Look at Qualifying

As with the practice sessions, we can use qualifying session laptime data to chart how each of the drivers makes use of the qualifying session, noting that not all drivers will be allowed to pass through into the second and third rounds of qualifying.

At first glance, it might seem as if we could simply reuse the functions developed to chart practice session utilisation to reveal how each driver made use of qualifying. (As before, the session laptimes are obtained by scraping timing information sheets published by the FIA.) However, this approach would result in purple and green laptimes be calculated across all three qualifying sessions, rather than being calculated from a clean slate within each qualifying session, as the following chart demonstrates.

For this chapter, we will use qualifying times for the 2015 British Grand Prix scraped from the official FIA timing sheets (data file link⁴⁵).

```
#Data file available at:  
#https://gist.githubusercontent.com/psychemedia/11187809/raw/gbr_2015_qualilaptimes.c\  
sv  
f12015test=read.csv("~/Dropbox/wranglingf1datawithr/src/gbr_2015_qualilaptimes.csv")
```

The data takes the following form:

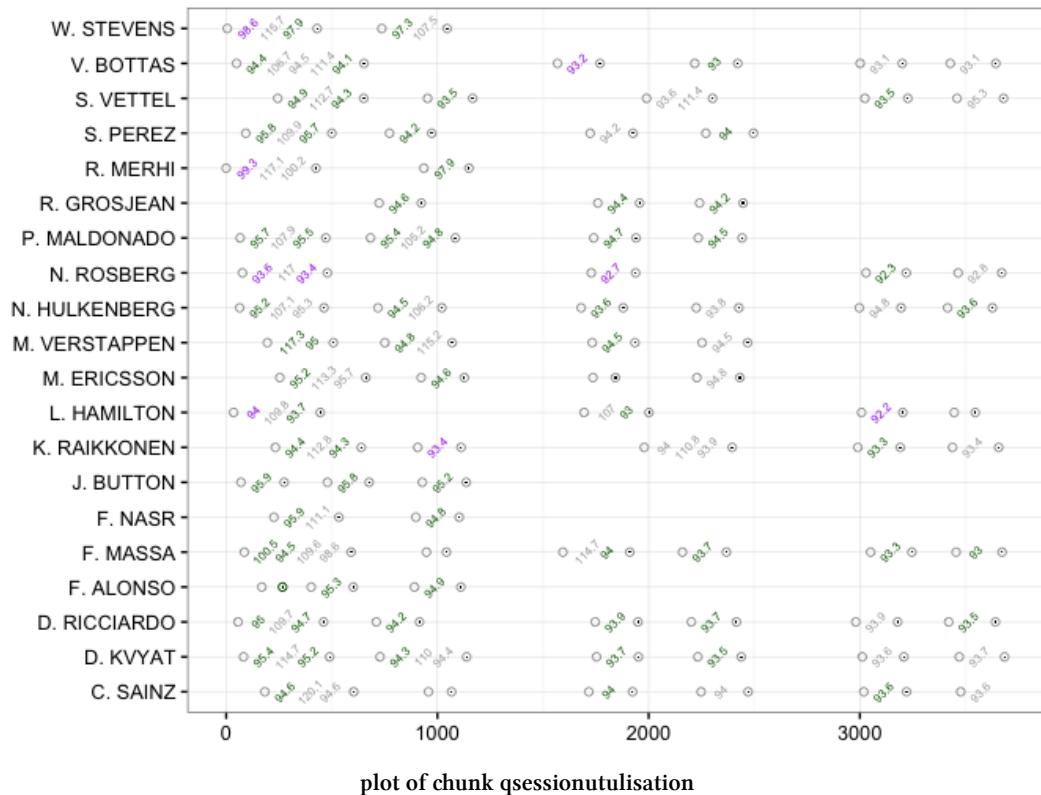
lapNumber	lapttime	name	number	pit	stime
1	00:00:56	D. RICCIARDO	3	False	56.000
2	1:35.047	D. RICCIARDO	3	False	95.047
3	1:49.740	D. RICCIARDO	3	False	109.740

This is the same form as the laptime data used to plot the session utilisation charts, so it's a simple enough matter to generate session utilisation charts that show how the track was used during qualifying.

⁴⁵https://gist.githubusercontent.com/psychemedia/11187809/raw/gbr_2015_qualilaptimes.csv

```
#Load in the laptime annotation and session utilisation plot functions
## described in the Practice Session Utilisation chapter
#These can be found in the file:
#https://gist.githubusercontent.com/psychamedia/11187809/raw/practiceQualiLaps.R
source('practiceQualiLaps.R')

f12015test=rawLap_augment_laptimes(f12015test)
plot_session_utilisation_chart(f12015test)
```



Unlike the practice session, however, the qualifying session is split into three parts. This suggests that rather than generate purple and green times relative to the qualifying session as a whole, we should really calculate those times relative to each part of qualifying. This means we need to find a way to identify the separate phases of qualifying from the laptime data itself, before finding purple and green times within each part of qualifying.

Clustering Qualifying Laptimes by Session

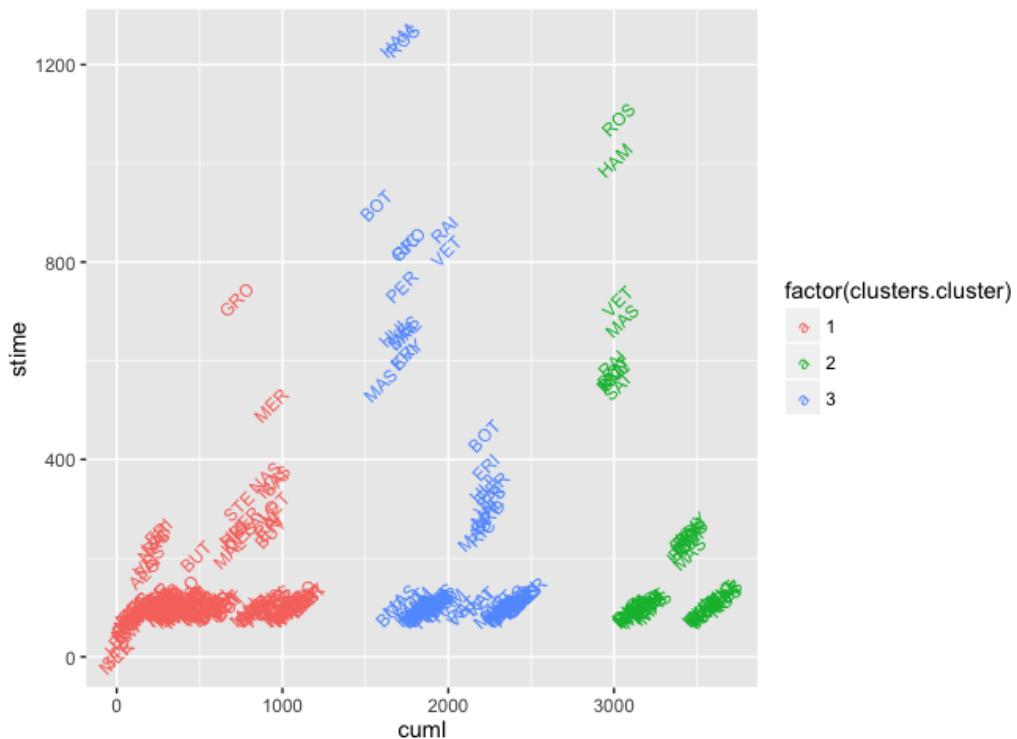
Several well known algorithms exist that are capable of clustering data elements that are in some way alike in an unsupervised way (that is, without reference to some sort of training set in which a known mapping exists from particular data points into particular groups or clusters).

One of the simplest and most widely used techniques is *k-means analysis*. When we run such an analysis over the 1-dimensional laptimes data, specifying the number of means (or groups into which the data should be clustered) as 3, one for each part of qualifying, this is what we get as a result:

```
# Attempt to identify qualifying session using K-Means Cluster Analysis
#Specify 3 means (3 sessions)
clusters = kmeans(f12015test['cum1'], 3)

# get cluster means - summary report
#aggregate(f12015test,by=list(clusters$cluster),FUN=mean)

f12015test = data.frame(f12015test, clusters$cluster)
ggplot(f12015test)+geom_text(aes(x=cum1,y=stime,label=code,
                                colour=factor(clusters.cluster)),
                            angle=45,size=3)
```



Good attempt - but no cigar... Whilst all the Q3 laptimes have been grouped together, and all of the Q2 laptimes have been grouped together, some of the Q1 laptimes have also been clustered with Q2 laptimes. This approach demonstrates how a naive or simplistic application of a powerful statistical clustering technique may not have the desired result...

Perhaps a less sophisticated, more cunning strategy might work instead?

The F1 regulations⁴⁶ suggest there are at least seven minutes between qualifying sessions. Using this information, if we look for at least a five minute gap, say, between consecutive laptimes, and assume that such a gap is unlikely to occur *within* a session (that is, we assume there will never be more than five minutes between two cars crossing a timing line while a session is running), we can identify stints on that basis.

⁴⁶https://www.formula1.com/content/fom-website/en/championship/inside-f1/rules-reg/Practice_qualifying_and_race_start_procedure.html

```

#Order by accumulated laptimes
f12015test=arrange(f12015test,cuml)
#Find the gap between consecutively recorded accumulated laptimes
f12015test['gap']=c(0,diff(f12015test[, 'cuml']))
#If the gap exceeds a certain threshold value, assume it's the intersession gap
f12015test['gapflag']= (f12015test['gap']>=300)
#Run an accumulated count of identified gaps to give the session number
f12015test['qsession']=1+cumsum(f12015test[, 'gapflag'])

```

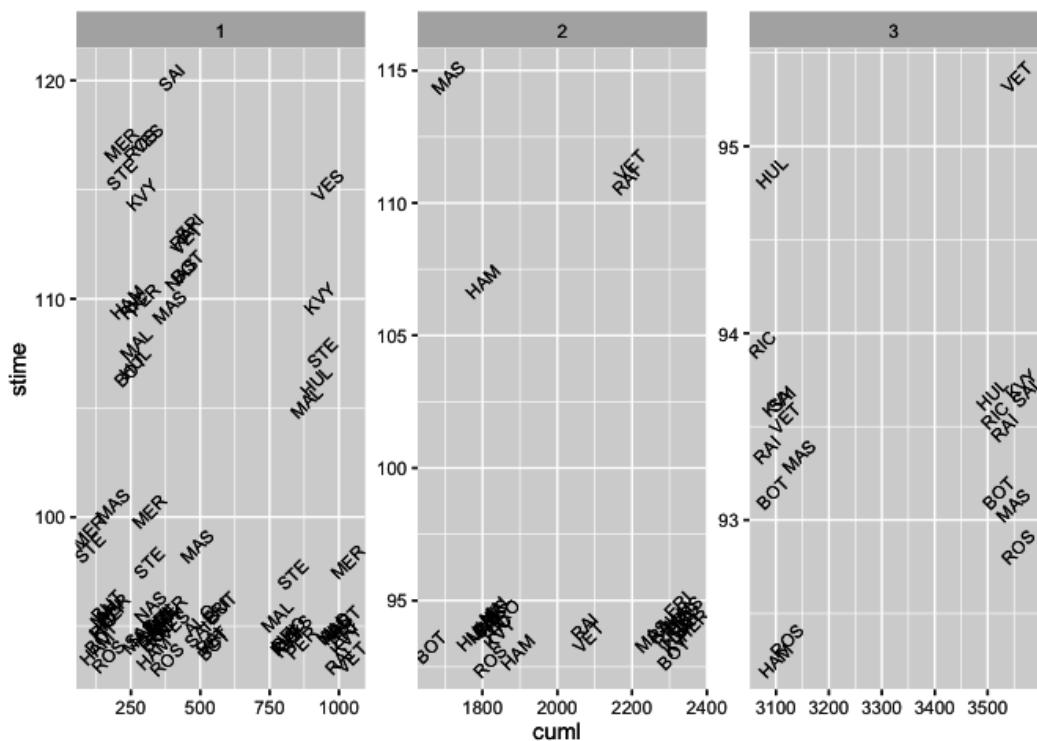
(Note that if a session is red flagged for a period of time, we might incorrectly identify this period as an inter-session gap.)

We can now use the qsession number to separate out the laptimes from the different sessions, limiting the data to show complete laps (that is, not laps that are inlaps or outlaps).

```

g=ggplot(f12015test[!f12015test['outlap'] & !f12015test['pit'],])
g=g+geom_text(aes(x=cuml,y=stime,label=code),angle=45,size=3)
g+facet_wrap(~qsession,scale="free")

```



plot of chunk session_facetscatter

To make these charts easier to read, we need to do something about the labels overflowing the margins, and perhaps use a cleaner theme.

Purple and Green Laptimes in Qualifying

Having identified the different phases of qualifying and associated individual laptimes with the appropriate qualifying session (Q1, Q2 or Q3), we can now calculate the purple and green laptimes *within* those separate sessions.

There are several different ways we can report on purple and green times. For example:

- keeping a running tally of purple and green times as they are recorded through a session (an *online* algorithmic approach);
- displaying just the session best laptime for each driver as the green time (or, for the fastest lap in the session overall, just a single purple laptime per session). This approach

represents an *offline* algorithmic process, in that *all* the laptimes from a session are required before the overall session best green and purple times can be calculated.

Online (Evolving) Purple & Green Qualifying Session Lap Times

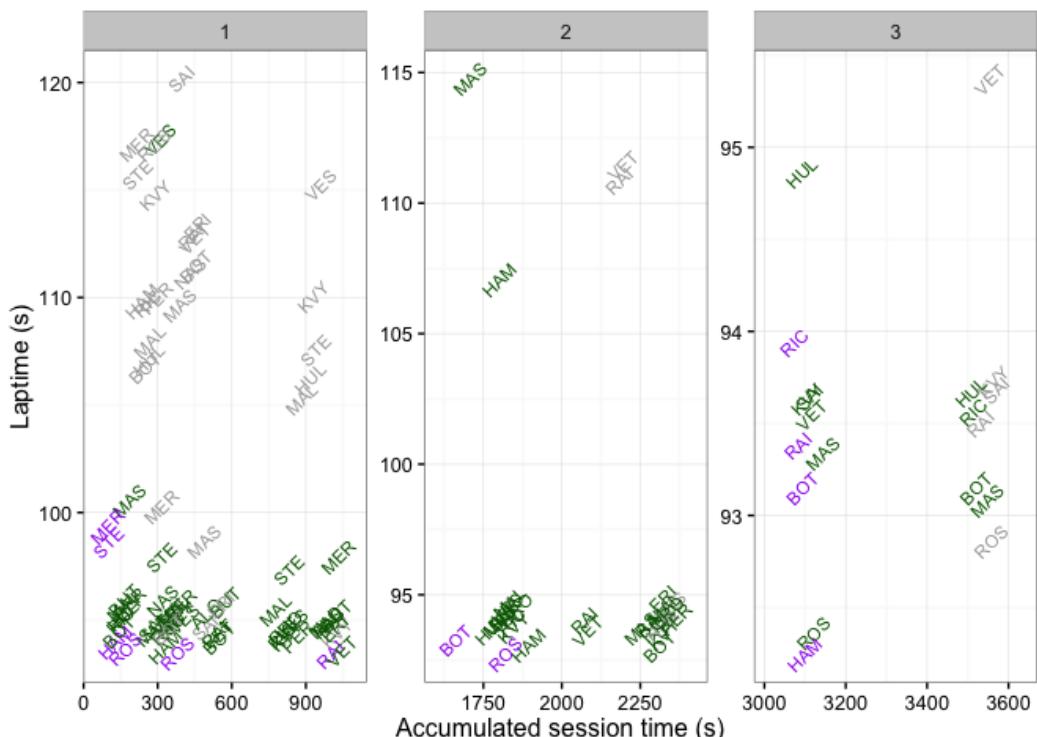
To begin with, we can consider the online algorithmic approach. Let's create a function that will identify the current best time within a session for each driver as the session evolves, treating these as green times. Taking the cumulative minimum driver best time over each session gives us the evolving purple times within a session. We can then use a nested conditional statement to assign an appropriate colour to each time.

```
quali_purplePatch=function(df){
  #Order the drivers by name and lapnumber
  df=arrange(df,name, lapNumber)
  #Group by session and driver, and record each driver's evolving best laptime
  #as their cumulative minimum laptime
  df=ddply(df,.(qsession,name),transform,
    driverqbest=cummin(c(9999,stime[2:length(stime)])))
  #Now order by accumulated laptime
  df=arrange(df,cuml)
  #Group by session and find the cumulative minimum laptime (current purple)
  df=ddply(df,.(qsession),transform,qpurple=cummin(driverqbest))
  #Colour code the laptime
  #The colourx colouring scheme is the online colouring scheme
  df['colourx']=ifelse(df['stime']==df['qpurple'],
    'purple',
    ifelse(df['stime']==df['driverqbest'] & !df['pit'] & !df['outl\
ap'],
      'green',
      'black'))
  df=arrange(df,name, lapNumber)
  df
}
```

This online colouring scheme requires the chart to be read left to right, as time evolves. The colours show how as each time is recorded, it may set a new personal or overall best laptime in the session so far.

```
f12015test=quali_purplePatch(f12015test)

qsession_plot=function(f12015test,col='colourx'){
  g=ggplot(f12015test[!f12015test['outlap'] & !f12015test['pit'],])
  g=g+geom_text(aes_string(x='cuml',y='stime',label='code',
                            colour=paste('factor(',col,')',sep='')),angle=45,size=3)
  g=g+facet_wrap(~qsession,scale="free")
  #Expand the x axis to prevent plotted labels overflowing the margins
  g=g+scale_x_continuous(expand=c(0,100))
  g=g+scale_colour_manual(values=c('darkgrey','darkgreen','purple','red'))
  #Tidy up the axis labels
  g=g+xlab('Accumulated session time (s)')+ylab('Lapttime (s)')
  g+ guides(colour=FALSE) +theme_bw()
}
qsession_plot(f12015test)
```



The online approach captures how best laptimes evolved over the course of a session.

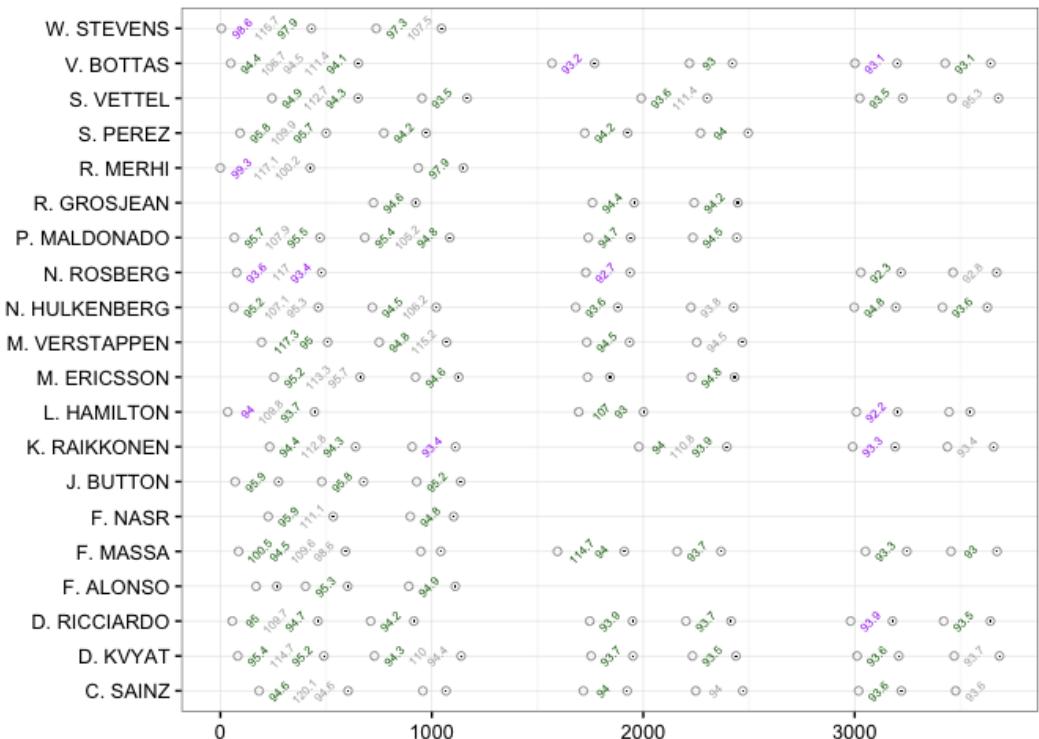
However, this chart can become quite cluttered with green times and it can be hard to see which of the green times is actually the final best laptime recorded for any particular driver by the end of the session.

To capture that information, we need to follow an offline approach where *all* the laptime information is available to us so that we can spot the best overall laptimes.

Revisiting the Session Utilisation Chart

With purple and green times calculated relative to each part of qualifying, we can now revisit the session utilisation chart to using a colouring scheme that treats the times recorded in each part of qualifying separately.

```
#Note that f12015test has now been patched with session relative colourx values
plot_session_utilisation_chart(f12015test)
```



Qualifying session utilisation chart, coloured relative to each part of qualifying using an online algorithm

In this case, where the marks are all well spaced, we can use the online times to show how green and purple times were recorded across each part of the qualifying session.

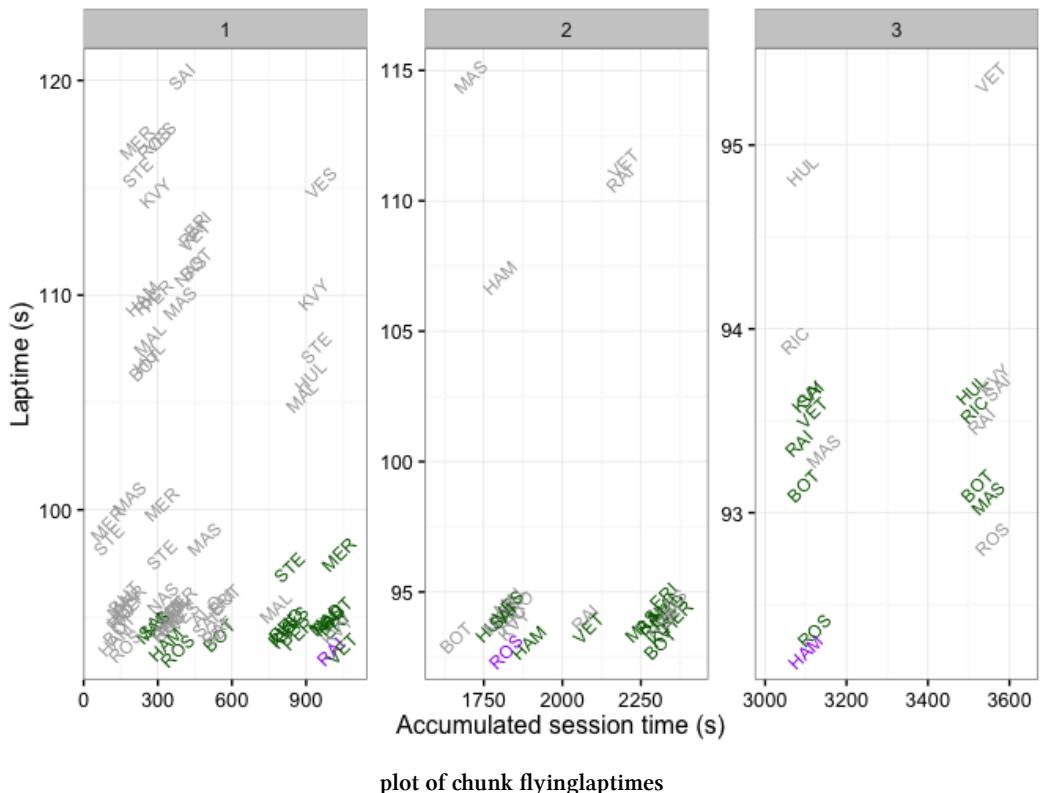
Offline (Session Best) Purple & Green Qualifying Session Lap Times

To find the session best times, we need to identify the best overall laptime for each driver within each session, and final session purple time either as the last recorded purple time or the best purple time. Once again, we can use a bit of logic to determine an appropriate colour label.

```
#Identify as "flying laps" those laps that aren't outlaps or inlaps
df_flying=f12015test[!f12015test['outlap'] & !f12015test['pit'],]
#Group by driver and session to find the best (minimum) laptime for each driver
df_flying=ddply(df_flying, .(code,qsession), transform,
                 driverqsbest=min(driverqbest))
#Find the overall session best (purple) time
df_flying=ddply(df_flying,.(qsession),transform,qspurple=min(qpurple))
#The coloury colouring scheme is the offline colouring scheme
df_flying['coloury']=ifelse(df_flying['stime']==df_flying['qspurple'],
                           'purple',
                           ifelse(df_flying['stime']==df_flying['driverqsbest'],
                                 'green', 'black'))
```

Using this colouring scheme, it's far easier to see when each driver recorded their best laptime and when the session best laptime was recorded.

```
qsession_plot(df_flying,'coloury')
```



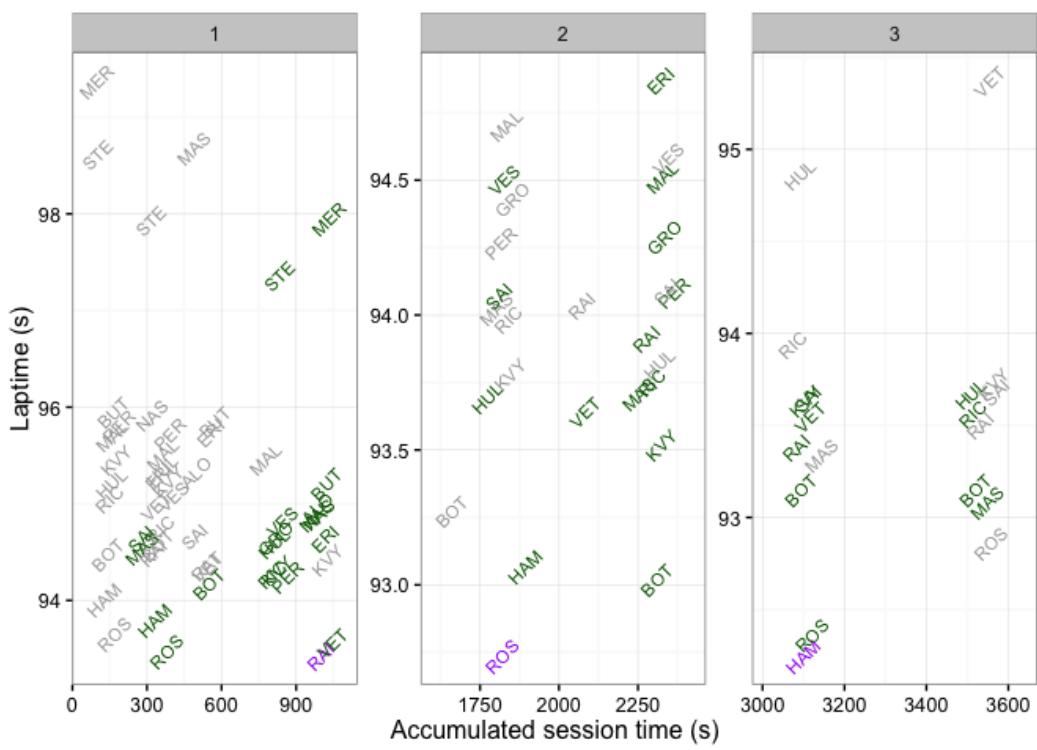
Applying a 107% cutoff limit to a chart laptime axis

To make the laptime related charts easier to read, we might want to automatically limit the displayed area to some multiple of the the session best laptime, such as the 107% limit, also suppressing inlap and outlap times.

```

session107=function(df){
  sessionbest=ddply(df[!df['pit'] & !df['outlap'],],
    .(qsession),
    summarise,
    sbest=min(stime),
    sb107=1.07*sbest)
  df=merge(df,sessionbest,by='qsession')
  df[!df['outlap'] & !df['pit'] & df['stime']<=df['sb107'],]
}
qsession_plot(session107(df_flying), 'coloury')

```



This chart summarises the best recorded laptimes within a session that are within the 107% cutoff time. But where did the actual final cutoff time for the session fall?

How do Session Cut-off Times Evolve Over the Course of Qualifying?

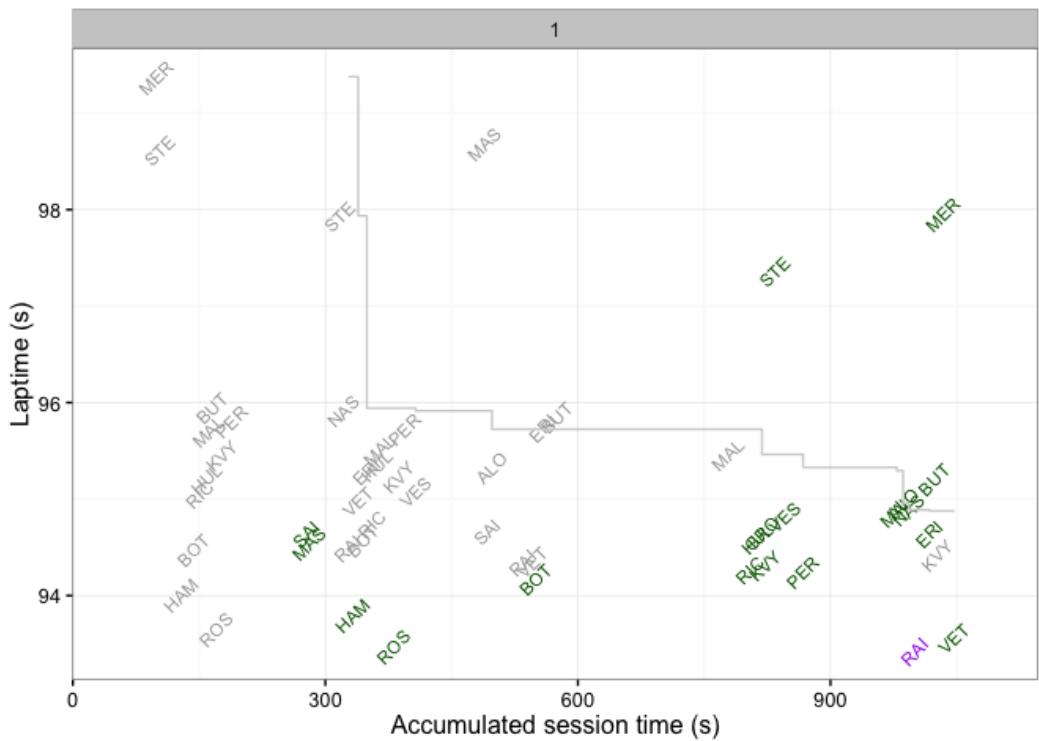
Within each qualifying session, the cut-off time for the session evolves as the drivers record improved laptimes. To calculate the cut-off time at any particular point in a particular session of qualifying, with a cut-off of N drivers going from one session to the next, we cannot simply look at the N^{th} best laptime in the session because it might be the case that one (or more) particular driver achieved more than one of the N fastest laptimes recorded to date.

Instead, as the session progresses, we need to identify the fastest laptime recorded by each driver so far in the session, and then find the Nth best drivers' best laptime to give us the current cutoff time. (We shall define the cutoff time to be the time of the Nth best driver for a cutoff of N drivers. This is then the time that must be beaten in order to make it into one of the top N positions.)

If we order the laptimes by the cumulative time in session, and then group by session and driver, we can find the minimum laptime so far in the session for each driver. At any point, if the number of drivers recording a laptime in a session exceeds the number of cars that can make it through the session (the cutoff number, N), we record the best laptime so far of the N 'th ranked driver in the session as the current session cutoff time.

```
}
```

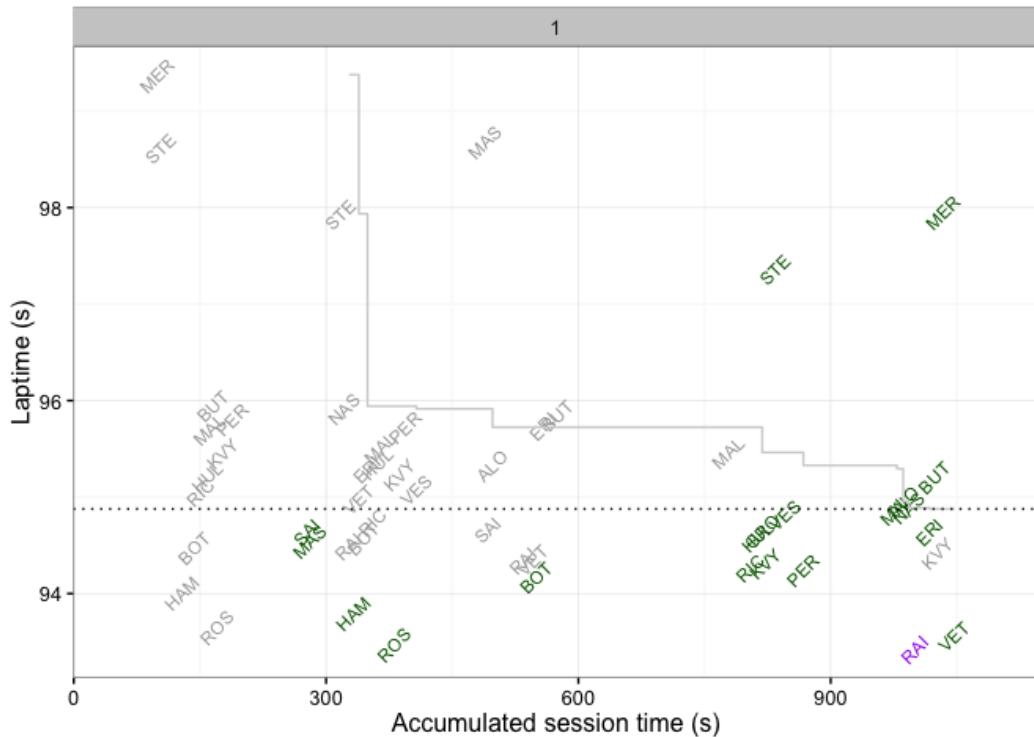
We can now use `geom_step()` to display how a session's cutoff time evolves across the session. The `geom_step()` function connects plotted points using “stairs” that run horizontally and then vertically. That is, the step change appears at the point a laptime is recorded that causes the cutoff time to change, otherwise it continues to run horizontally.



Charting the evolution of the cutoff time in a qualifying session

One final annotation we might make to the chart is to include a dotted line that shows the final cutoff time:

```
dfdd=ddply(cutoffTimes(q1session),.(qsession),summarise,cutoffdbest=min(dbest))
g=g+geom_hline(data=dfdd,
                 aes(yintercept=cutoffdbest),linetype='dotted')
g
```

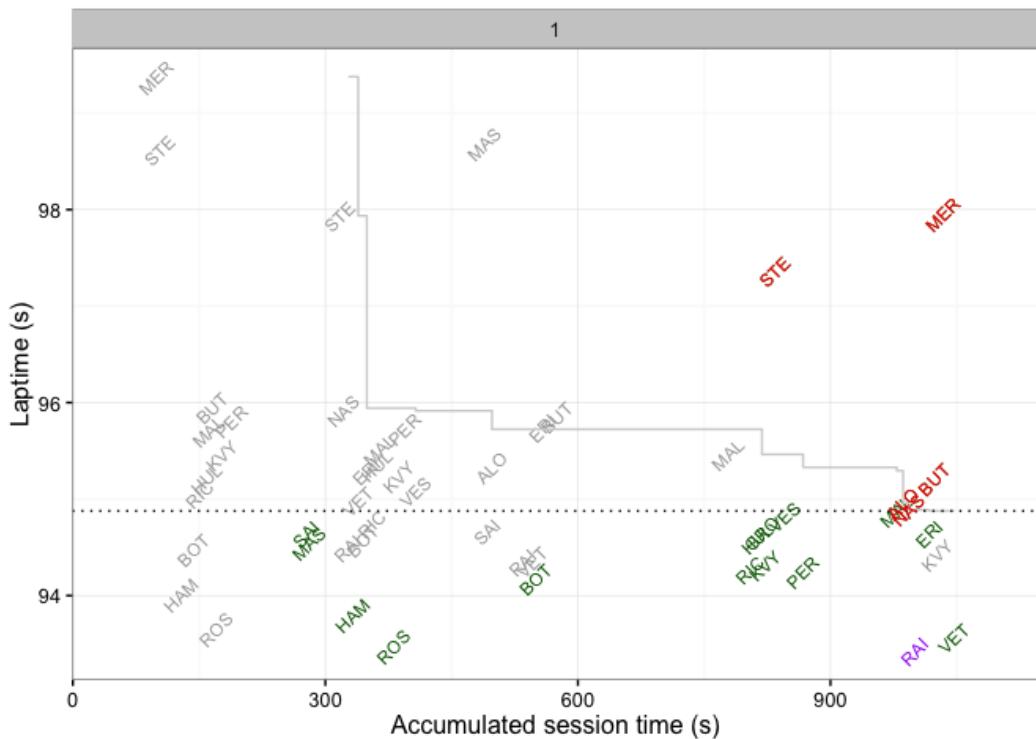


Annotating the cutoff time evolution chart with the final cutoff time

Identifying Drivers That Missed the Cutoff

Finally, let's further annotate the chart to more clearly identify those driver's best laptimes that missed the cutoff.

```
dfx=merge(q1session,dfdd,by='qsession')
g+geom_text(data=dfx[dfx['stime']==dfx['driverqsbest'] & dfx['driverqsbest']>dfx['cutoffdbest']],,
            aes(x=cuml,y=stime,label=code,colour=factor('red')),  
angle=45,size=3)
```



Highlighting drivers' best times that missed the cut

Now we can even more clearly see which drivers' best times missed the cutoff.

Note: it might also make sense to explore the use of the `ggrepel` library function `geom_text_repel()` to try to improve the legibility of the chart in terms of moving overlapping labels.

Summary

In this chapter, we have explored several ways of analysing laptimes associated with a particular qualifying session (Q1, Q2, Q3), such as identifying purple and green laptimes calculated relative to a particular phase of qualifying using both an online and an offline algorithmic approach

We have also described how to track the evolution of the cutoff time across a session by maintaining a running order of the current fastest laptimes recorded to date by each driver and then tracking the top N'th time as a cutoff time.

We can make further use of the overall cut-off time as a threshold value for identifying drivers whose session best laptimes failed to make the cut.

Lapcharts and the Race Slope Graph

The lap chart is one of the iconic motor racing charts, describing for each driver, as represented by a separate coloured line, their race position at the end of each lap. Using a design style strikingly reminiscent of the clean coloured lines of Harry Beck's London Underground map, the lap chart combines a rank based y-axis identifying race position with an x-axis corresponding to lap number to describe the changing position of each car lap on lap over the course of the race. This style of rank tracking chart is often referred to as a *bumps chart* (occasionally corrupted to *bump chart*) following Edward Tufte's reference to it in his book *Beautiful Evidence* (2006) as a way of displaying the results of many years of University of Cambridge "bumps" rowing races.

Along with the lap chart, we'll also look at radically simplified version of it, the *race slope graph*, that just indicates the positions changes going from the grid to the end of the first lap, and the end of the first lap to the end of the race. This chart can also be used as an extension of the slope graphs that show progression through qualifying, or even practice and then qualifying.

Creating a Lap Chart

One of the problems with a lapchart is that it may become hard to read during those periods of the race when many pit stops are taking place and significant position changes appear to have occurred for a lap or two. On the other hand, how confused the chart appears to be in terms of crossed lines that indicate position changes gives an idea, *at a glance*, of how frequently positions changed over the course of a race (at least, as recorded at the end of each lap!), where in the field they took place, and even how processional the race was.

The chart also shows how quickly cars dropped out and the laps on which they did so.

The data we need in to produce a basic lap chart is quite straightforward, comprising just the race positions for each driver at the end of each lap and the lap number.

One source of this information is the downloaded *ergast* database:

```

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

#Load in data relating to drivers' positions on each lap for a particular race

#When querying the database, we need to identify the raceId.
#This requires the year and either the round, circuitId, or name (of circuit)
raceId=dbGetQuery(ergastdb,
                   'SELECT raceId FROM races WHERE year="2012" AND round="1"')
#There should be only a single result from this query,
# so we can refer to its value directly.
q=paste('SELECT driverRef, l.driverId AS driverId, lap, position
         FROM lapTimes l JOIN drivers d
         ON l.driverId=d.driverId
         WHERE raceId=',raceId[[1]])
lapPos=dbGetQuery(ergastdb,q)
#Set the driverRef to be a factor
lapPos$driverRef=factor(lapPos$driverRef)

```

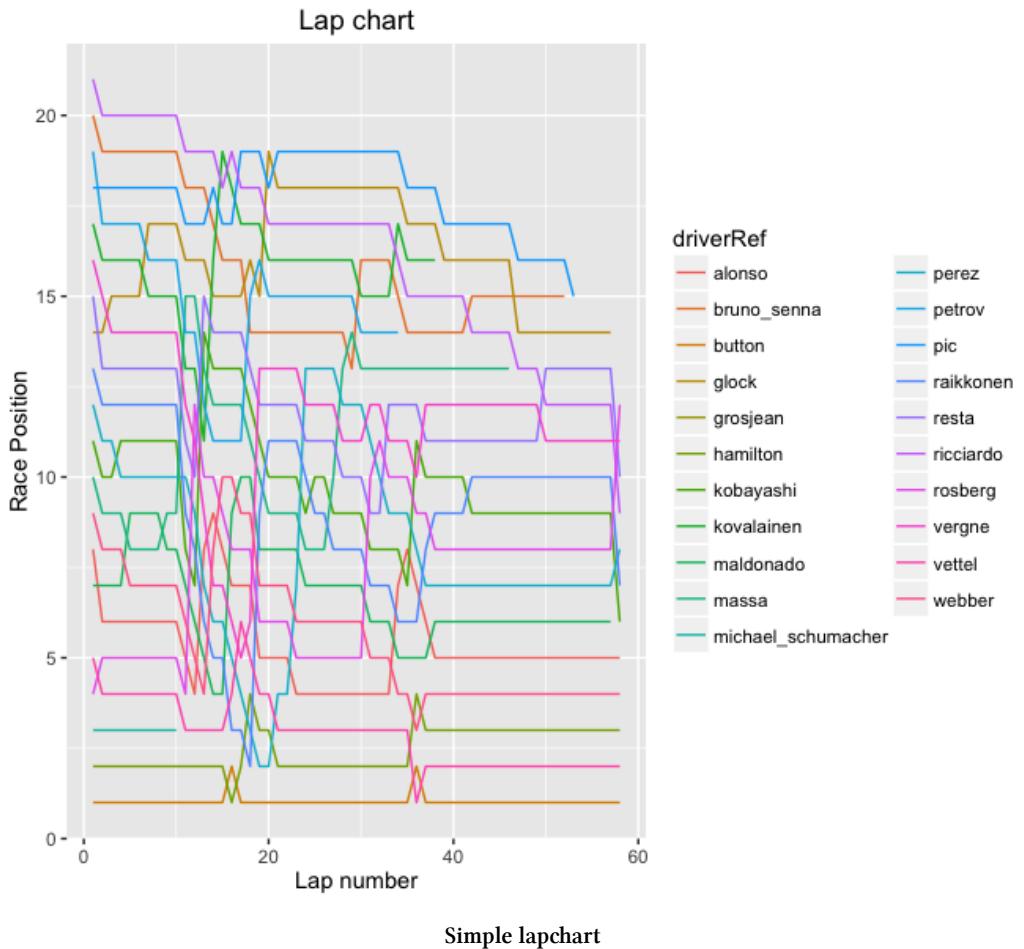
We can also obtain the data directly from the ergast API using the `lapsData.df()` function call, as well as from raw laptime data, though in the latter case we need to take care when calculating race positions from total (summed) elapsed laptime for drivers not on the lead lap.

From this data, we can simply plot the position of each driver against lap, using a separate and differently coloured line to identify each driver.

```

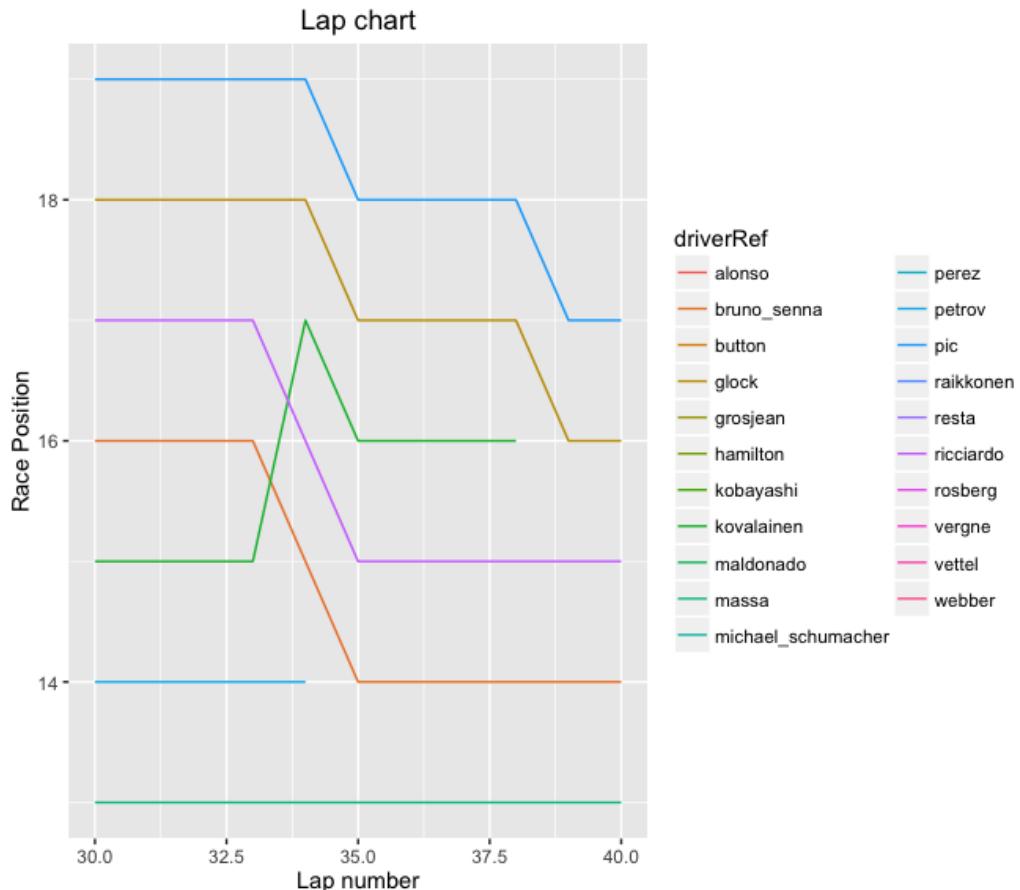
library(ggplot2)
g=ggplot(lapPos)
#The simplest lap chart simply plots each driver's position as a coloured line
g=g+geom_line(aes(x=lap, y=position, col=driverRef))
g=g+labs(title='Lap chart',x='Lap number', y='Race Position')

```



A step down in the trace that marks out the position of the lowest ranked driver on any given lap shows that at least one other driver dropped out on that lap. If you look closely at the example chart you can see where a particular driver's trace stops short of finishing the race, indicating the lap number of the last lap completed by that driver. We can zoom in to the chart by limiting the axis ranges to show such an example:

```
g+ xlim(30,40)+ ylim(13,19)
```

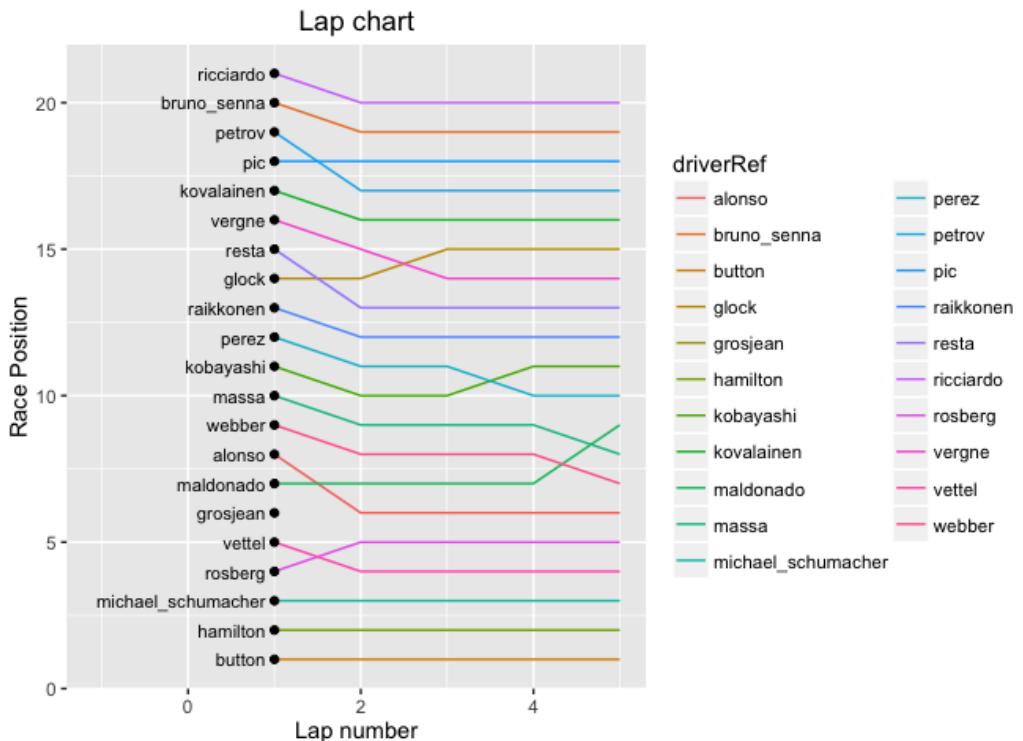


Zooming in on the simple lapchart

One issue with using the lap data is that cars that do not make it to the end of the second lap are not identified. If you look at the original chart, you should notice that that is no mark in position 6 for the first lap - the driver who finished the first lap in sixth did not make the end of the second lap so no line can be drawn.

We can get round this problem by using an additional layer that plots a point showing that there is indeed a driver positioned in a particular place at the end of the first lap even if they didn't make it to the end of the second. We can further clarify which driver is which at the end of the first lap by adding a `geom_text()` label at the start of the chart, the `hjust` parameter setting the right hand side alignment of these labels.

```
#Identify each driver line by name
g + geom_text(data=lapPos[lapPos['lap']==1,],
               aes(x=0.9, y=position, label=driverRef), hjust=1, size=3) +
  geom_point(data=lapPos[lapPos['lap']==1,],
             aes(x=1,y=position)) +
  xlim(-1.1,5)
```



plot of chunk lapChartLap1fixAndLabel

Now we can see that Grosjean was in sixth at the end of the first lap but did not make it as far as the end of the second.

A second problem is that we have no hint from this chart about whether any cars that did make it to the grid failed to make the end of the first or the second lap. Indeed, the lap data has nothing to say about the grid position, so there is no way we can chart this from the lap data in and of itself.

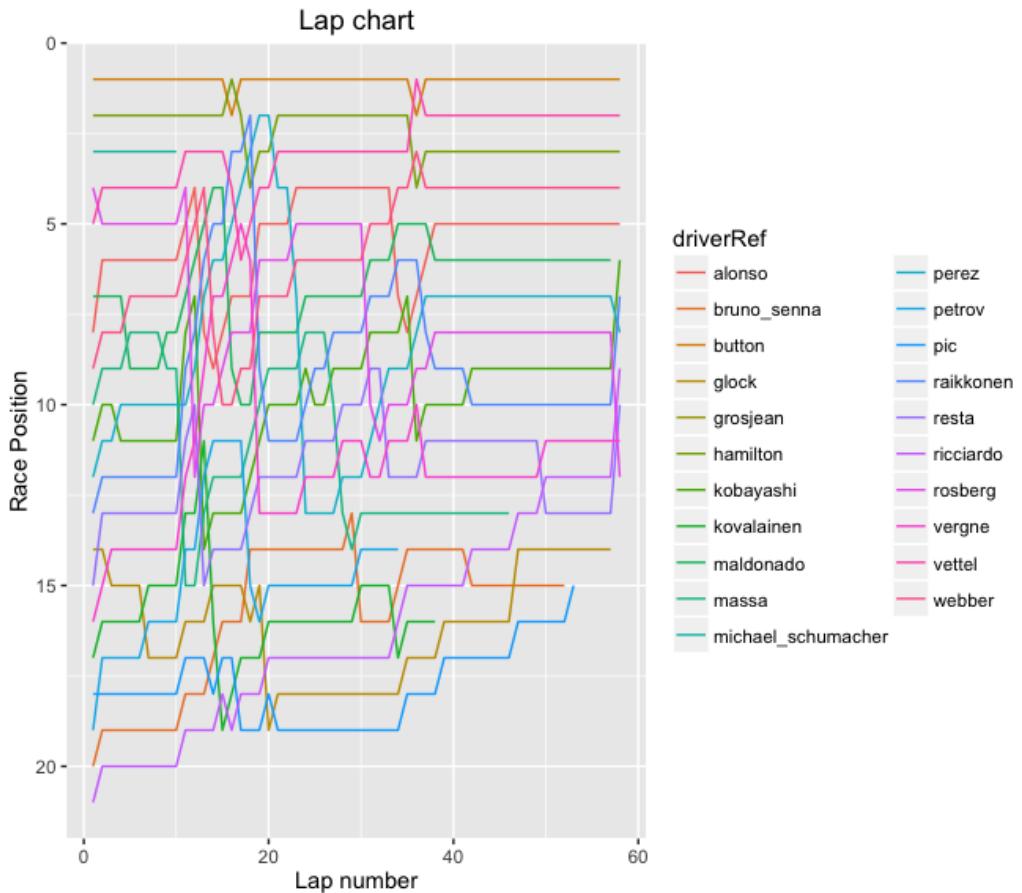
Ordering the Lap Chart Axes

In certain respects, the orientation of the lap chart's vertical y-axis may be perceived as a matter of personal or practical preference. For example, F1 journalist Joe Saward maintains a lap chart for each race that, presumably for convenience, has the lead driver at the top of the chart (*Joe Blogs F1* post on Lap Charts⁴⁷).

We can achieve this effect simply by reversing the direction of the vertical y-axis, placing the driver in *first position* at the *top* of the chart and the driver in last position at the bottom:

```
g=g+scale_y_reverse()
```

⁴⁷<http://joesaward.wordpress.com/2011/04/14/lap-charts/>



Lapchart flipped to show winning driver at the top right of the chart

Annotated Lapcharts

To emphasise those situations where a driver retires or is forced to retire, we can annotate the plot with an additional marker. If we look at the *ergast* database *results* table, we see there is a *statusId* column that identifies the status of each race result. The interpretation of each *statusId* is given in the *status* table.

```
#First five statusId interpretations
dbGetQuery(ergastdb, 'SELECT * FROM status limit 5')
```

```
##   statusId      status
## 1       1    Finished
## 2       2 Disqualified
## 3       3     Accident
## 4       4    Collision
## 5       5     Engine
```

If we grab the interpretations of the *statusIds* into a dataframe, we can merge these with the race results. We could do that using a SQL JOIN in the following query, or we could do it by grabbing the results into an R dataframe from a separate query and merging them with the dataframe containing the original query results.

We can also pull in the three letter code used to identify each driver (this may not be true for some of the older historical data in the *ergast* database) which is perhaps a slightly tidier way of identifying each driver.

```
q=paste('SELECT driverRef, code, rs.driverId AS driverId,
         grid, position, laps,statusId
      FROM results rs JOIN drivers d
      ON rs.driverId=d.driverId
      WHERE raceId=',raceId)
results=dbGetQuery(ergastdb,q)
#Again, we want the driverRef to have factor levels
#results$driverRef=factor(results$driverRef)
status=dbGetQuery(ergastdb,'SELECT * FROM status')
results=merge(results,status, by='statusId')
```

Unfortunately, where drivers are not classified, the race results table shows a *position* of NA, which corresponds to the driver's official classification, rather than giving the race position at the time the driver withdrew. In order to annotate the lap chart using information gleaned from the *status* table using the *position* value for a driver's final lap of the race, we need to merge that information with the lap data information.

The rows we want to annotate amongst the lap data correspond to the laps referenced in the results data for each driver. If we grab the *driverRef* and *laps* data along with the *status*, we can then inject the *status* into the lap data based on *driverRef* and *lap(s)* count (that is, the last lap associated with the driver).

```

results.status=subset(results,select=c('driverRef','status','laps'))
lapPos.status=merge(lapPos,
                     results.status,
                     by.x=c('driverRef','lap'),
                     by.y=c('driverRef','laps'))

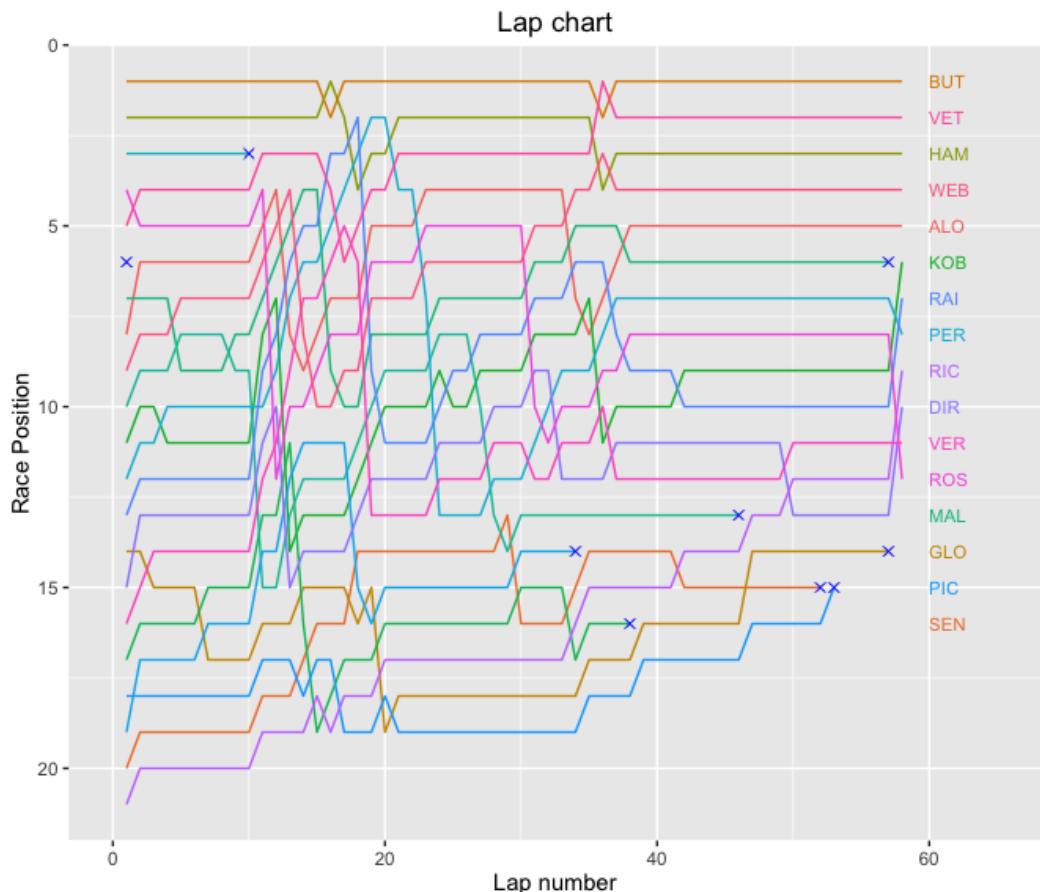
```

Having enriched our data set, we can now overplot onto the lap chart. For example, we might highlight all those final laps completed by a driver where the result *status* is **not** *Finished*, as well as adding driver codes to each row based on their final classification position and a cross symbol (*pch*=4) to denote a non-finisher.

```

g=g+geom_text(data=results,
               aes(x=60, y=position, label=code, col=driverRef),
               hjust=0,size=3) + xlim(0,65) + guides(colour=FALSE)
#Add a cross symbol to identify non-finishers
g+geom_point(data=subset(lapPos.status, status!='Finished'),
              aes(x=lap, y=position), pch=4, size=2, col='blue')

```



Annotating a lapchart to show drivers that didn't finish the race

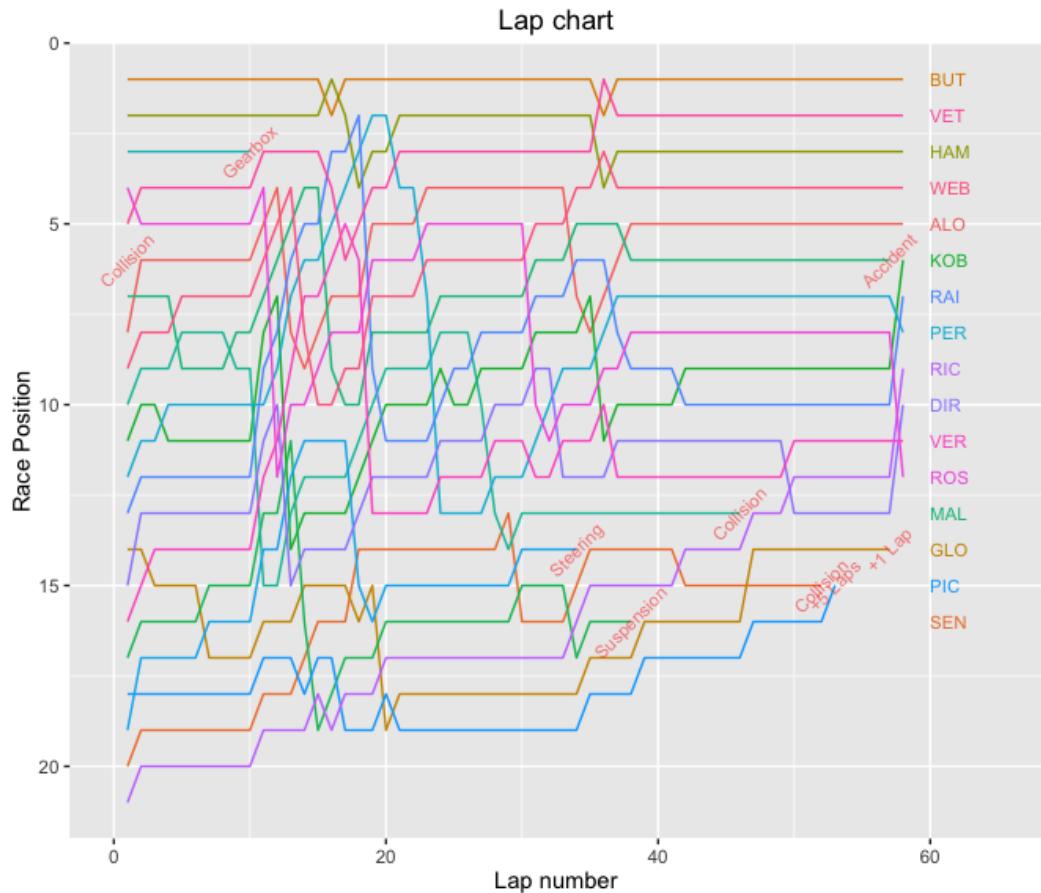
Note how the driver codes are only given for values where the *position* value in the *results* table is set to a non-NA value - that is, a valid final classification. The names of drivers who were unclassified are not listed by this printing method. In addition, as the above chart shows, the placement of the labels at the end of the chart may lose their clear relationship to the driver lines they are supposed to be associated with. This may occur if there are sudden position changes at the end of the race caused by one or more incidents, perhaps, that result in some classified non-finishers that are hard to distinguish between.

The final classification may include drivers who did not in fact finish the race but did complete enough of it to merit classification. It would perhaps be even more informative to distinguish between drivers who were classified but did not finish from those who did not finish and

were not classified, perhaps by overprinting the non-finisher cross with an empty circle to denote the one from the other.

Another way of annotating the chart is with a label describing the reason why a driver failed to finish.

```
#Explicitly identify why a driver failed to finish
g=g+geom_text(data=subset(lapPos.status,status!='Finished'),
               aes(x=lap,y=position,label=status),
               size=3,
               angle=45,
               col='red', alpha=0.5)
```



Annotating a lapchart to show reasons why drivers withdrew from a race

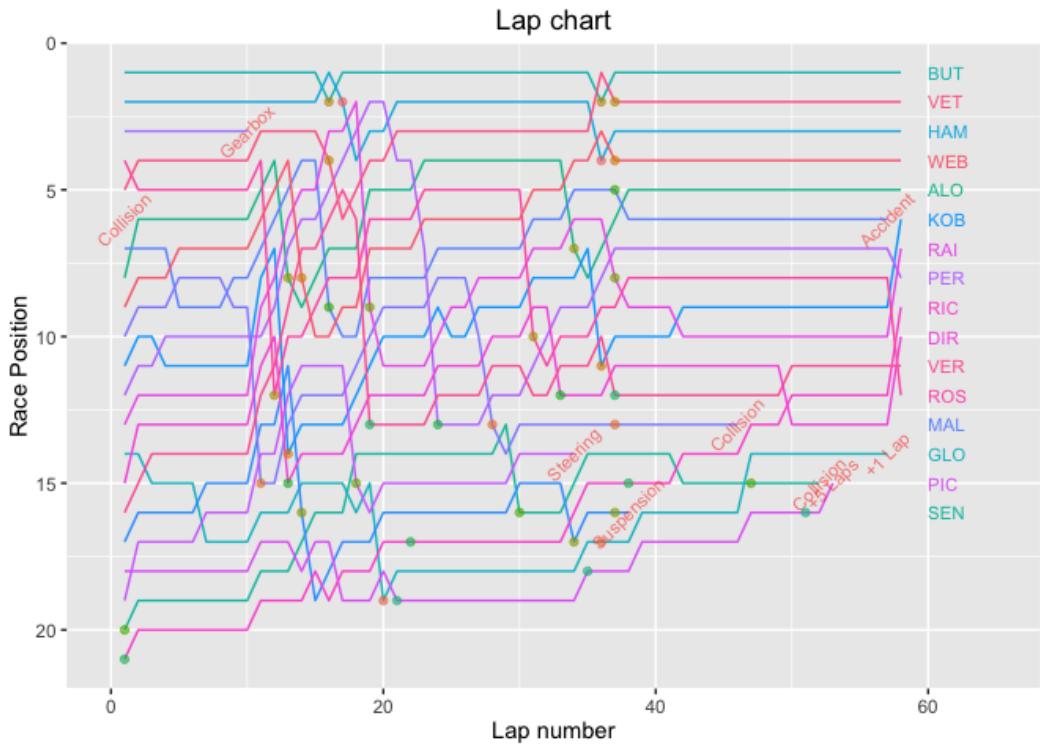
Pit Stop Annotations

A further annotation that is likely to help us identify one of the major reasons for a position change is a pit stop annotation that highlights the lap or laps on which each driver pitted.

```

q=paste("SELECT driverRef, p.driverId AS driverId, lap
        FROM pitStops p JOIN drivers d
        ON p.driverId=d.driverId
        WHERE raceId=",raceId)
pitStops=dbGetQuery(ergastdb,q)
pitStops=merge(pitStops,lapPos[,c("driverRef","lap","position")],
               by=c("driverRef","lap"))
g+geom_point(data=pitStops,aes(x=lap,y=position,colour=factor(driverId)),alpha=0.6)

```



Further annotating the lap chart with pit events

Using this sort of annotation, we can clearly see that changes in first position resulted from pit stop activity even if they lost the position only briefly.

A successful undercut would be shown by a driver pitting one lap before the driver immediately ahead, and taking that position from the driver originally ahead, once the current round of pit stops are completed.

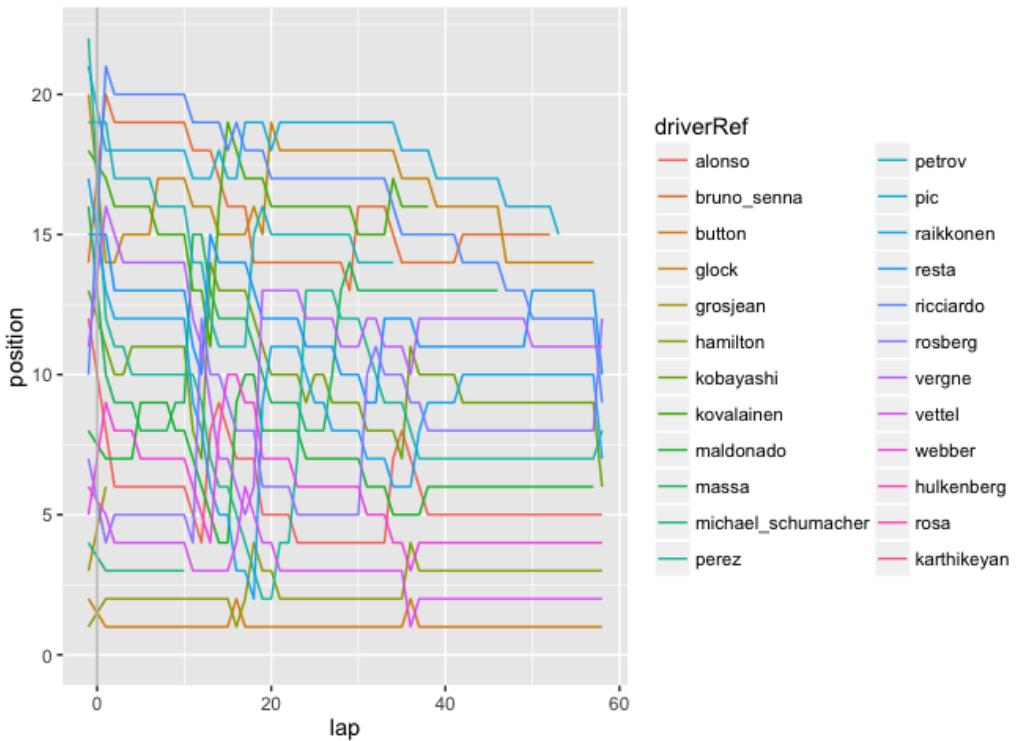
Undercuts are considered in more detail in the section on *Event Detection*.

Extending the Lap Chart - Including Grid Positions

One of the shortcomings with the simple lap chart is that it does not show whether there were any positions changes off the grid. Ideally, we should have added the grid position data prior to the first lap. If we set the grid position at “lap -1” rather than “lap 0”, it gives us some clear separation between the grid positions and the lap positions at the end of lap 1, reinforcing the special, and often incident filled, nature of the first lap, along with the fact that it begins with a standing start.

```
#Create a grid position item at lap number -1
fullRacePos=lapPos[,c('driverRef','lap','position')]
gridPos=results[,c('driverRef','grid')]
gridPos$lap=-1
names(gridPos)=c('driverRef','position','lap')
#Add the grid positions to the race position data
fullRacePos=rbind(fullRacePos,gridPos)

g=ggplot(fullRacePos)
g=g+geom_line(aes(x=lap,y=position,group=driverRef,col=driverRef))
g+geom_vline(xintercept=0,colour='grey')
```



Lap chart with initial grid position marked and distinction between grid position and first lap position highlighted by a grey line

By adding in a vertical start line, we can also emphasise the idea that a change right at the start of the race is a change potentially straight off the grid, or at least on that arose during the first lap.

A Lap Charter Function

Let's now repackage what we've done in the form of a lapcharter function that can accept a championship year and round number and generate an annotated lapchart for it directly.

```

library(DBI)
library(ggplot2)
library(plyr)

getLapPos=function(ergastdb,year,round){
  q=paste('SELECT driverRef, l.driverId AS driverId, d.code AS code, lap, position
    FROM lapTimes l JOIN drivers d JOIN races r
    ON l.driverId=d.driverId AND l.raceId=r.raceId
    WHERE year=',year,' AND round=''',round,'''',sep='')
  lapPos=dbGetQuery(ergastdb,q)
  #Set the driverRef to be a factor
  lapPos$driverRef=factor(lapPos$driverRef)
  lapPos
}

getResults=function(ergastdb,year,round){
  q=paste('SELECT driverRef, code, rs.driverId AS driverId, grid, position, laps, sta\
tusId
    FROM results rs JOIN drivers d JOIN races r
    ON rs.driverId=d.driverId AND rs.raceId=r.raceId
    WHERE r.year=',year,' AND r.round=''',round,'''',sep='')
  results=dbGetQuery(ergastdb,q)
  #Again, we want the driverRef to have factor levels
  #results$driverRef=factor(results$driverRef)
  status=dbGetQuery(ergastdb,'SELECT * FROM status')
  merge(results,status, by='statusId')
}

getPitStops=function(ergastdb,year,round){
  q=paste('SELECT driverRef, p.driverId AS driverId, d.code AS code, lap
    FROM pitStops p JOIN drivers d JOIN races r
    ON p.driverId=d.driverId AND p.raceId=r.raceId
    WHERE r.year=',year,' AND r.round=''',round,'''',sep='')
  pitStops=dbGetQuery(ergastdb,q)
  pitStops
}

getFullRacePos=function(lapPos,results){
  fullRacePos=lapPos[,c('driverRef','code','lap','position')]
  gridPos=results[,c('driverRef','code','grid')]
  gridPos$lap=-1
  names(gridPos)=c('driverRef','code','position','lap')
  #rbind can align the columns by name
}

```

```

#The columns do not need to be presented in the same order
fullRacePos=rbind(fullRacePos,gridPos)
arrange(fullRacePos,code,lap)
}

lapCharter.chart=function(year,round){
  ergastdb =dbConnect(RSQLite::SQLite(), './ergastdb13.sqlite')

  results=getResults(ergastdb,year,round)
  lapPos= getLapPos(ergastdb,year,round)

  fullRacePos=getFullRacePos(lapPos,results)

  pitStops=getPitStops(ergastdb,year,round)
  pitStops=merge(pitStops,
                 lapPos[,c("driverRef","code","lap","position")],
                 by=c("driverRef","code","lap"))

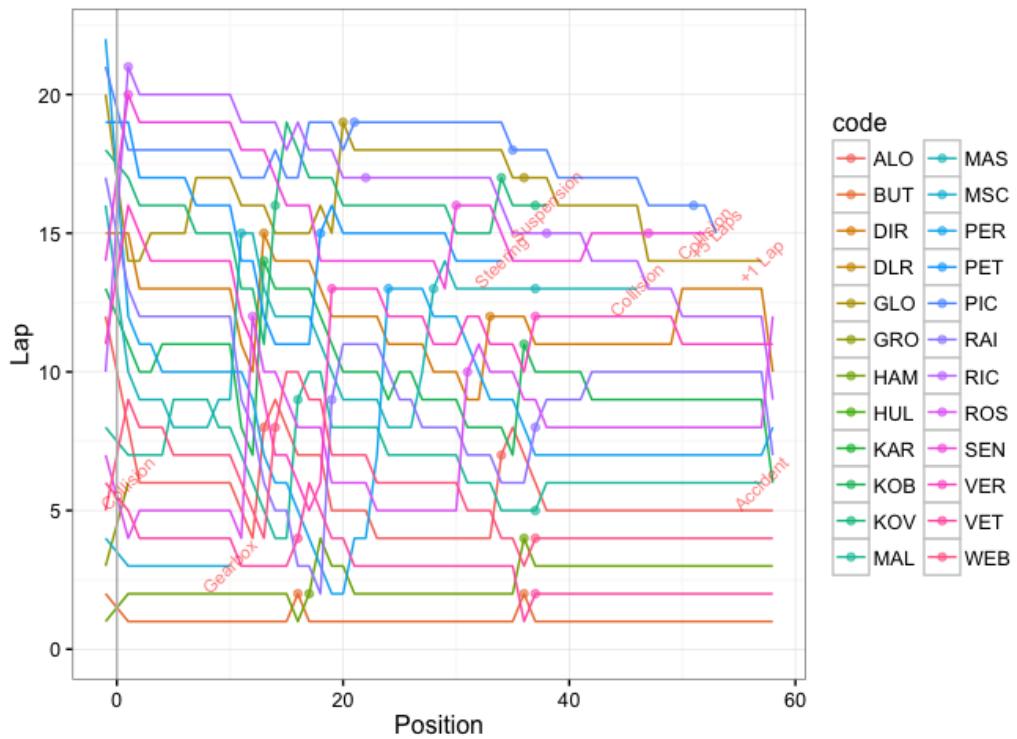
  results.status=subset(results,select=c('driverRef','status','laps'))
  lapPos.status=merge(lapPos,
                      results.status,
                      by.x=c('driverRef','lap'),
                      by.y=c('driverRef','laps'))

  g=ggplot(fullRacePos)
  g=g+geom_line(aes(x=lap,y=position,group=code,col=code))
  g=g+geom_vline(xintercept=0,colour='grey')
  g=g+geom_point(data=pitStops,aes(x=lap,y=position,colour=factor(code)),alpha=0.6)
  g=g+geom_text(data=subset(lapPos.status,status!="Finished"),
                 aes(x=lap,y=position,label=status),
                 size=3,
                 angle=45,
                 col='red', alpha=0.5)
  g+theme_bw() +xlab("Position") +ylab("Lap")
}

```

We can now call the lapcharter function with a specific year and race round.

```
lapCharter.chart(2012,1)
```



Example of the `lapcharter` function

This code can be found in the file `lapCharter.R`⁴⁸ and loaded in with the command `source("lapCharter.R")`.

The chart might be further annotated by drawing on inspiration from the session summary annotations applied to the practice session utilisation charts.

Lap Trivia

With data to hand, there are plenty of opportunities to engage in lap related statistics Inspired by the lapchart, what else might we be able to learn from the lap data?

Laps Led

The number of *laps led* is one of the season long indicators that may be used to help rank drivers at the end of the year. To calculate this popular sports statistic for a single race, we

⁴⁸<https://gist.githubusercontent.com/psychamedia/11187809/raw/lapCharter.R>

might count the number of laps for each driver where the driver was actually leading the race as they crossed the start/finish line (rather than holding first place briefly in the middle of sector 2 on a single lap, for example). The calculation itself is quite straightforward: all we need to do is group the laps by driver and then count the number of laps for which the driver was in first position.

Let's start by counting the number of laps led by each driver in the 2012 Australian Grand Prix.

```
library(plyr)
#Count the number of laps led by each driver
lapsled=ddply(lapPos,.(driverRef),summarise,num=sum(position==1))

#Limit the result to only drivers who led at least one lap
lapsled=lapsled[lapsled['num']>0,]

#Order the result in terms of decreasing count
lapsled[order(-lapsled$num),]

##      driverRef num
## 3      button   56
## 6    hamilton    1
## 20    vettel    1
```

We can also calculate the number of laps led as a proportion, for example, of the number of laps completed by a driver during a season, or during a single race.

```
driverLaps=ddply(lapPos,.(driverRef),summarise,driverLaps=length(position))
lapsled=merge(lapsled,driverLaps,by='driverRef')
#Calculate laps led as a proportion of the number of laps completed by the driver
lapsled$percent=100.0*lapsled$num/lapsled$driverLaps
lapsled[order(-lapsled$percent),]
```

```
##   driverRef num driverLaps    percent
## 1     button    56      58 96.551724
## 2   hamilton     1      58  1.724138
## 3    vettel     1      58  1.724138
```

Counting Laps Led Using a Single SQL Query

A single SQL query can also be used for counting the number of laps led by a particular driver in a range of scenarios. For example, to count the number of laps led by each driver during the 2012 season, we can use a query of the following form:

```
dbGetQuery(ergastdb,
  'SELECT driverRef, COUNT(*) AS lapsled
   FROM lapTimes l, drivers d, races r
   ON d.driverId=l.driverId AND r.raceId=l.raceId
   WHERE year=2012 AND position=1
   GROUP BY driverRef
   ORDER BY COUNT(*) DESC')

##   driverRef lapsled
## 1     vettel     368
## 2   hamilton     229
## 3    alonso     216
## 4     button     136
## 5    webber      66
## 6   rosberg      48
## 7  raikkonen      44
## 8  maldonado      37
## 9 hulkenberg      30
## 10    perez      12
## 11 grosjean        4
## 12    massa        1
## 13    resta        1
```

Were the data available, we could then adapt this sort of query to find the five drivers with the highest counts of laps led to date. At the current time, the *ergast* database only has laptime data starting from the 2011 season. The results below are also calculated using the 2013 version of the database (that is, containing results to the end of the 2013 season).

```
dbGetQuery(ergastdb,
  'SELECT driverRef, COUNT(*) AS lapsled
   FROM lapTimes l, drivers d, races r
   ON d.driverId=l.driverId AND r.raceId=l.raceId
   WHERE position=1
   GROUP BY driverRef
   ORDER BY COUNT(*) DESC
   LIMIT 5')

##   driverRef lapsled
## 1 vettel      1791
## 2 hamilton    445
## 3 alonso      373
## 4 button       232
## 5 webber       194
```

A fuller treatment of laps led calculations and visualisations is provided in the chapter *Laps Completed and Laps Led*.

As well as the *total* number of laps led in a race, season, or career, we can also calculate the number of *consecutive* laps led. See the chapter on *Streakiness* for examples of how to calculate runs and streaks that you can then apply to the laps led data.

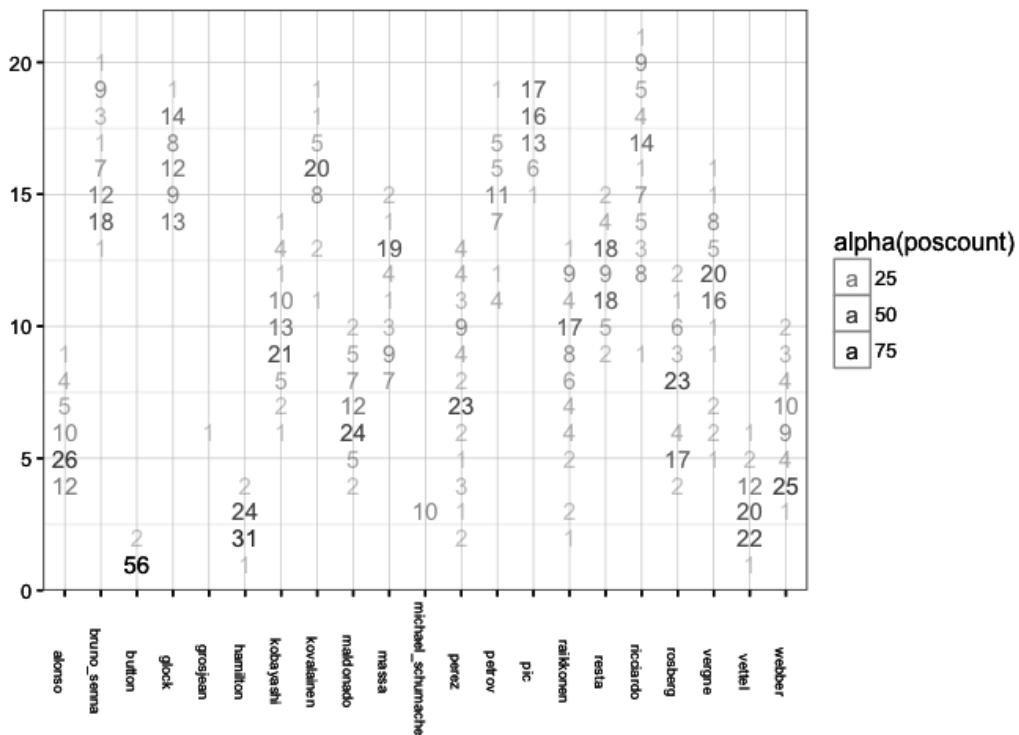
Lap Position Status Charts

For each driver, we can summarise the count of how many laps they completed in each particular race position, not just at the front of the race, using a *lap position status chart* (or perhaps, *lap position summary chart*).

```
#Count the number of laps each driver held each position for
posCounts=ddply(lapPos, .(driverRef,position),
               summarise, poscount=length(lap))

#Set the transparency relative to the proportion of the race in each position
alpha=function(x) 100*x/max(lapPos$lap)
#Rotate the x-tick labels
xRotn=function(s=7) theme(axis.text.x=element_text(angle=-90,size=s))

g=ggplot(posCounts)
#For each driver, plot the number of laps in each race position
g=g+geom_text(aes(x=driverRef,y=position,label=poscount,alpha=alpha(poscount)),
              size=4)
g+theme_bw() +xRotn() +xlab(NULL)+ylab(NULL)
```



Summary chart showing the number of laps in each position for each driver in a single race

The default ordering of the x-axis on this chart is alphabetical. A better ordering might be

one based on final classification.

In many respects, this sort of chart may be thought of as a form of *semi-graphic display*. Semi-graphic displays were a display technique in which text-mode characters could be used to approximate graphical displays from a time when non-alphanumeric graphical displays were still hard to programme. In this particular example, we are using a primarily text based representation combined with a spatial layout to provide a graphical effect. Additional emphasis is provided through the use of font size, along with font weight modeled using transparency.

“If we are going to make a mark, it may as well be a meaningful one. The simplest - and most useful - meaningful mark is a digit.” John W Tukey, “Some Graphic and Semigraphic Displays”, commenting on stem-and-leaf plots, in T A Bancroft,ed., Statistical Papers in Honor of George W Snedecor,1972, p296, also quoted in Edward R. Tufte, Envisioning Information, 1990, p46.

To make this chart a little more meaningful to read, it makes sense to reorder the drivers by their final rank. For drivers that finished the race, this corresponds to their final classification. For drivers that don’t finish the race, we order them according to who got furthest into the race and, for ties, who was higher placed on their final lap for two drivers who went out of the race on the same lap.

We also need to add in any drivers who started but did not make it to the end of the first lap, in which case we should probably order by grid position.

```
#Add drivers who didn't make the end of the first lap
firstLapOuts=results[results['laps']==0 & results['status']!='Did not qualify',
                     c('driverRef','driverId','laps','position')]
names(firstLapOuts)[names(firstLapOuts) == 'laps'] = 'lap'
lapPos=rbind(lapPos,firstLapOuts)
#Generate a ranking based laps completed and position on driver's last lap
finalPos=ddply(lapPos, .(driverId), tail, 1)
finalPos=arrange(finalPos,-lap,position)
finalPos$finalPos=1:nrow(finalPos)
```

We can extend this data frame to include other useful information. For example, the grid position of each driver:

```

finalPos=merge(finalPos,results[,c('driverId','grid','position')]),
           by='driverId',all.x=TRUE)
#Rename the columns
names(finalPos)[names(finalPos) == 'position.x'] = 'position'
names(finalPos)[names(finalPos) == 'position.y'] = 'classification'

```

Let's also add in the position each driver was in at the end of the first lap.

```

finalPos=merge(finalPos,lapPos[lapPos['lap']==1,c('driverId','position')],
               by='driverId',all.x=TRUE)
#Rename the columns
names(finalPos)[names(finalPos) == 'position.x'] = 'position'
names(finalPos)[names(finalPos) == 'position.y'] = 'lap1pos'

```

And finally, the status of each driver in the final classification.

```

finalPos=merge(finalPos,results[,c('driverId','status')]),
             by='driverId',all.x=TRUE)

```

driverId	driverRef	lap	position	finalPos	grid	classification	lap1pos	status
1	hamilton	58	3	3	1	3	2	Finished
3	rosberg	58	12	12	7	12	4	Finished
4	alonso	58	5	5	12	5	8	Finished
5	kovalainen	38	16	18	18	NA	17	Suspension
8	raikkonen	58	7	7	17	7	13	Finished

In addition to ordering the drivers, we can overlay additional information on to the chart. In the following example, we add:

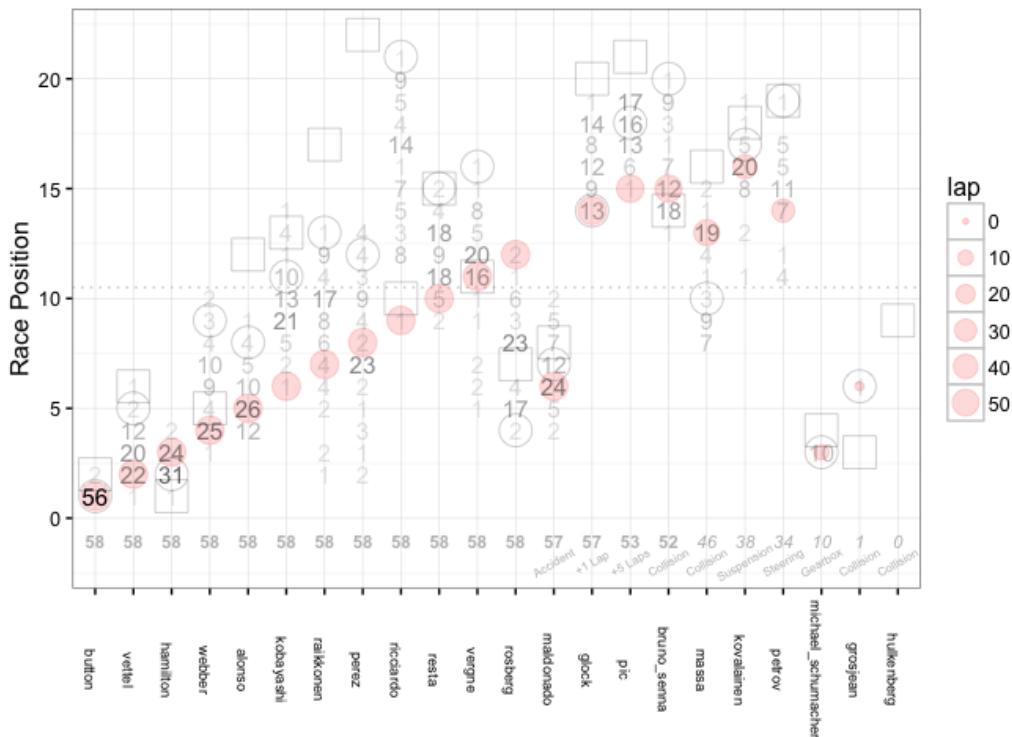
- *grid position*: identified using an empty grey square;
- *race position after the first lap*: identified using an empty grey circle;
- *race position on each driver's last lap*: y-value (position) of corresponding pink circle;
- *points cutoff line*: a faint grey dotted line to show which positions are inside - or out of - the points;
- *number of laps completed by each driver*: size of pink circle;
- *total laps completed by each driver*: greyed annotation at the bottom of the chart;

- whether a driver was classified or not: the total lap count is displayed using a bold font for classified drivers, and in italics for unclassified drivers;
- finishing status of each driver: classification statuses other than *Finished* are also recorded at the bottom of the chart.

```

#Reorder the drivers according to a final ranked position
g=ggplot(finalPos,aes(x=reorder(driverRef,finalPos)))
#Highlight the points cutoff
g=g+geom_hline(yintercept=10.5,colour='lightgrey',linetype='dotted')
#Highlight the position each driver was in on their final lap
g=g+geom_point(aes(y=position,size=lap),colour='red',alpha=0.15)
#Highlight the grid position of each driver
g=g+geom_point(aes(y=grid),shape=0,size=7,alpha=0.2)
#Highlight the position of each driver at the end of the first lap
g=g+geom_point(aes(y=lap1pos),shape=1,size=7,alpha=0.2)
#Provide a count of how many laps each driver held each position for
g=g+geom_text(data=posCounts,
              aes(x=driverRef,y=position,label=poscount,alpha=alpha(poscount)),
              size=4)
#Number of laps completed by driver
g=g+geom_text(aes(x=driverRef,y=-1,label=lap,
                  fontface=ifelse(is.na(classification), 'italic' , 'bold')),,
              size=3,colour='grey')
#Record the status of each driver
g=g+geom_text(aes(x=driverRef,y=-2,label=ifelse(status!='Finished' , status, '')),,
              size=2,angle=30,colour='grey')
#Styling - tidy the chart by removing the transparency legend
g=g+theme_bw() +xRotn() +xlab(NULL)+ylab("Race Position") +guides(alpha=FALSE)

```



Lap position summary chart with drivers ordered by final classification. The chart is further annotated with the position the driver was in on their last lap (pink filled circle, size proportional to number of laps completed), their grid position (empty square) and their position at the end of the first lap (empty circle).

Viewing the chart, we see that whilst Maldonado was classified in the middle of the order (13th), he actually made it a long way into the race (from the size of the pink circle), before exiting the race in sixth position. He literally “fell out” of the order due to an accident in the last lap of the race. The chart also shows how Rosberg’s race fell to pieces at the very end of the race after a start that saw him move from 7th on the grid to 4th at the end of the first lap.

To identify drivers in the points, we need to look along the dotted line to find the last scoring position at the end of the race, and then drop down to the driver names. Adding rank position numbers would add to much additional clutter, I think?

The chart clearly shows how both Hamilton and Vettel each completed a significant number of laps in both second and third positions. However, this chart *does not* tell us whether these laps were grouped in long runs (for example, Hamilton ahead in the first part of the race, Vettel in the latter part), or whether the drivers were battling throughout and changing position every few laps. To learn that, we need to look to another indicator that is more indicative of

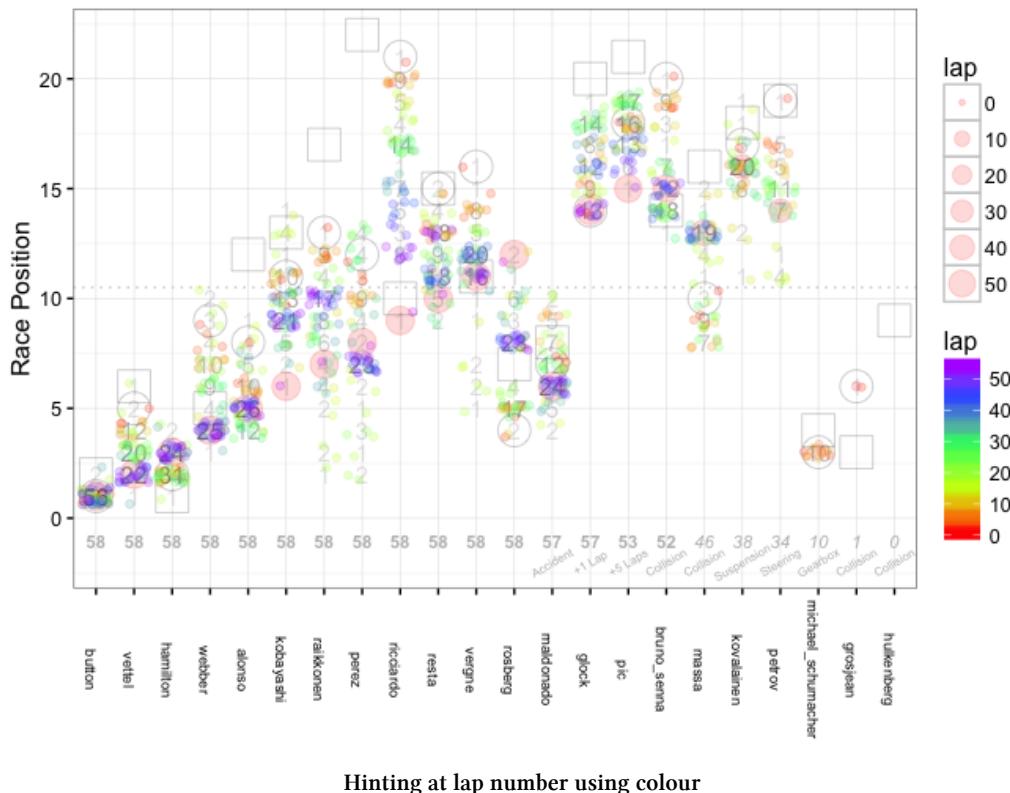
position changes.

However, one thing the race summary chart does not show is the effect of any grid penalties. If a driver qualified well but suffered a grid penalty, we might expect them to make a charge from their position on the grid. It could therefore make sense to further annotate the chart with a small device showing where each driver's qualifying session classification.

The Race Summary Chart

One way of demonstrating how the race evolved on a lap by lap basis is to use *colour* to represent the lap number and a jittered scatterplot to place marks for each driver showing their position by (coloured) lap. Setting a low alpha level makes the dots semi-transparent, which means a large collection of dots will present a more solid appearance.

```
#The jitter plot spreads out multiple points with the same co-ordinates
g+geom_jitter(data=lapPos,aes(driverRef,position,colour=lap),alpha=0.2)+  
  scale_colour_gradientn(colours = rainbow(5))
```



In the coloured view of this chart, we can now see that Vettel was running in fourth in the early stages of the race, then third for a while, then second for the latter part of the race; Hamilton started off in second (a faint hint of red marks, with the more concentrated marks of the green middle stage of the race on a higher graphical layer), and then ran in third for the later stages. From the apparent late stage running of Kobayashi to Maldonado, compared to the final classification, it seems that an accident to Maldonado at the end of the race may have had a significant effect on the final result?

Tweaking the transparency levels and colours of all the layers could possibly improve the legibility of this chart further. It might also be interesting to flip the x and y axes, with the higher placed drivers at the top of the chart and the lower placed drivers at the bottom, to see if it makes reading off the performance of each individual driver any easier.

Another form of colouring may also help to draw out the story of the race: colouring by stint rather than lap.

Position Change Counts

A simple indicator for highlighting how many times a position changed hands is a count of driver changes between consecutive laps for each position.

Calculating the difference from the laptime data is straightforward enough, but we should also bear in mind the need to count any changes between grid position and the position at the end of the first lap.

We'll start by finding changes in position that occur between consecutive laps.

```
#Work about this in a roundabout way
#Start by ordering the lap times by position and lap
#For each position, find laps where the driverId is different to the previous one
poschange=ddply(arrange(lapPos, position,lap),
                 .(position),
                 transform,
                 change=(diff(c(-1,driverId))!=0))
```

One problem with this is that the first *change* value is actually a dummy value. We really want it to state whether there was a change from the grid position.

Let's create a dataframe that allows us to detect changes between grid position and the position at the end of the first lap.

```
firstLap=merge(poschange[poschange["lap"]==1,],
               results[,c('driverId','grid')], by='driverId')
#The position changed after the first lap if it differs from the grid position
firstLap$change= (firstLap$grid != firstLap$position)
```

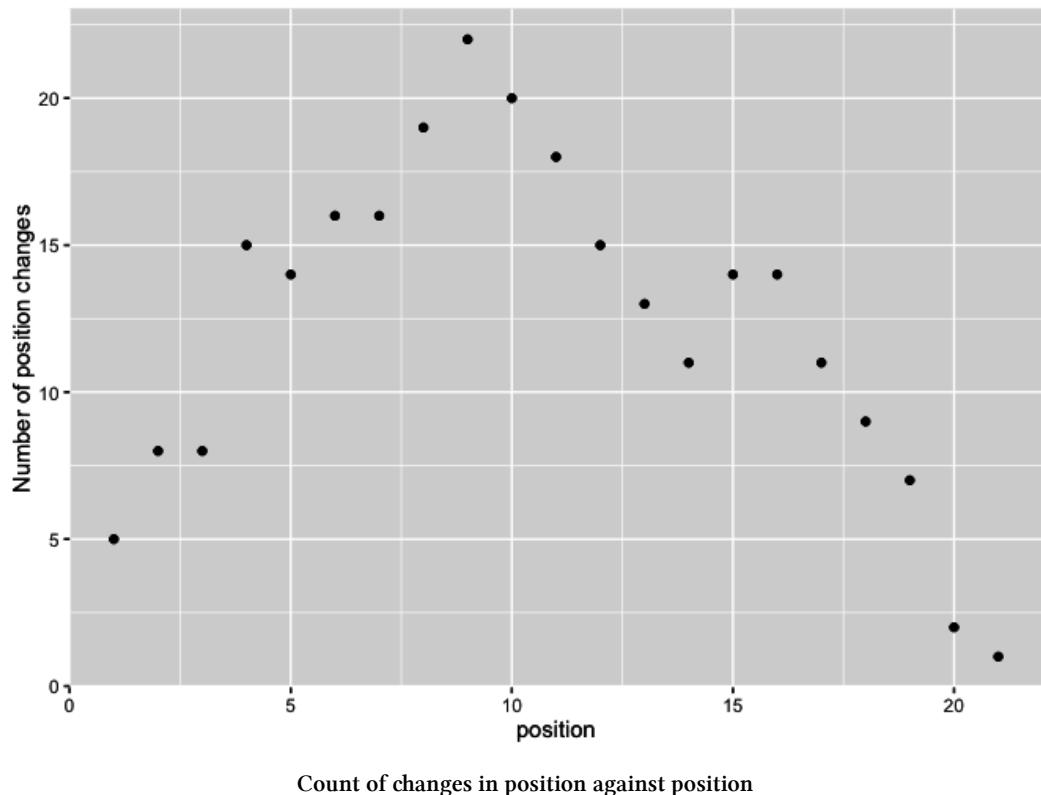
driverId	driverRef	lap	position	change	grid
1	hamilton	1	2	TRUE	1
3	rosberg	1	4	TRUE	7
4	alonso	1	8	TRUE	12

We can now set the original dummy value for the change state at the end of the first lap to correctly reflect whether there was a change in position at the end of the first lap from the original grid start.

```
poschange[poschange[ "lap"]==1 , ]$change = arrange(firstLap,position)$change
```

We can now simply count how many position changes there were in each race position across the race as a whole.

```
#For each position, count the number of changes
dch=ddply(poschange,.(position),summarise,changes=sum(change))
#Plot them as a scatter plot
g=ggplot(dch)+geom_point(aes(x=position,y=changes))
g+ylabel("Number of position changes")
```



This tells us, for example that first position changed hands 5 times during the race. However, looking back at one of the previous lap charts, we see that several of these were quick succession changes associated with pitting. To get a more useful measure of churn, we perhaps need to consider alternative metrics, for example by paying more heed to the number of laps

a driver held a position, or taking note of whether a driver lost a position temporarily due to a pit stop. In other words, perhaps we need to take a more considered view around lap position *streaks* or *runs*. This in turn might suggest additional ways of re-presenting the *Lap Position Status/Summary Chart*. See the chapter *Keeping an Eye on Competitiveness - Tracking Churn* for measuring churn in rankings within and across seasons that might also be used to track churn within a race.

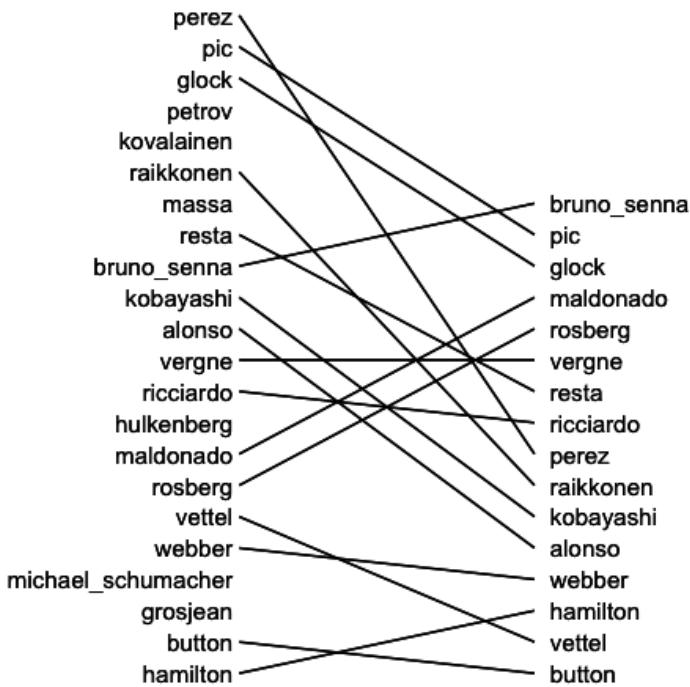
As well as simply tracking position changes, it might also be worth distinguishing those position changes that arise from one or more cars exiting from the race. For example, if the car leading the race is forced to retire, every other car still in the race will benefit from a position change on that lap.

The Race Slope Graph

A simplification of the lap chart as a *slope graph* connects the starting and finishing rank positions of each driver with a single straight line. Crossed lines show how one driver changed order relative to another when comparing their grid and final classification positions.

```
#Let's use a clean white theme
slopeblank=theme(panel.border=element_blank(),panel.grid.major=element_blank(),
                 panel.grid.minor=element_blank(),
                 axis.line=element_blank(),axis.ticks=element_blank(),
                 axis.text = element_blank())

g=ggplot(results[results['grid']>0,])
g=g+geom_text(aes(x=0.98,y=grid, label=driverRef),hjust=1,size=4)
g=g+geom_text(aes(x=2,y=position,label=driverRef),hjust=0,size=4)
g=g+geom_segment(aes(x=1,xend=1.95,y=grid,yend=position,group=driverRef))
g+theme_bw()+xlim(0,3) +xlab(NULL)+ylab(NULL)+slopeblank
```



Sketch of a simple skop graph comparing grid and final positions

Adding the grid position and final classification as explicit labels may help further improve the glanceability of this chart, as might using standard three letter driver codes rather than driver name labels.

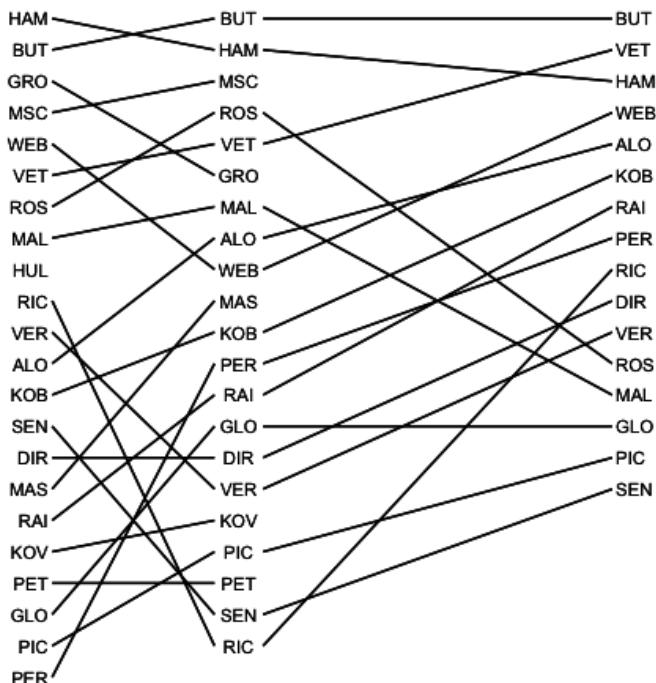
Slope graphs are often presented using real number interval scales, rather than rank based scales, as a means of demonstrating the distance between grouped elements on each axis as well as the extent of the change in values from one situation to the next for each element. Although axis lines and regular tick marks are omitted as part of the very clean, minimal aesthetic of the chart, the y-values associated with each element are displayed alongside each element and as such are easily read.

We can extend the technique to show the relative rank positions of each driver going from the grid to the end of the first lap and from their first lap position to their final rank position.

```

raceHighlights=merge(results[,c('driverRef','code','grid','position')],
  lapPos[lapPos['lap']==1,c('driverRef','position')],
  by='driverRef',all.x=TRUE)
g=ggplot(raceHighlights[raceHighlights['grid']>0,])
#Place the labels
g=g+geom_text(aes(x=0.98,y=grid, label=code),hjust=1,size=3)
g=g+geom_text(aes(x=2.0,y=position.y,label=code),,size=3)
g=g+geom_text(aes(x=4.02,y=position.x,label=code),hjust=0,size=3)
#And then place the line segments
g=g+geom_segment(aes(x=1,xend=1.87,y=grid,yend=position.y,group=code))
g=g+geom_segment(aes(x=2.13,xend=4,y=position.y,yend=position.x,group=code))
#Finally, render the chart using the clean theme
g+theme_bw() + xlim(0,5) + xlab(NULL) + ylab(NULL) + slopeblank + scale_y_reverse()

```



Extended slope graph showing grid, end of lap 1, and final positions

If we add additional text labels to the chart, we can increase the amount of information communicated without overloading it too much. For example, we might add text annotations

that identify the number of pit stops each driver had, the number of laps in the lead position, the number of consecutive laps the driver held their classified position for at the end of the race, or the lap number at which a driver retired. We might also include another rank ordered column depicting the qualifying position of each driver to identify any drivers who took a grid penalty going in to the race.

We can also *back refer* information traces. For example, we could use italics to emphasise a driver in the grid column who did not make the end of the first lap, or highlight drivers in the end of lap 1 column who were not classified at the end of the race. (This column may be the appropriate place to record the lap on which a retired driver retired.)

The slope graph could also be extended by adding a semi-transparent lower layer showing how far into the race each unclassified driver got and the position they were in at the end of their race by projecting lines into the space between the end of lap and final classification columns.

In the same way that the race slope graph reapplys the techniques used to create the qualifying slope graph, the same idea can be further extended to generate “race weekend” slope graphs. These could show the final classification at the end of each practice session, after each part of qualifying, the grid position (after any grid penalties, for example), positions at the end of the first lap and the end of the race, and the final official classification. Dotted vertical lines could be used to identify those transitions where position changes were due to stewards’ actions, or the application of penalties in order to distinguish them from driver performance measures.

Further Riffs on the Lapchart Idea

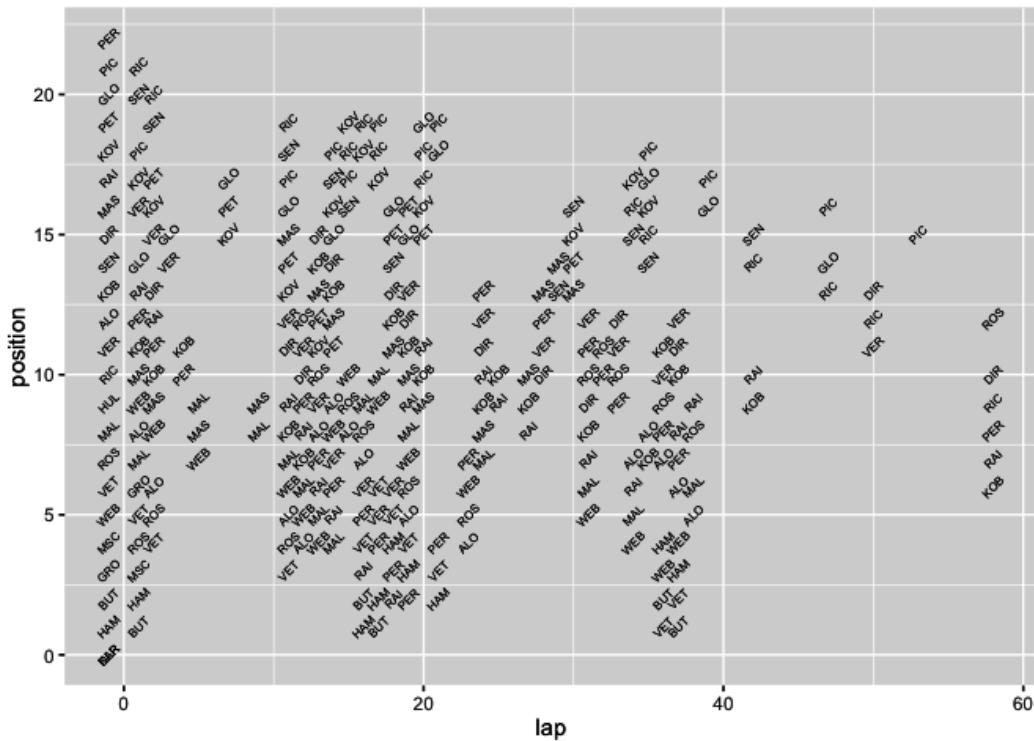
In a post from April 2011 on Lap charts...⁴⁹, F1 journalist Joe Saward gives a couple of examples of lap charts he had made during the course of a particular race.

If you look closely at the lapcharts shown in that post, you will see how lines are used to highlight position changes; this suggests a style of chart where lines are *only* used to show position changes. The easiest way to plot this would be to generate a *position change* dataset within which each row specified a particular driver, their previous lap position and their current lap position *if the two positions are different*.

We can quickly doodle a sketch to get a feel for what such a chart might look like. Firstly, let’s just record the drivers as they change position:

⁴⁹<http://joesaward.wordpress.com/2011/04/14/lap-charts/>

```
fullRacePos.df=function(year,round){  
  ergastdb =dbConnect(RSQLite::SQLite(), './ergastdb13.sqlite')  
  results=getResults(ergastdb,year,round)  
  lapPos= getLapPos(ergastdb,year,round)  
  getFullRacePos(lapPos,results)  
}  
  
fullRacePos = fullRacePos.df(2012,1)  
  
#A position change for a driver is identified if there is difference  
#in position at the end of consecutive laps  
poschange=ddply(arrange(fullRacePos, code,lap),  
  .(code),  
  transform,  
  change=(diff(c(-1,position))!=0))  
  
g=ggplot(poschange[poschange['change']==TRUE,])  
g+geom_text(aes(x=lap,y=position,label=code),size=2,angle=45)
```



Text plot sketch showing laps where drivers changed position during a race

This chart shows just the position changes for each driver over the course of the race; the leftmost labels correspond to grid positions, the other marks record the lap on which a driver attained a particular position.

Emphasising losses in position using italicised labels, or pit stop laps though the use of an underline, would add further richness to this chart. Adding the final race positions to the right hand margin would also improve the utility of this chart as a race summary device.

Further Ideas

Joe Saward's lapchart notes also identify other techniques that we might be able to draw on, as well as some thoughts on the personal design of an effective lapchart:

A list of numbers does not tell the whole story; it does not indicate the gaps between the cars or any incidents that might occur. So lap charters develop their

own personal systems to record the action on a given lap. On my personal chart there are different marks and squiggles each of which record a particular point. If a car is catching the one in front, for example, I will often put an arrow. When cars are close together this becomes a line. When cars are nose to tail, there may be two lines to indicate a battle. If there are three lines it usually indicates that an accident is about to happen as the fight is getting out of control. You will also see small horizontal lines in places between numbers, which indicate gaps, although I do not always include these. A circled number indicates a car which has pitted. A puff of smoke beside a number indicates that a car has blown up or is trailing smoke. A circular arrow around a number or initials indicate a spin... When there is a passing manoeuvre I tend to put in a pair of crossed lines so I can see what happened on what lap. At the bottom of the chart you see a line which indicates the cars that have been lapped, or in some cases, lapped twice.

Joe Saward, *Lap Charts*⁵⁰

If you haven't already read it, the post is well worth reading in full.

Lap Charts as Network Graphs

Several years ago I started to explore the idea of representing the laptimes and positions of each driver within a race as disconnected components of a graph (in the sense of Graph Theory, that is, as a network) with each lap representing a node and consecutive laps as nodes connected by a directed edge showing which lap preceded the other. *See the section on Event Detection for an example of how we might use graph theory to detect a particular sort of event.*

This approach led to some of the chart styles described at bottom of a blog post entitled Thoughts on a Couple of Possible Lap Charting Apps⁵¹. The use of graph theory ideas is potentially a very rich one, with opportunities for representing values as edge weights that can be used to influence the layout of connected nodes, as well as node attributes that can be visualised using colour or size.

This area is left as a topic for future exploration...

⁵⁰<http://joesaward.wordpress.com/2011/04/14/lap-charts/>

⁵¹<http://blog.ouseful.info/2011/04/30/thoughts-on-a-couple-of-lap-charting-apps/>

Summary

In this chapter, we have explored the construction of one of the iconic racing charts, the *lap chart*, as a means of recording driver positions at the end of each lap. These charts can be annotated to display additional information, such as pit stop laps, as well as the reason for any early finish. A slope graph was also presented as an idealisation of certain key points in the lapchart (grid position, position at end of first lap, position at end of race), with the observation that a “race weekend” slope graph might usefully summarise classifications across all sessions, as well as highlighting any position changes due to stewards’ actions.

We also used laptimes to generate some alternative charts: *lap position status charts* provide a summary count of how many laps each driver spent in each position, with additional annotations also capturing grid position and the position at the end of both the first lap and the end of the race; *race summary charts* used an additional colour field to try to capture when in the race each driver held a particular position.

A simple *position change count chart* hinted at positions where changes were occurring, but it’s not clear how effective this metric really is at identifying the most competitive areas of the race. A *position change text plot* was also demonstrated that identifies when each driver moved into a particular position; though tricky to read at first, the use of emphasis and the addition of final classification information might improve the informational value of this chart.

Use creatively, hopefully you’ll see how raw laptime data on its own can be put to a multitude of purposes!

Race History Charts

If you wanted a chart that summarised a race from the perspective of lap times, what sort of chart would you produce? In many racing circles, a *race history chart* is often used to present this data.

However, at first thought, it might seem as if a simple plot of the laptime recorded by each driver for each lap of the race might do the job. So let's construct just such a chart for a single driver in a single race using data from the *ergast* database.

Let's remind ourselves of what tables are available, and how to refer to them:

```
library(DBI)
ergastdb =dbConnect(RSQLite::SQLite(), './ergastdb13.sqlite')
tbs=dbGetQuery(ergastdb,'SELECT name FROM sqlite_master WHERE type = "table"')
tbs

##          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
```

We can access the laptime data for a particular race in a given year from the *races* table. I'm going to pick round 1 of 2012 for this example:

```
#Load in data relating to the lapTimes for a particular race

#When querying the database, we need to identify the raceId.
#This requires the year and either the round, circuitId, or name (of circuit)
raceId=dbGetQuery(ergastdb,'SELECT raceId FROM races WHERE year="2012" AND round="1"')

#There should be only a single result from this query,
# so we can index its value directly.
q=paste('SELECT * FROM lapTimes WHERE raceId=',raceId[[1]])
lapTimes=dbGetQuery(ergastdb,q)
#Note that we want the driverId as a factor rather than as a value
lapTimes$driverId=factor(lapTimes$driverId)

#We want to convert the time in milliseconds to time in seconds
#One way of doing this is to take the time in milliseconds column
lapTimes$rawtime = lapTimes$milliseconds/1000
```

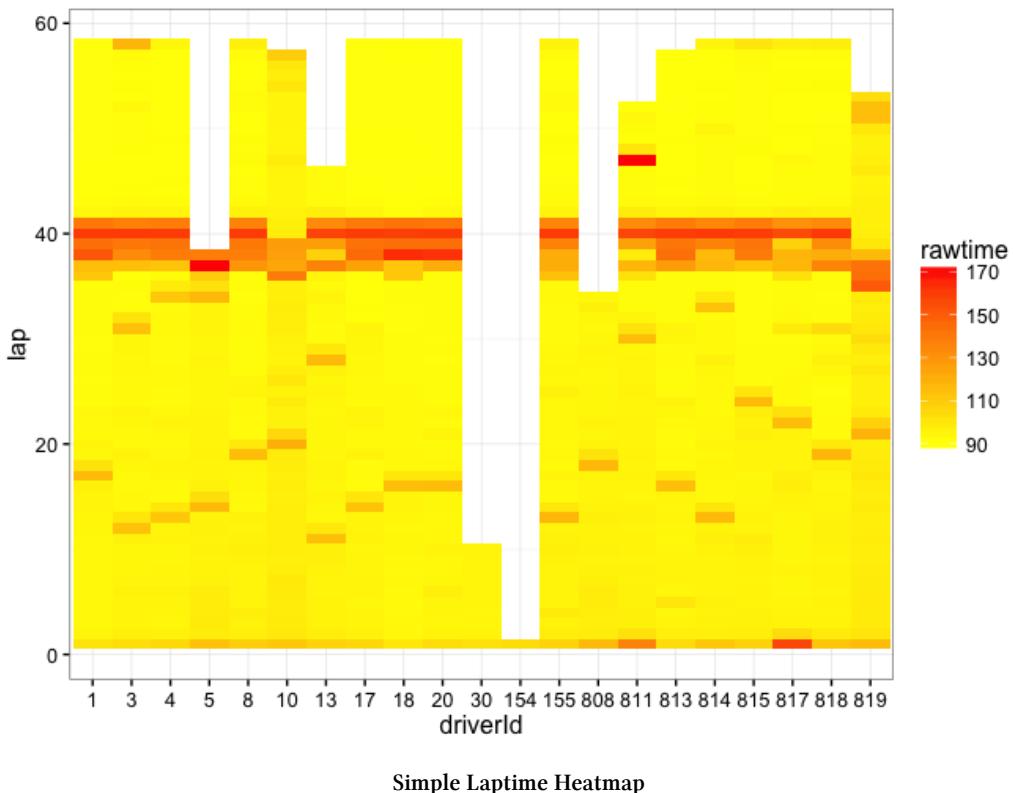
Getting a Feel for Laptimes Over the Course of a Race - Heatmaps

Heatmaps are widely used technique for visualising correlated features in a dataset. In a heatmap, a colour gradient is mapped onto a normalised range of values, allowing the values of multiple (normalised) variables to be compared on a colour basis.

The following sketch shows a simple heatmap representation of the laptimes for each driver over the course of a single race.

```
library(ggplot2)

g=ggplot(lapTimes)+ geom_tile(aes(x=driverId, y=lap, fill = rawtime))
g+scale_fill_gradient(low="yellow",high="red") + theme_bw()
```



Simple Laptime Heatmap

The yellow colour represents faster times and the red colour slower times. The orange/red banding around lap 40 - slow times for all the cars - suggests the presence of a safety car. The other (darker) red marks tend to show the laps on which each driver pitted.

The presence of the slow safety car laps skews the colour mappings, as perhaps do the slow laptimes from the first lap, starting as it did from a standing start. If we could remove the slow first lap, pit and safety car laps, to give a smaller range of laptimes, we would be able to distinguish between those times on a colour basis much more clearly.

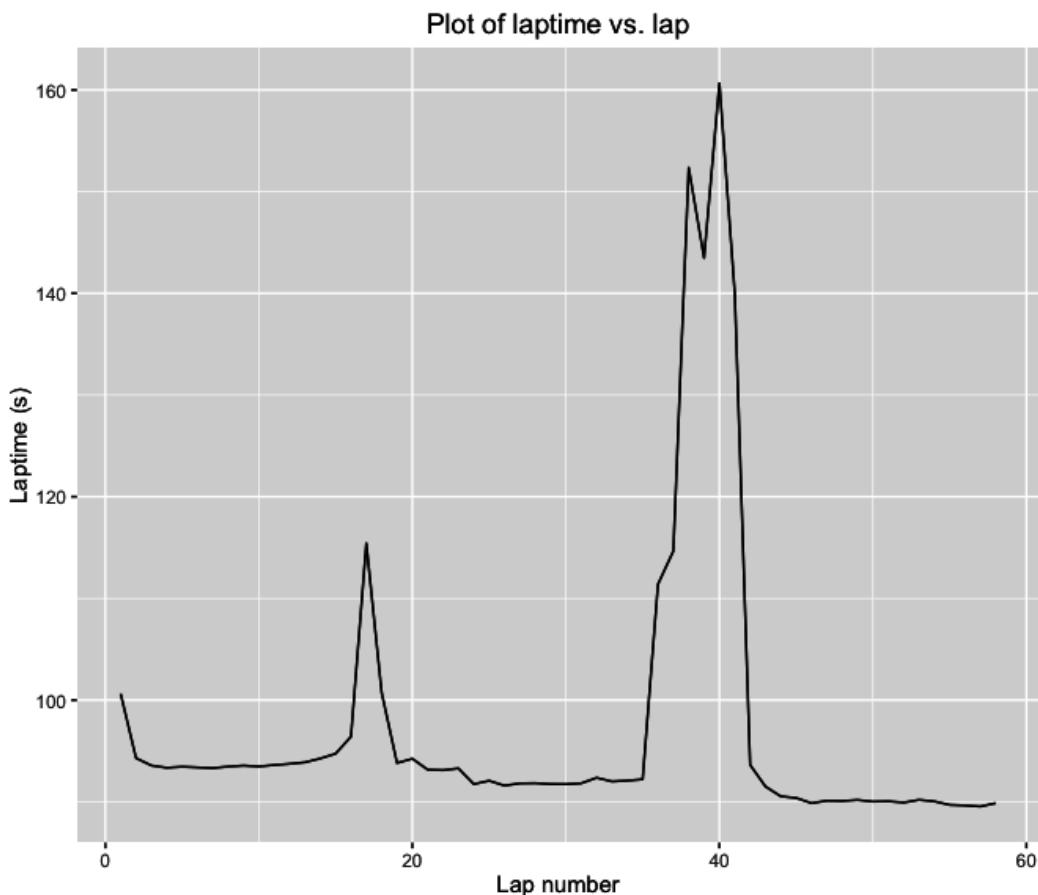
The heatmap provides an at a glance view of laptime trends across the course of the race, or at least, helps us easily spot the slow laps!), but it's quite hard to make comparisons of the actual times recorded and all but impossible to work out how drivers were placed relative to each other over the course of the race.

So starting from this data, and other data values, derived from it, how can we work towards the construction of a *race history chart* that does let us compare the race histories of all the drivers?

The Simple Laptimes Chart

What sort of chart do we get if we simply plot laptimes against lap number for a particular driver?

```
g=ggplot(subset(lapTimes,driverId==1))
g=g+geom_line(aes(x=lap,y=rawtime))
g+labs(title='Plot of laptimes vs. lap',x='Lap number', y='Laptimes (s)')
```



Line chart showing laptimes versus lap for a single driver

After a relatively slow first start (from a standing start, and with the confusion of the first few corners), the laptimes appear (at this scale) to show a slight downward trend

(decreasing/improving laptime as the fuel burns off), apart from when the driver pits around about lap 18 and something more significant happens around lap 40. In fact, there are likely to be two opposing factors influencing the laptime - a *decrease* in laptime associated with the car using fuel as it goes round the track, becoming lighter as it does so; and a likely *increase* in laptime associated with tyre wear. (In fact, the tyre model is likely to be more complicated than that, as the tyres may improve for a few laps - and hence tend to *reduce* laptime - as they come into their optimal operating window for temperature and pressure.)

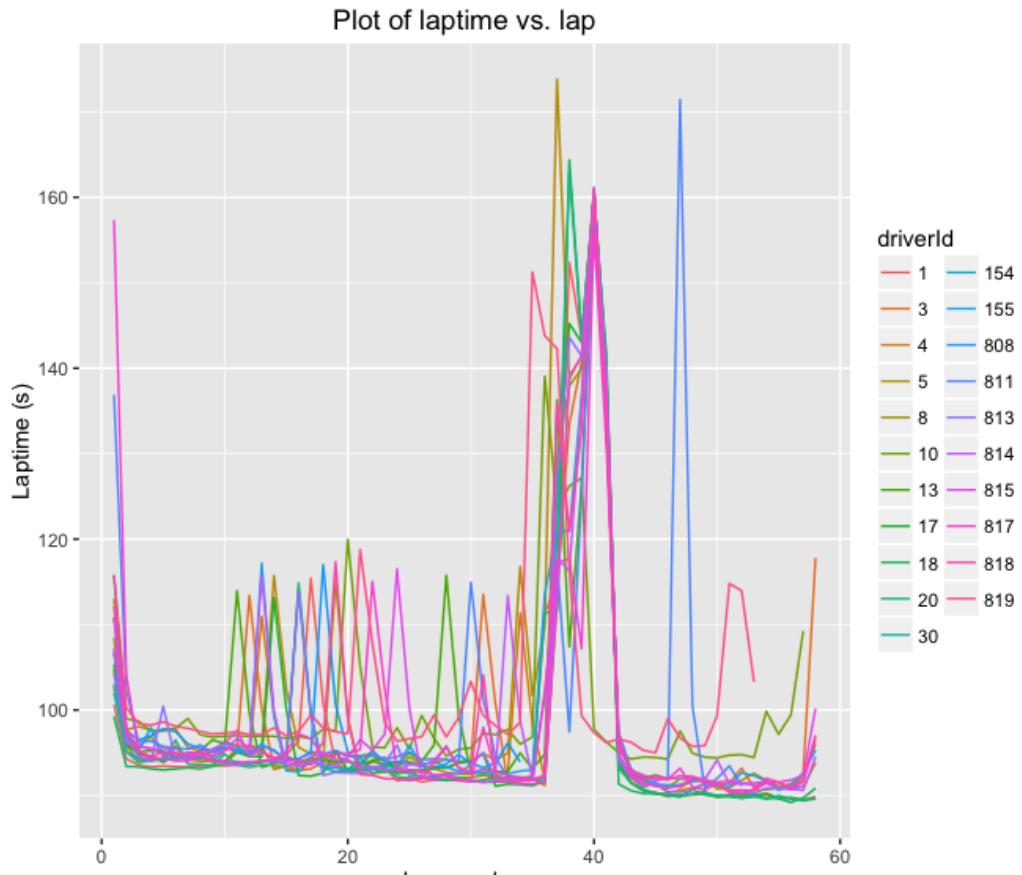
By inspection of the laptimes in general, we see that there are two slow laps associated with the pit event:

- the *in-lap*, from which the driver *enters* the pits, and
- the *out-lap*, which begins with the driver leaving the pits.

The chart also shows a prolonged period of slow laps around about lap 40, laps that are almost a minute slower per lap than the racing laps. This pattern in the laptime chart is typical of a safety car, rather than a sudden weather change, for example, because after the period of slow laps, the racing laps recover to (and even improve on by almost a second a lap) their previous times.

To plot the laptimes for each driver on a single plot, we might group each driver's times and then highlight them using colour.

```
g=ggplot(lapTimes)
g=g+geom_line(aes(x=lap,y=rawtime, group=driverId, colour=driverId))
g+labs(title='Plot of laptime vs. lap',x='Lap number', y='Laptime (s)')
```



Line chart of laptime versus lap, group and coloured by driver

This chart is very cluttered and doesn't really help us see the relative race positions of each driver. However, it does confirm that the really slow laps around lap 40 applied to all cars in the field and can definitely be accounted for by the presence of a safety car. That all cars are not affected equally by the safety car (that is, they do not all have the same, or even similar, lap times under the safety car, although the times are all slower than the racing laps) shows how the drivers can make up significant amounts of time under the safety car as they catch it up.

In this example, there is also a closing up of laptimes as the pack bunches up at the end of the safety car period.

Accumulated Laptimes

To explore the relative race positions of each driver, we might consider looking at the *accumulated race time* over laps for each driver.

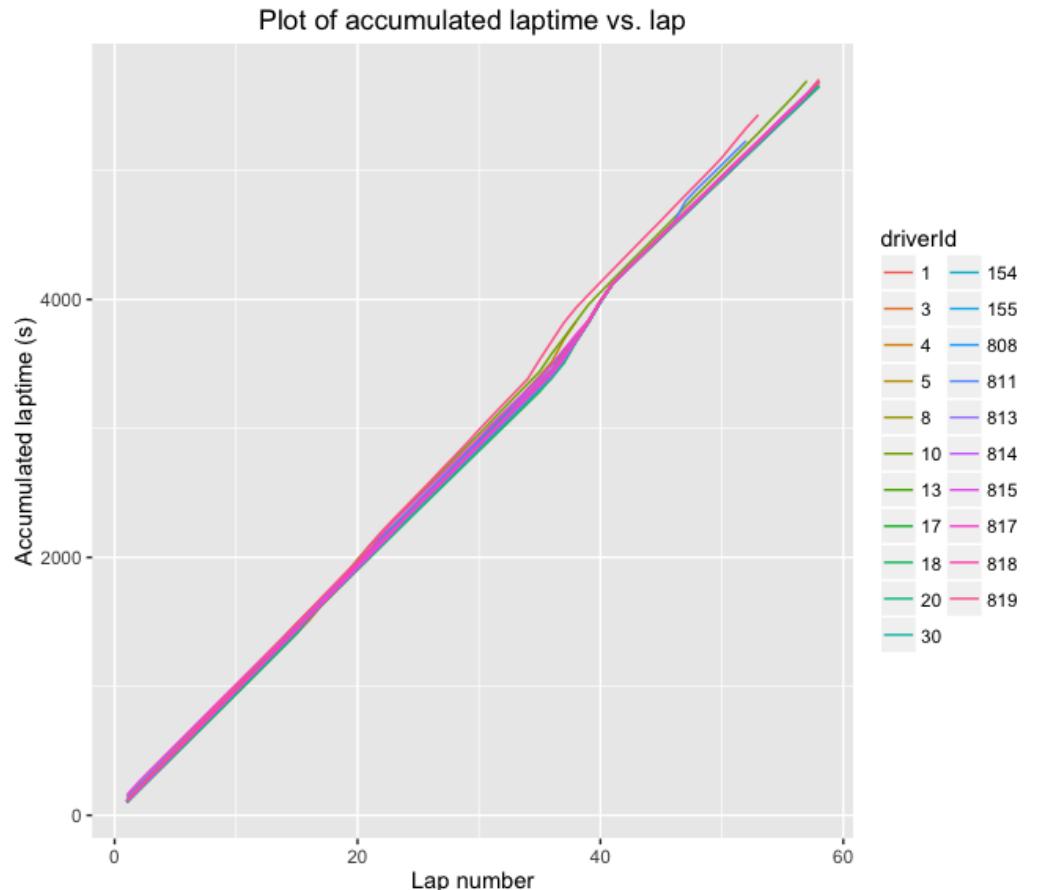
We already have the laptime data in lap order, so what we need to do now is sum the *rawtime* (that is, the raw laptime) for each driver over the increasing lap count. Using the split-apply-combine recipe, we run a cumulative sum of their laptimes for each driver and add it to a new column, *acctime*:

```
#install.packages("plyr")
library(plyr)
lapTimes=ddply(lapTimes, .(driverId), transform, acctime=cumsum(rawtime))
head(lapTimes,n=3)

##   raceId driverId lap position      time milliseconds rawtime acctime
## 1     860         1    1        2 1:40.622       100622 100.622 100.622
## 2     860         1    2        2 1:34.297       94297  94.297 194.919
## 3     860         1    3        2 1:33.566       93566  93.566 288.485
```

We can then plot this accumulated laptime against lap for each driver.

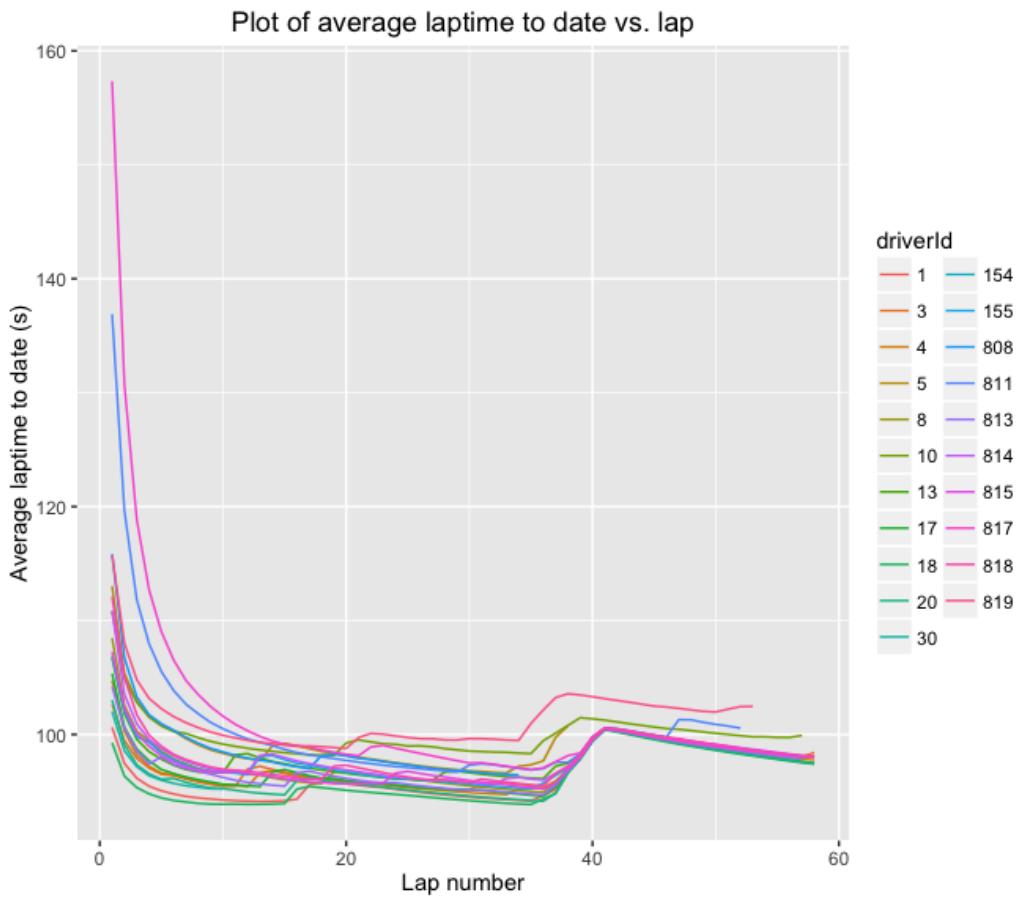
```
g=ggplot(lapTimes)
g=g+geom_line(aes(x=lap,y=acctime, group=driverId, colour=driverId))
g+labs(title='Plot of accumulated laptime vs. lap',
      x='Lap number', y='Accumulated laptime (s)')
```



Grouped line chart showing accumulated laptime over laps for each driver

Hmmm.. this chart is hard to read and doesn't seem to be that interesting. The lines for each driver do diverge, but it's hard to see the divergence clearly. One way we can try to introduce some separation between the lines is to take the average laptime to date. That is, divide the accumulated laptime by the lap number.

```
g=ggplot(lapTimes)
g=g+geom_line(aes(x=lap,y=acctime/loop, group=driverId, colour=driverId))
g+labs(title='Plot of average laptime to date vs. lap',x='Lap number', y='Average lap\ntime to date (s)')
```



Grouped line chart showing average lap time to date

Now perhaps we're starting to get somewhere: there's definitely some definition starting to appear between the different drivers and the lines aren't so noisy as the lines were in the simple laptime chart - the averaging process is smoothing out the curves somewhat. We're also starting to see something of a more marked decreasing trend in laptime over the course of each stint than in the simple *laptime vs lap number* chart.

Gap to Leader Charts

How about if we try to plot how far behind the lead driver each driver is at the end of each lap? *Silicon Alchemy* twists this idea slightly on the Lapalyzer website⁵² by charting the *session gap*, that is, the difference in accumulated laptime at the end of each lap between each driver and the accumulated laptime of the driver that provisionally won the race (that is, that finished the last lap in first position).

Before we work out how to plot that chart, let's consider something simpler: the accumulated time difference at the end of each lap between the race leader at the end of that lap and each individual driver. The lap count we will use us the lap number recorded for each driver, rather than the lead lap count at a particular accumulated race time. We can calculate that gap in several ways, but here's one way that gives us some additional information. It proceeds in two steps:

- firstly, calculate the *gap* - that is, the difference in accumulated laptime - between each driver and the driver ahead of him at the end of each lap;
- secondly, for each driver on each lap, calculate the sum of the gap times for the drivers ahead. This gives an overall *gap to leader*.

As before, we can use *ddply* to create new columns containing these values. Note that the *diff()* function finds the difference between consecutive values presented to it. For N drivers, there are thus N-1 gap values. We define the initial gap to be 0, and add this as the first gap value (that is, the gap for the driver in first position on that lap).

```
#Order laptimes by lap and position
lapTimes=lapTimes[order(lapTimes$lap,lapTimes$position),]
#For each lap, find the difference (the gap) between the accumulated
# laptimes of consecutively place drivers
lapTimes = ddply(lapTimes, .(lap), transform, gap=c(0,diff(acctime)) )
#Now calculate the summed gaps for all drivers ahead of a particular driver
lapTimes = ddply(lapTimes, .(lap), transform, leadergap=cumsum(gap) )

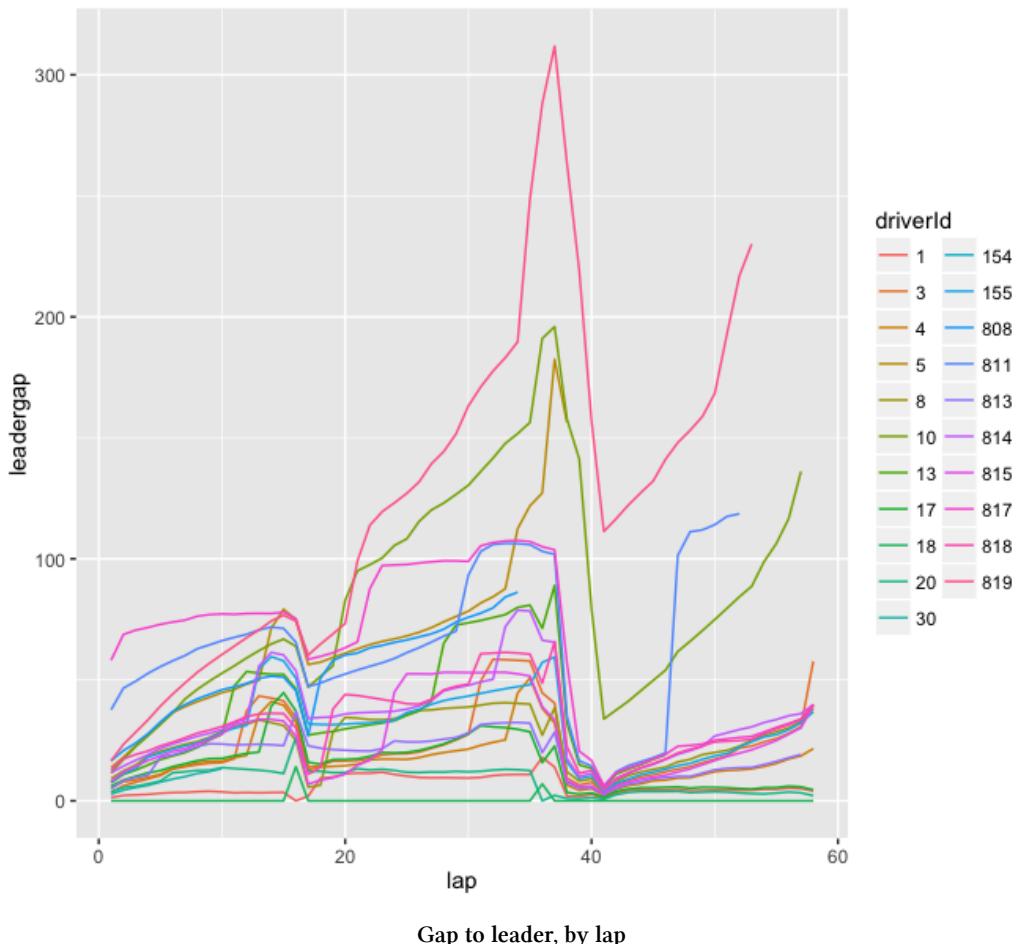
head(lapTimes[,c('driverId','lap','position','acctime','gap','leadergap')],n=3)
```

⁵²<http://www.lapalyzer.com/>

```
##   driverId lap position acctime   gap leadergap
## 1         18    1        1 99.264 0.000      0.000
## 2          1    1        2 100.622 1.358     1.358
## 3         30    1        3 102.002 1.380     2.738
```

We can then plot the gap to the current leader on a lap by lap basis:

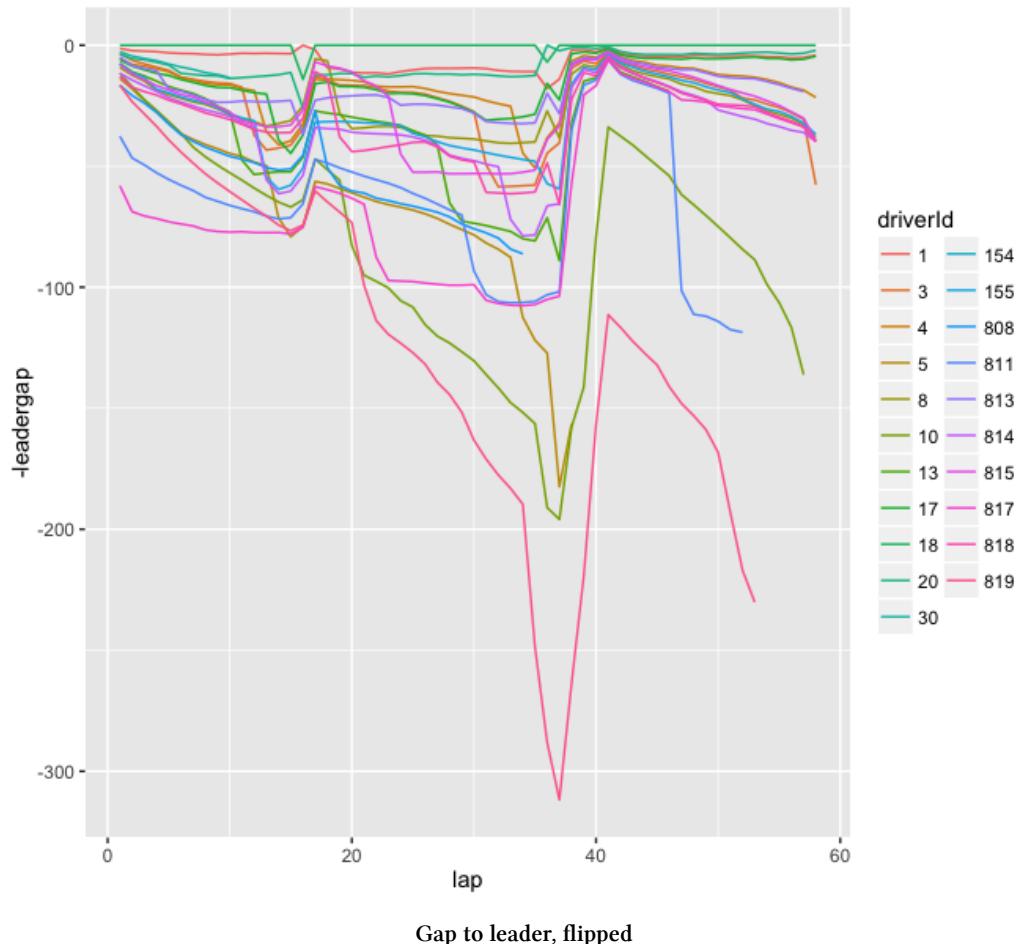
```
ggplot(lapTimes)+geom_line(aes(x=lap,y=leadergap,group=driverId, colour=driverId))
```



A more conventional way of presenting this sort of chart is to show drivers that are behind further down the chart. So let's flip the y-axis to show the amount of time *behind* the eventual

lap leader each driver was at the end of each lap, with the leader at the end of each lap aligned along the top of the chart:

```
ggplot(lapTimes)+geom_line(aes(x=lap,y=-leadergap,group=driverId, colour=driverId))
```



One useful feature of this sort of chart is that we can see whether there is a change in leadership (the line at the top of the chart) by noting changes to the line at the top of the chart, with gap 0, at the end of each lap.

The Lapalyzer Session Gap

Looking back at the *Lapalyzer* site, the session gap is subtly different to this - it measures the accumulated time difference at the end of each lap between each driver and the driver who eventually finished the last lap in first position. So how do we frame the data in this case? We need to find the cumulative time at the end of each lap for the eventual first place finisher of the race, and then the delta to this time from the accumulated time for each driver at the end of each corresponding lap.

In the corresponding chart, if the eventual first place finisher was not in lead position at any stage of the race, the lines would show a negative lead time (that is, a “positive” time) ahead of the eventual winner for the laps on which the eventual winner was not in the lead position.

Eventually: The Race History Chart

We’re now almost in a position to plot a *race history chart*. This sort of chart is widespread in motorsport, and is used to show how the lap times for each driver compare to that of the winner. Again, an averaging method is used, though this time based on the average laptime of the winner taken over the whole of the race.

The ‘race history time’ for each driver at the end of each lap is given as:

$$(\text{winner mean laptime}) * \text{laps} - (\text{accumulated lap time})$$

(If we are plotting the race history chart in real time, a so-called *online* algorithmic approach, we use the accumulated time of the lead car divided by the number of laps completed so far.)

James Beck, on the Intelligentf1 blog⁵³, describes the race history chart as follows:

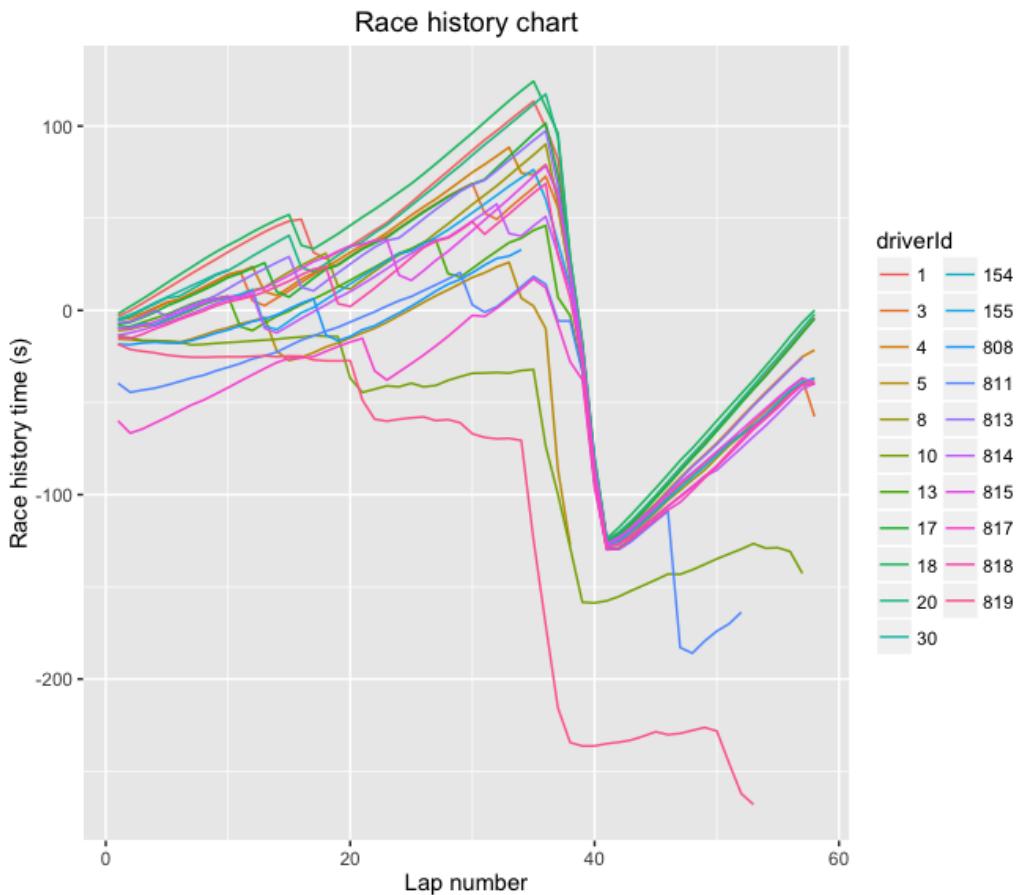
The horizontal axis is lap number, and the vertical axis shows the time each car is in front of (or behind) a reference average race time. This reference average time is often taken as the race time for the winner, such that the line representing the winner finishes at zero.... As this reference time is arbitrary, it can be set to different values to best view the race performances of different cars - this has the effect of shifting the lines up and down the graph.

⁵³<http://intelligentf1.wordpress.com/the-intelligentf1-model/>

```
#Let's fudge a way of identifying the winner.  
#The winner will be the person in position 1  
# at the end of the last (highest counted) lap  
winner = with( lapTimes,  
               lapTimes[lap==max(lapTimes$lap) & position==1, 'driverId'][[1]] )  
  
winnerMean = mean( lapTimes[ lapTimes$driverId==winner, 'rawtime' ] )  
lapTimes$raceHistory=winnerMean*lapTimes$lap - lapTimes$acctime
```

Let's see what that looks like:

```
g=ggplot(lapTimes)  
g=g+geom_line(aes(x=lap,y=raceHistory, group=driverId, colour=driverId))  
g=g+labs(title='Race history chart',x='Lap number', y='Race history time (s)')  
g
```



Race History Chart with unadjusted safety car period

In contrast to many typical race history charts, where the lines are predominantly below the origin line, the increased lap time introduced by the safety car has clouded the clarity of this chart in terms of helping us see the underlying pace of the cars.

Neutralising Safety Car Laps Within the Lap Chart

In the *intelligentF1* blog post for the same race Australian Grand Prix [2012]: Story from the Data⁵⁴, a more traditional chart can be seen in which the safety car laps have been “normalised” (should this be “neutralised”?), although no method for achieving this is described.

⁵⁴<http://intelligentf1.wordpress.com/2012/03/19/australian-grand-prix-story-from-the-data/>

As a first attempt at correcting for the safety car period, we can find the difference between the lap time of the leader from the start of the safety car period and the laptimes just prior to the safety car, then subtract this value from every driver's laptimes during the safety car period. This corrected laptimes can then be used for the calculation of the safety car adjusted race history chart.

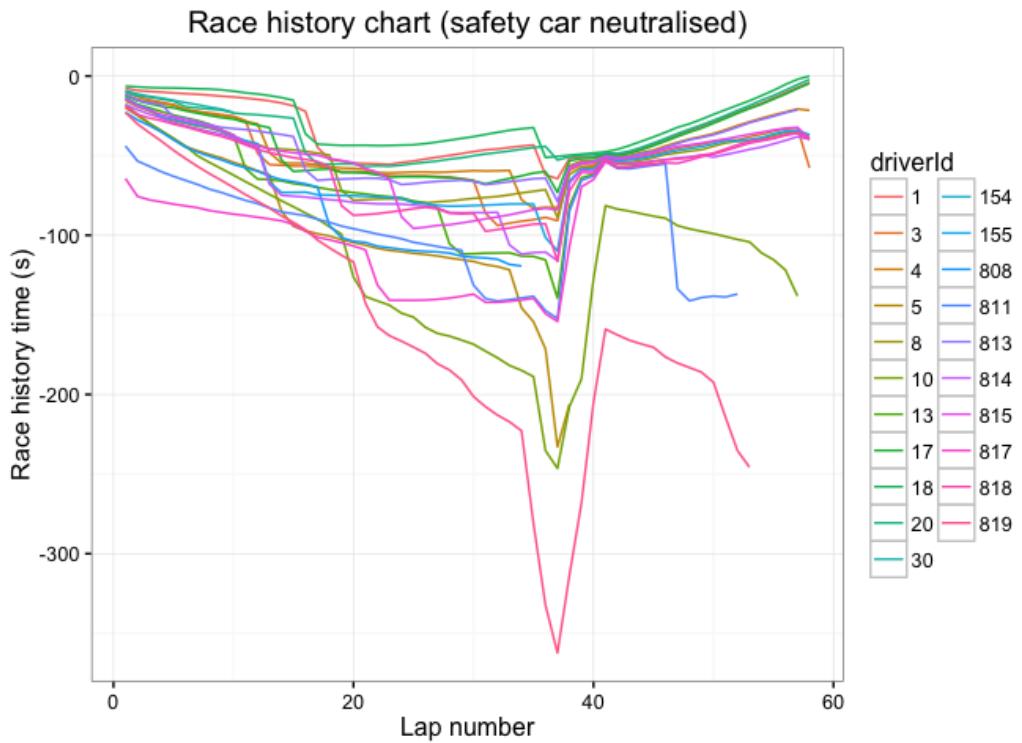
```
#Find the leader's laptimes just before the safety car
base=lapTimes[ lapTimes$position==1 & lapTimes$lap==35, 'rawtime' ]
#Create a neutralisation function that uses difference between leader's laptimes
#and leader's laptimes prior to the safety car, as a corrective delta
neutralise=function(x,y){if (x>35 & x<42) return(y - base) else return(0)}
corrective=data.frame(corr=mapply(neutralise,
                                    lapTimes[ lapTimes$position==1,'lap' ],
                                    lapTimes[ lapTimes$position==1,'rawtime' ]))

#Apply the correction term to every driver's laptimes
lapTimes=ddply(lapTimes,"driverId",mutate,corrt=rawtime-corrective[lap,])

#Calculate the corrected race history times
lapTimes=ddply(lapTimes, .(driverId), transform, corracctime=cumsum(corrt))
corlleaderMean = mean( lapTimes[ lapTimes$driverId==winner,'corrt' ] )
lapTimes$corraceHistory=corlleaderMean*lapTimes$lap - lapTimes$corracctime
```

If we now generate the race history chart with the safety car laps neutralised, we see the evolution of the race far more clearly.

```
g=ggplot(lapTimes)
g=g+geom_line(aes(x=lap,y=corraceHistory, group=driverId, colour=driverId))
g+labs(title='Race history chart (safety car neutralised)',
      x='Lap number', y='Race history time (s)') + theme_bw()
```



To finish off this chart, we should really use driver codes for the legend rather than `driverId`, perhaps even adding the labels to the chart directly.

Summary

Despite being a mainstay of race charts, with its ability to show the relative pace of cars and time distances between them across a race, as well as pit events and, on a careful reading, safety car periods (although these may sometimes be confused with slowdowns due to weather events), the race history chart can often appear cluttered and hard to read. Interactive chart approaches, in which a user can select which traces to highlight, and perhaps compare, often improve matters, and similar highlighting techniques can also be used for emphasis in static charts.

However, race history charts don't clearly show where cars are *on track* at any particular time, or whether a lapped car is causing some sort of obstacle between two otherwise racing

cars.

From Battlemaps to Track Position Maps

Battlemaps are a custom chart style designed to illustrate the competition between a particular driver and the cars in race positions immediately ahead or behind them at a particular stage in a race. Battlemaps can also be used to display the battle for a particular race position over the course of a race.

As well as revealing the gap to the car immediately ahead or behind in race position terms, *battlemap* displays also include cars on a different race lap (either backmarkers, or race leaders on laps ahead when considering lower placed positions) that have a track position in between that of a target vehicle and car in the race position either immediately or behind one position. (The aim here is to illustrate whether there are any off-lap vehicles that may interfere with any positional battles.)

As the graphic relies on information about track position as well as race position, we need to have access to data that reveals this information or that can be used to generate it. One way of obtaining this information is to derive it from the laptime data published as part of the *ergast* database.

Identifying Track Position From Accumulated Laptimes

Given a set of laptime data for a particular race, how can we identify track position information from it?

The first observation we might make is that a race track is a closed circuit; the second that the accumulated race time to date is the same for each driver, given that they all start the race at the same time. (The race clock is not started as each driver passes the start finish line - the race clock starts when the lights go green. To this extent, drivers lower placed on the grid serve a positional time penalty compared to cars further up grid. This effective time penalty corresponds to the time it takes a lower placed car to physically get as far up the track as the cars in the higher placed grid positions.)

Let's start by getting hold of all of the lap time data for a particular race:

```

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

#There should be only a single result from this query,
# so we can index its value directly.
q=paste('SELECT d.driverId,d.driverRef,d.code, l.lap,l.position,l.time,l.milliseconds
FROM drivers d JOIN lapTimes l JOIN races r
WHERE year="2012"
AND round="1"
AND r.raceId=l.raceId
AND d.driverId=l.driverId')
lapTimes=dbGetQuery(ergastdb,q)
#Note that we want the driverId as a factor rather than as a value
lapTimes$driverId=factor(lapTimes$driverId)

```

The laptimes are described as a time in milliseconds. At first glance it might appear to be more convenient to work in seconds, calculated by dividing the milliseconds time by 1000.

```

#We want to convert the time in milliseconds to time in seconds
#One way of doing this is to take the time in milliseconds column
lapTimes$rawtime = lapTimes$milliseconds/1000

```

However, in practice we see that when calculating differences between values represented in this way as floating point numbers, we get floating point errors in the smaller decimal positions. Working instead with milliseconds allows us to do integer arithmetic which incurs no such arithmetical precision errors.

In order to find the track position of a car, we first need to identify which leader's lap each driver is on and then use this as the basis for deciding whether a car is on the same lap - or a different one - compared with any car immediately ahead or behind on track. One way of doing this is on the basis of accumulated race time. If we order the laps by the accumulated race time, and flag whether or not a particular driver is the leader on particular lap, we can count the accumulated number of "lap leader" flags to give us the current lead lap count for each driver on each lap irrespective of how many laps a given driver has completed.

```

library(plyr)

#For each driver, calculate their accumulated race time at the end of each lap
lapTimes=ddply(lapTimes, .(driverId), transform,
               acctime=cumsum(milliseconds))

#Order the rows by accumulated lap time
lapTimes=arrange(lapTimes,acctime)
#This ordering need not necessarily respect the ordering by lap.

#Flag the leader of a given lap - this will be the first row in new leader lap block
lapTimes$leadlap= (lapTimes$position==1)
head(lapTimes[lapTimes$position<=3,c('driverRef','leadlap')],n=5)

##          driverRef leadlap
## 1          button    TRUE
## 2      hamilton   FALSE
## 3 michael_schumacher FALSE
## 22         button    TRUE
## 23      hamilton   FALSE

```

A Boolean TRUE value has numeric value 1, a Boolean FALSE numeric value 0.

```

#Calculate a rolling count of leader lap flags.
#Recall that the cars are ordered by accumulated race time.
#The accumulated count of leader flags is the lead lap number each driver is on
lapTimes$leadlap=cumsum(lapTimes$leadlap)
#The lapsbehind count is how many laps behind the leadlap a driver is
lapTimes$lapsbehind=lapTimes$leadlap-lapTimes$lap
head(lapTimes[lapTimes$position<=3,c('driverRef','leadlap')],n=6)

```

```

##           driverRef leadlap
## 1             button      1
## 2             hamilton    1
## 3 michael_schumacher    1
## 22            button      2
## 23            hamilton    2
## 24 michael_schumacher    2

```

Let's now calculate the track position for a given lead lap, where the leader in a given lap is in both race position and track position 1, the second car through the start/finish line is in track position 2 (irrespective of their race position), and so on. (In your mind's eye, you might imagine the cars passing the finish line to complete each lap, first the race leader, then either the car in second, or a lapped back marker, and so on.) Specifically, we group by lead lap *and then* accumulated race time within that lap, and assign track positions in incremental order.

```

lapTimes=arrange(lapTimes,leadlap,acctime)
lapTimes=ddply(lapTimes,.(leadlap),transform,
               trackpos=1:length(position))
head(lapTimes[lapTimes$leadlap==33,
             c('code','lap','position','acctime','leadlap','trackpos')],n=10)

```

```

##   code lap position acctime leadlap trackpos
## 616 BUT  33       1 3100735     33      1
## 617 HAM  33       2 3111538     33      2
## 618 VET  33       3 3113745     33      3
## 619 SEN  32      16 3115035     33      4
## 620 RIC  32      17 3115829     33      5
## 621 ALO  33       4 3125951     33      6
## 622 WEB  33       5 3131009     33      7
## 623 MAL  33       6 3133006     33      8
## 624 RAI  33       7 3141269     33      9
## 625 KOB  33       8 3147051     33     10

```

In this example, we see Timo Glock (GLO) has only completed 32 laps compared to 33 for the race leader and the majority of the field. *On track*, he is placed between Kobayashi (KOB) and Perez (PER).

Calculating DIFF and GAP times

As well as calculating track positions, we can also calculate various gap times, such as the standard GAP to leader and the +/- DIFF times to any car placed directly ahead or behind a particular car, either in race position terms *or* in terms of track position.

The GAP (time to leader) is calculated as the difference between the accumulated race time of the race leader at the end of a lap and the accumulated race time of driver when they complete the same race lap.

For a driver, d on lap l with laptime $t_{d,l}$, the accumulated race time $t_{d,N}$ for a driver d on lap N is then given as:

$$t_{d,N} = \sum_{l=1}^N t_{d,l}$$

For the leader on lap N , declare $d = L$ to give the accumulated race time for the leader at the end of lap N as $t_{L,N}$.

The GAP between a driver d after N laps and the leader at the end of lap N for $d \neq L$ is given as:

$$GAP_{d,N} = t_{d,N,GAP} = t_{d,N} - t_{L,N}$$

Alternatively, we can calculate the gap as the sum of differences between consecutively placed drivers between d and the race leader. The interval or DIFF between drivers in positions m and n at the end of N laps, where m is ahead of n (that is, $m < n$) is given as:

$$DIFF_{n,m,N} = t_{n,N} - t_{m,N}$$

The GAP between a driver in position P and the leader $L=1$ is then:

$$t_{P,N,GAP} = DIFF_{2,1,N} + D_{3,2,N} + \dots + DIFF_{P,P-1,N}$$

and where $GAP_{L,N} = t_{L,N,GAP} = 0$.

We can write this more succinctly as:

$$GAP_{P,N} = t_{P,N,GAP} = \sum_{p=2}^P DIFF_{p,p-1,N} = \sum_{p=2}^P (t_{p,N} - t_{p-1,N})$$

We can implement these calculations directly as follows:

```
#Order the drivers by lap and position
lapTimes=arrange(lapTimes,lap,position)
#Calculate the DIFF between each pair of consecutively placed cars
# at the end of each race lap
#Then calculate the GAP to the leader as the sum of DIFF times
lapTimes=ddply(lapTimes, .(lap), mutate,
               diff=c(0,diff(acctime)),
               gap=cumsum(diff) )
```

For completeness, we might also want to capture the DIFF to the car behind, which we shall represent as `chasediff`, further requiring that it is a negative quantity. That is, we have $CHASEDIFF_{q,r} = -DIFF_{r,q}$ for race positions $r > q$ (that is, q is ahead of r).

```
#Order the drivers by lap and reverse position
lapTimes=arrange(lapTimes,lap, -position)
#Calculate the DIFF between each pair of consecutively reverse placed cars at the end\
of each race lap
lapTimes=ddply(lapTimes, .(lap), mutate,
               chasediff=c(0,diff(acctime)) )

#Print an example
head(lapTimes[lapTimes$lap==35,c('code','lap','diff','chasediff')],n=5)
```

```
##      code lap  diff chasediff
## 658  PIC  35  92327       0
## 659  GLO  35  34462    -92327
## 660  KOV  35 14749     -34462
## 661  RIC  35  1281    -14749
## 662  SEN  35 25011    -1281
```

Typically, timing sheets do not show the GAP to the leader for cars other than those cars on the lead lap. Instead, they provide a count of the number of laps behind the race leader that a driver is. If required, we can generate this sort of view from our extended laptimes data set.

```

lapTimes$tradgap=as.character(lapTimes$gap)
#Define a function to find gap time to leader
# or number of laps behind if not on lead lap
lapsbehind=function(lap,leadlap,gap){
  if (lap==leadlap) return(gap)
  paste("LAP+",as.character(leadlap-lap),sep=' ')
}

lapTimes$tradgap=mapply(lapsbehind,lapTimes$lap,lapTimes$leadlap,lapTimes$gap)

#Print an example
lapTimes[lapTimes$lap==35,c('code','lap','leadlap','tradgap')]

##      code  lap leadlap tradgap
## 658  PIC   35     37  LAP+2
## 659  GLO   35     36  LAP+1
## 660  KOV   35     36  LAP+1
## 661  RIC   35     36  LAP+1
## 662  SEN   35     36  LAP+1
## 663  MAS   35     35  80864
## 664  DIR   35     35  78460
## 665  VER   35     35  60621
## 666  ROS   35     35  57772
## 667  PER   35     35  51594
## 668  ALO   35     35  50737
## 669  KOB   35     35  47972
## 670  RAI   35     35  39980
## 671  MAL   35     35  32088
## 672  WEB   35     35  28514
## 673  VET   35     35  12514
## 674  HAM   35     35  10890
## 675  BUT   35     35      0

```

Here we see that the drivers up to and including Massa (MAS) are on the lead lap, and as such an explicit time gap to the leader is reported. For Bruno Senna (SEN) and the lower placed drivers, they are one lap behind the leader, except for Charles Pic, who is two laps down.

Calculating the time between cars based on track position

To calculate the time difference to the car ahead on track (`car_ahead`) and the car behind on track (`car_behind`), we can simply calculate the differences between accumulated laptimes for appropriately ordered rows.

```

#Arrange the drivers in terms of increasing accumulated race time
lapTimes = arrange(lapTimes, acctime)
#For each car, calculate the DIFF time to the car immediately ahead on track
lapTimes$car_ahead=c(0,diff(lapTimes$acctime))
#Identify the code of the driver immediately ahead on track
lapTimes$code_ahead=c(NA,head(lapTimes$code,n=-1))
#Identify the race position of the driver immediately ahead on track
lapTimes$position_ahead=c(NA,head(lapTimes$position,n=-1))

#Now arrange the drivers in terms of decreasing accumulated race time
lapTimes = arrange(lapTimes, -acctime)
#For each car, calculate the DIFF time to the car immediately behind on track
lapTimes$car_behind=c(0,diff(lapTimes$acctime))
#Identify the code of the driver immediately behind on track
lapTimes$code_behind=c(NA,head(lapTimes$code,n=-1))
#Identify the race position of the driver immediately behind on track
lapTimes$position_behind=c(NA,head(lapTimes$position,n=-1))

#Put the lapTimes dataframe back to increasing accumulated race time order.
lapTimes = arrange(lapTimes, acctime)

```

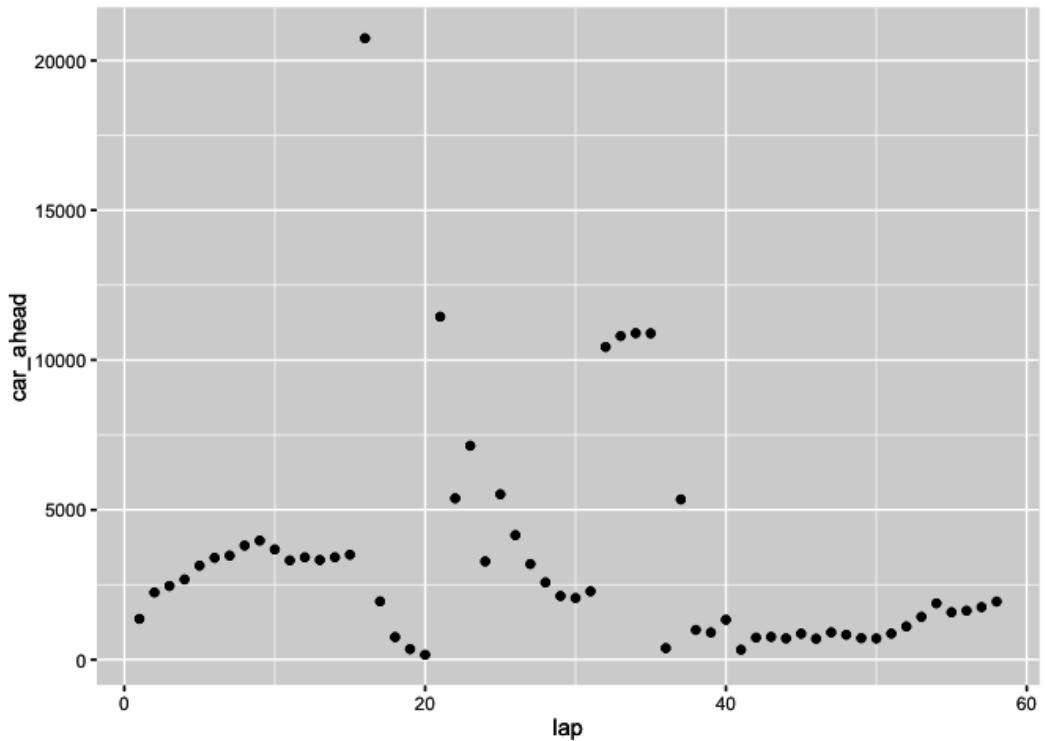
Notice how the `diff()` function finds the difference between column values on consecutive rows working down the column. To find the gap to the car ahead, we sort on *increasing* accumulated race time and then apply the `diff()` function. To find the gap to the car behind, we reverse the order, sorting on *decreasing* accumulated lap time, before applying the `diff()` function.

Having calculated the time to car ahead - or the car behind - on track, we can use a simple scatter plot to show the time in milliseconds to the car ahead, for each lap .

```

library(ggplot2)
#Display the car one position ahead of a named driver
g=ggplot(lapTimes[lapTimes['code']=='HAM',])
g+geom_point(aes(x=lap,y=car_ahead))

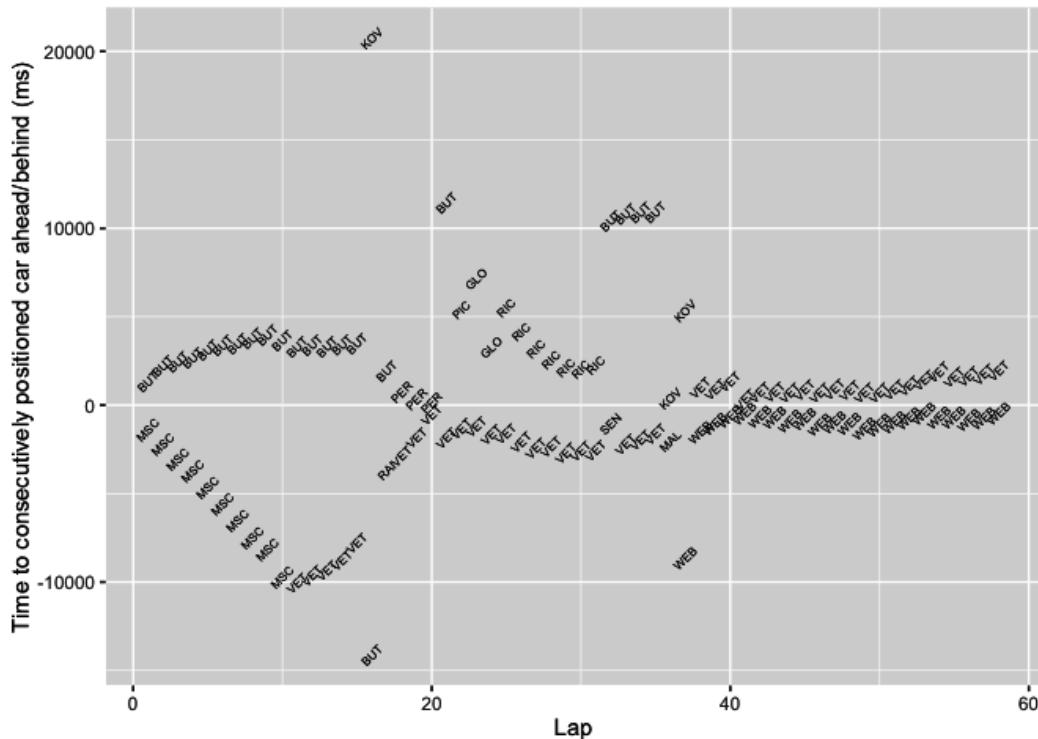
```



For a selected driver, chart the time to the car ahead

If instead we use a text plot, we can identify which driver in particular was in the car ahead or behind on track.

```
#Display the positioned and identity of cars immediately ahead of and behind
#the named driver
g=g+geom_text(aes(x=lap,y=car_ahead,label=code_ahead),angle=45,size=2)
g=g+geom_text(aes(x=lap,y=car_behind,label=code_behind),angle=45,size=2)
g+xlab('Lap')+ylab('Time to consecutively positioned car ahead/behind (ms)')
```



For a selected driver, identify which driver is ahead or behind on track, and by how much time, on each lap

Here we see that Hamilton draws away from Michael Schumacher at a constant rate over the first 10 laps of the race, dropping behind Jenson Button over that same period, then keeps pace with him between laps 10 and 15.

Note that the driver codes specified refer to the driver immediately ahead or behind on the track, irrespective of whether they are on the same lap. The chart would perhaps be more informative if we could identify whether the car immediately ahead or behind is actually on the same *racing lap* as our selected driver.

One way to approach this is to generate new columns that identify the driver immediately ahead or behind each car in terms of race position, rather than track position. This new information will allow us to test whether the car ahead or behind on track is in a battle for position with the selected driver.

In the battlemap, we shall refer to the driver about whom (that is, *for whom*) the map is drawn as the *target driver*.

```

#Sort the laptimes by increasing lap and position
lapTimes = arrange(lapTimes, lap, position)
#Find the code of the driver ahead
lapTimes = ddply(lapTimes, .(lap), transform,
                 code_raceahead=c(NA,head(code,n=-1)))
#Reverse the sort order to decreasing lap and position
lapTimes = arrange(lapTimes, -lap, -position)
#Find the code of the driver behind
lapTimes = ddply(lapTimes, .(lap), transform,
                 code_racebehind=c(NA,head(code,n=-1)))
#Put the sort order back to increasing accumulated time
lapTimes = arrange(lapTimes, acctime)

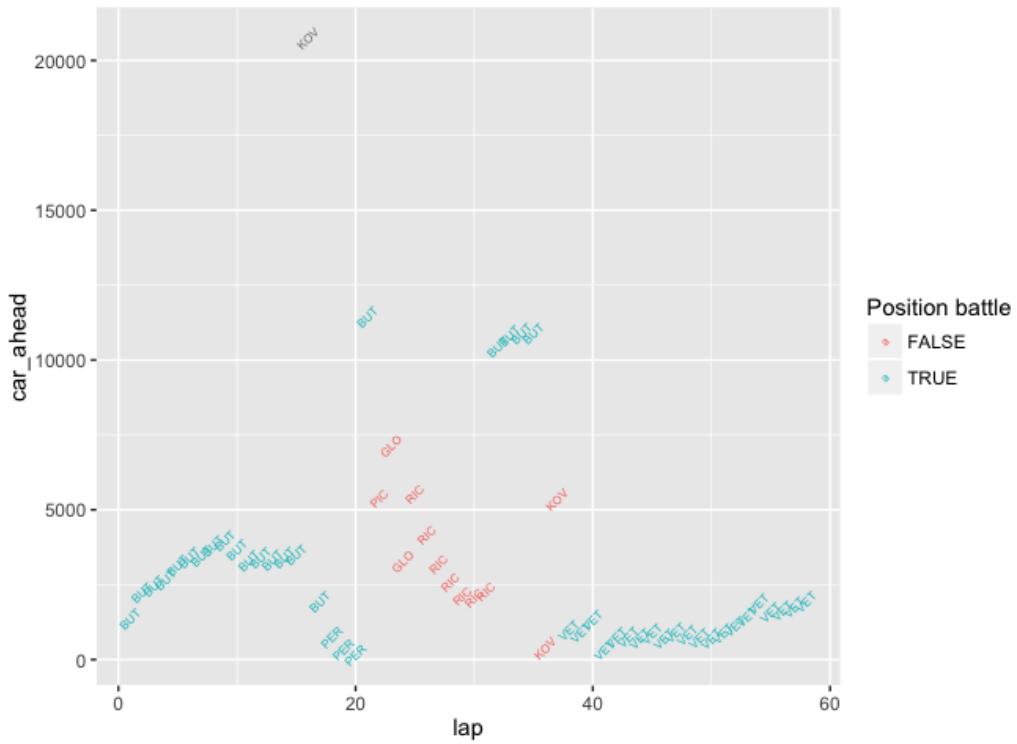
```

The test is one simply of equality: *is the driver one place ahead on track the driver in one place ahead in terms of race position?* The answer to this test allows us to visually distinguish between whether there is a battle for position going on with the car directly ahead on track:

```

battlesketch1=function(driverCode){
  g=ggplot(lapTimes[lapTimes['code']==driverCode,])
  g=g+geom_text(aes(x=lap,
                     y=car_ahead,
                     label=code_ahead,
                     #Test whether we are in a direct battle with the car ahead
                     col=factor(code_ahead==code_raceahead)),
                 angle=45,size=2)
  g+guides(col=guide_legend(title="Position battle"))
}
battlesketch1('HAM')

```



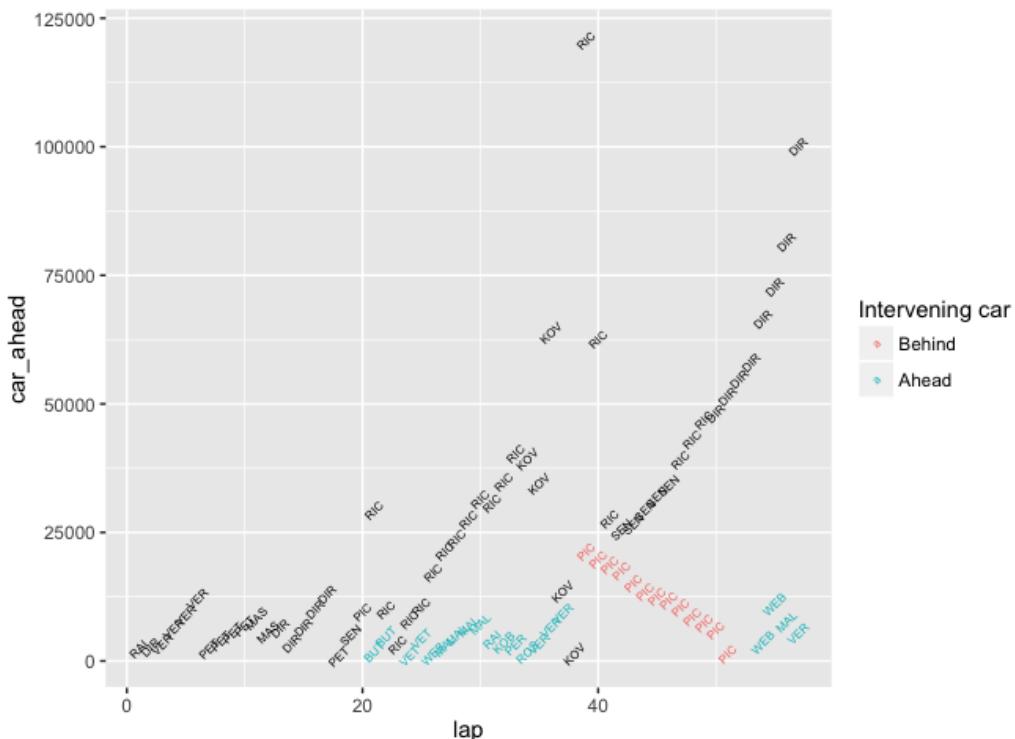
plot of chunk battlesketchHAM

A useful side effect here is that if `code_raceahead` is undefined (because the selected driver is in the lead at the start of a particular lap), the `code_ahead==code_raceahead` test is also undefined, no colour is set, and the text label color defaults to grey.

It would be useful to further refine the chart so that it additionally shows the driver immediately ahead (or behind) in terms of *race* position if the car ahead on track is not the car immediately ahead in terms of position.

We can achieve this by adding another layer:

```
battlemap_ahead=function(driverCode){  
  g=ggplot(lapTimes[lapTimes['code']==driverCode,])  
  #Plot the offlap cars that aren't directly being raced  
  g=g+geom_text(data=lapTimes[(lapTimes['code']==driverCode)  
    & (lapTimes['code_ahead']!=lapTimes['code_raceahead'])],  
  ],  
    aes(x=lap,  
        y=car_ahead,  
        label=code_ahead,  
        col=factor(position_ahead<position)),  
    angle=45,size=2)  
  #Plot the cars being raced directly  
  g=g+geom_text(aes(x=lap,  
    y=diff,  
    label=code_raceahead),  
    angle=45,size=2)  
  g=g+scale_color_discrete(labels=c("Behind","Ahead"))  
  g+guides(col=guide_legend(title="Intervening car"))  
}  
battlemap_ahead('GLO')
```



A first attempt at a battle map, showing cars in positions immediately ahead/behind on track, as well as in race position terms

In this case, cars on the same lap are coloured black, and cars in a track position that is out of race position is coloured either aqua (if it is in a race position ahead of the selected car) or orange if it is on at least one lap behind. From the chart, we notice how the target driver, Timo Glock, falls behind the car he is racing time and time again (the waves of activity that go up and to the right). The only car he gains on over a series of laps is that of Charles Pic, from about lap 39 onwards. Every so often, cars at least one lap ahead (coloured aqua), are in the space ahead between Timo Glock and the car in the race position ahead of him.

Let's now try to put the pieces together in a full battle map showing the state of the race immediately ahead of a particular driver in track position *and* race position terms, as well as immediately behind them, throughout the course of a race. We'll also add in a guide that identifies the DRS (drag reduction system) range of one second (1000ms).

We can further generalise the battle map charter so that it can plot just the battles with the cars ahead or behind. In the following example, we plot just the battle ahead.

```

dirattr=function(attr,dir='ahead') paste(attr,dir,sep='')

#We shall find it convenient later on to split out the initial data selection
battlemap_df_driverCode=function(driverCode){
  lapTimes[lapTimes['code']==driverCode,]
}

battlemap_core_chart=function(df,g,dir='ahead'){
  car_X=dirattr('car_',dir)
  code_X=dirattr('code_',dir)
  factor_X=paste('factor(position_,dir,<position)',sep=' ')
  code_race_X=dirattr('code_race',dir)
  if (dir=='ahead') diff_X='diff' else diff_X='chasediff'

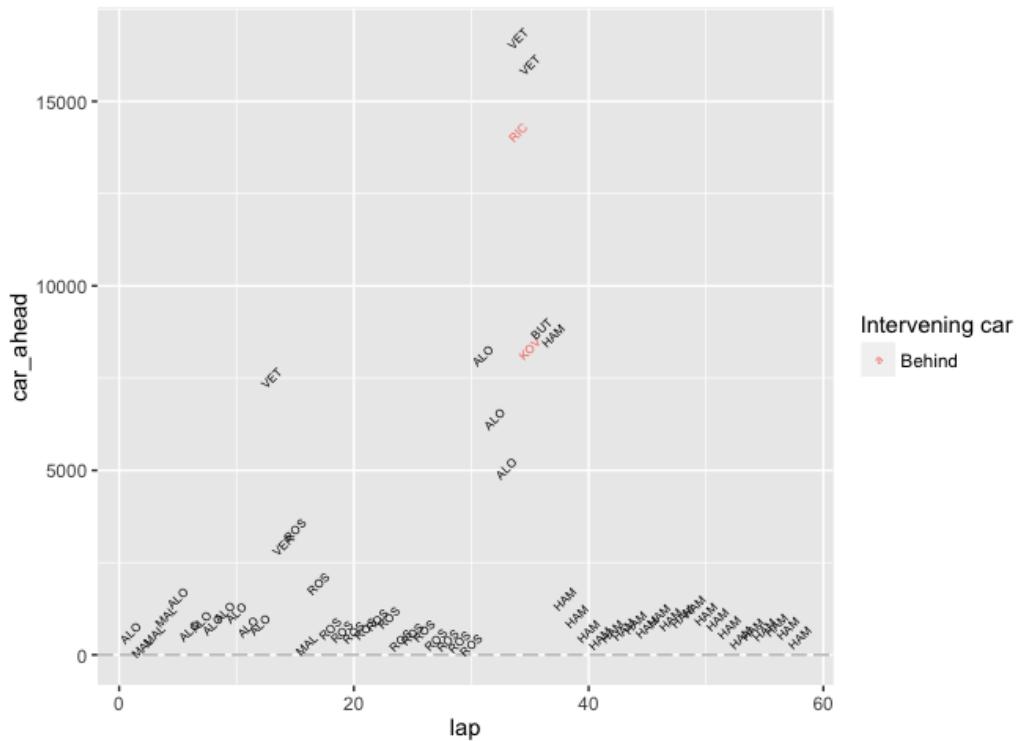
  if (dir=="ahead") drs=1 else drs=-1
  g=g+geom_hline(aes_string(yintercept=drs),linetype=5,col='grey')

  #Plot the offlap cars that aren't directly being raced
  g=g+geom_text(data=df[df[dirattr('code_',dir)]!=df[dirattr('code_race',dir)],],
    aes_string(x='1ap',
      y=car_X,
      label=code_X,
      col=factor_X),
    angle=45,size=2)
  #Plot the cars being raced directly
  g=g+geom_text(data=df,
    aes_string(x='1ap',
      y=diff_X,
      label=code_race_X),
    angle=45,size=2)
  g=g+scale_color_discrete(labels=c("Behind","Ahead"))
  g+guides(col=guide_legend(title="Intervening car"))

}

battle_WEB=battlemap_df_driverCode("WEB")
g=battlemap_core_chart(battle_WEB,ggplot(),'ahead')
g

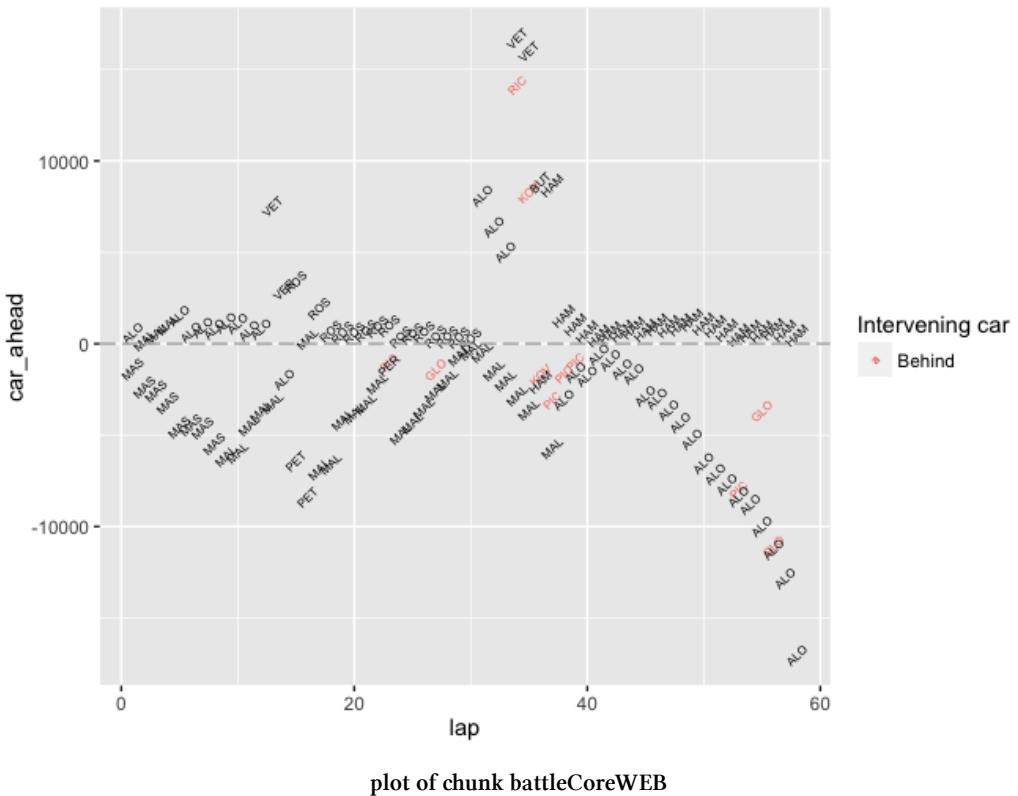
```



A first attempt at a complete battlemap.

And then we can add in any threats that are coming up from behind:

```
battlemap_core_chart(battle_WEB,g,dir='behind')
```



Here we see how at the start of the race Mark Webber kept pace with Alonso, albeit around about a second behind, at the same time as he drew away from Massa. In the last third of the race, he was closely battling with Hamilton whilst drawing away from Alonso.

Battles for a particular position

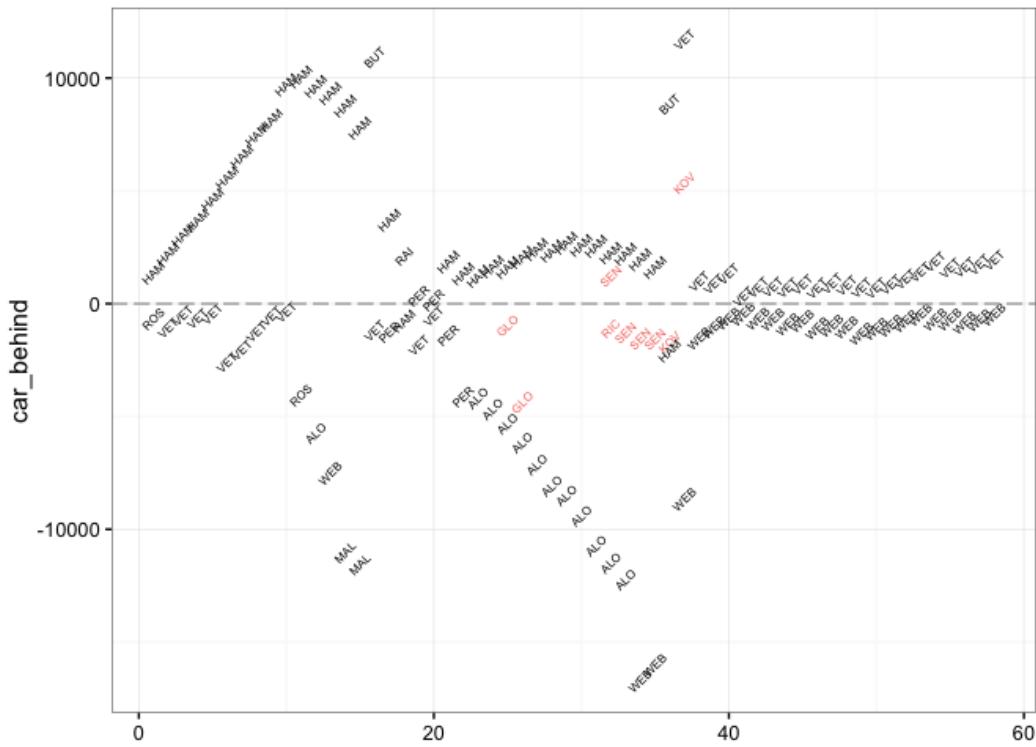
As well as charting the battles in the vicinity of a particular driver, we can also chart the battle in the context of a particular race position. We can reuse the chart elements and simply need to redefine the filtered dataset we are charting.

For example, if we filter the dataset to just get the data for the car in third position at the end of each lap, we can then generate a battle map of this data.

```
battlemapper_df_position=function(position){
  lapTimes[lapTimes['position']==position,]
}

battleForThird=battlemapper_df_position(3)

g=battlemapper_core_chart(battleForThird,ggplot(),dir='behind')+xlab(NULL)+theme_bw()
g=battlemapper_core_chart(battleForThird,g,'ahead')+guides(col=FALSE)
g
```



A position battle chart showing the fight for third in the course of a single race

In this case we see how in the opening laps of the race, the battle for third was coming from behind, with Vettel challenged for position from fourth, as the second placed driver (Lewis Hamilton) pulled away. In the middle third of the race, the car in third kept pace with 2nd placed Hamilton but pulled away from fourth placed Alonso. And in the last third of the race, the car in third is battling hard with Vettel ahead and defending hard against Webber behind.

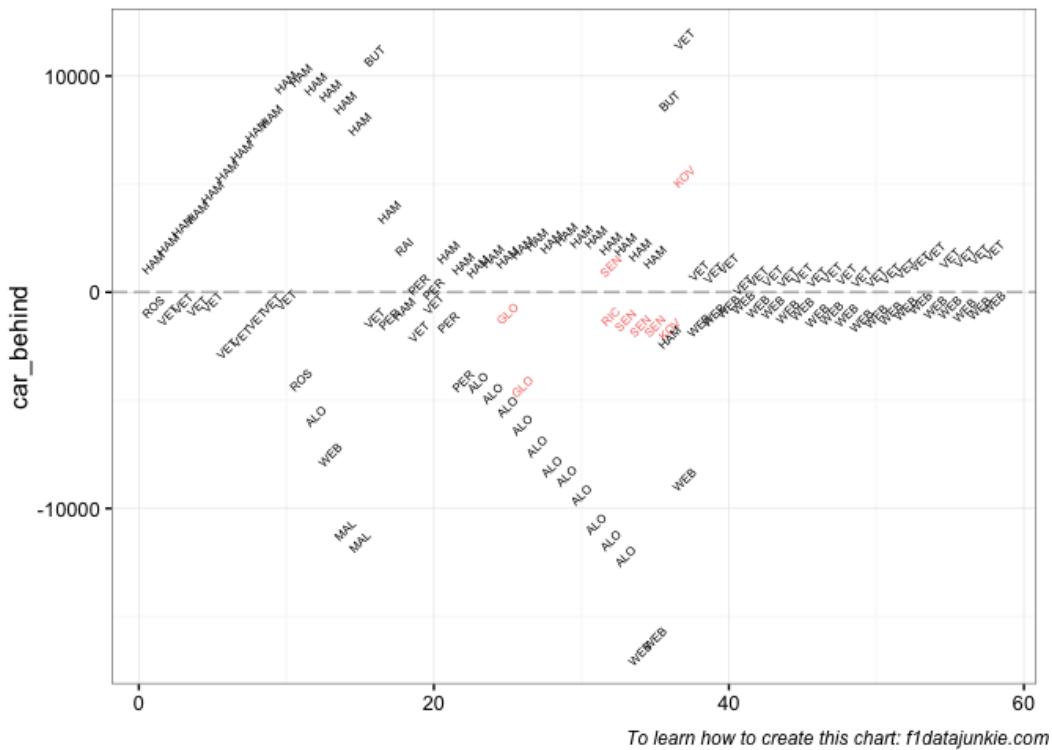
One thing this chart does not show is which driver was in third position on each lap. We might naturally think to add a layer on to the chart that displays the driver in the position we are charting around along the x-axis (that is, at $y=0$) but this is likely to lead to a very cluttered chart. It may make more sense to try to align this information in a marginal area at the bottom of the chart.

Finally, we observe that another sort of battle we might wish to depict is a battle between two particular drivers. However, with just two drivers to compare, we would need to think carefully about how to display this information - would one of the driver's details be aligned along the horizontal x-axis, and the gap to the other driver charted accordingly? Or should the x-axis represent the mid-point of the difference between the two drivers?

Adding Footnotes to a Chart

At times, we may wish to annotate a chart with a footnote, such as a copyright notice or further information notice. One way of doing this is to use a `textGrob()` object from the `grid` library, positioning it with a `grid.arrange()` call from the `gridExtra` package.

```
library(grid)
library(gridExtra)
grid.arrange(g, nrow=1,
             #top="My title",
             bottom = textGrob("To learn how to create this chart: f1datajunkie.com",
                               gp = gpar(fontface=3, fontsize=9),
                               hjust=1, x=1))
```



plot of chunk overprintCreditExample

Annotating charts in this way makes a graphic a standalone item that can be shared as an image file whilst still retaining things like attribution information embedded within it.

Generating Track Position Maps

A lot of the data manipulation work we have to do when piecing together the battlemaps relates to identifying the gap *on track* between different cars. We can use this derived data to plot a chart that shows how the cars are arranged on track according to each lead lap.

For each lap, we find the accumulated race time of the leader and use that as a reference point to offset the time of every other car. We then need to find the ‘on track’ time distance between each car and the leader.

```

#Ensure that the laptimes are arranged according to elapsed time
lapTimes=arrange(lapTimes, acctime)
#Find the accumulated race time for the race leader at the start of each lap
lapTimes=ddply(lapTimes, .(leadlap), transform, lstart=min(acctime))

#Find the on-track gap to leader
lapTimes['trackdiff']=lapTimes['acctime']-lapTimes['lstart']

```

We can also identify the time distance between the leader of the current lap and the leader of the next lap - this gives us an estimate of the “time width” of the lap.

```

#Construct a dataframe that contains the difference between the
#leader accumulated laptime on current lap and next lap
#i.e. how far behind current lap leader is next-lap leader?
11=data.frame(t=diff(lapTimes[lapTimes['position']==1,'acctime']))
#Generate a de facto lap count
11['n']=1:nrow(11)
#Grab the code of the lap leader on the next lap
11['c']=lapTimes[lapTimes['position']==1 & lapTimes['lap']>1, 'code']

```

For each lead lap, we can now plot the distance between each car and the lap leader, as well as showing the position of the leader of the next lap. We identify lapped cars by overplotting the chart with an indication of how many laps behind the lap leader each lapped car is. Additional overplotting can be used to highlight a specified driver, in this case, Alonso.

```

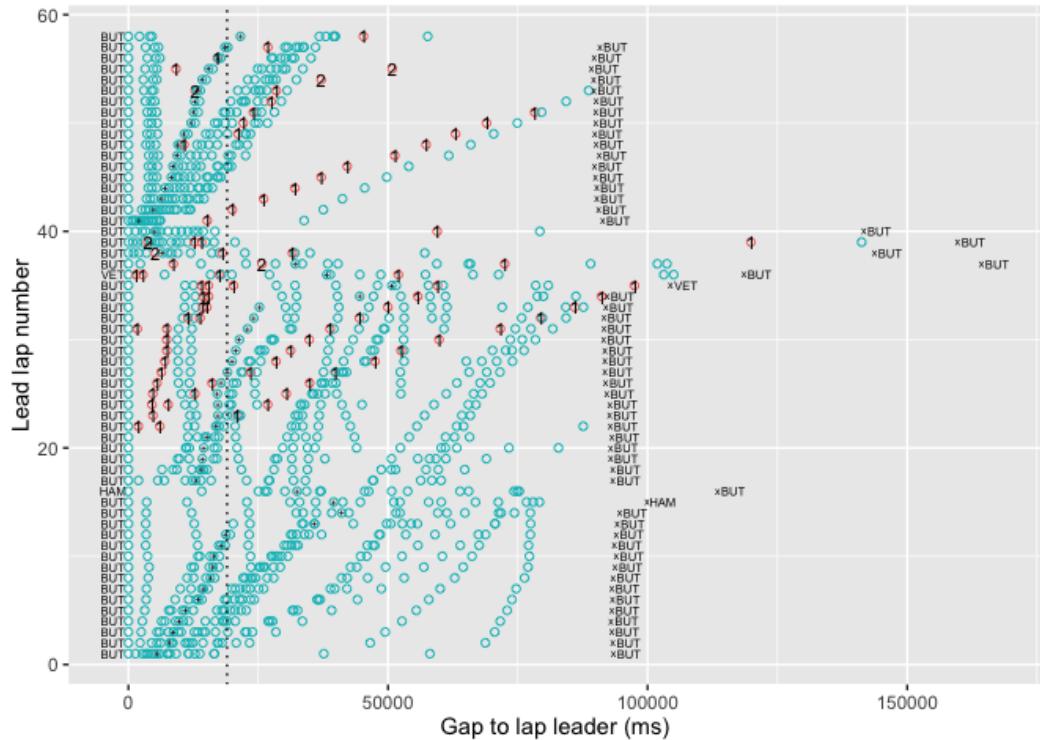
#Plot the on-track gap to leader versus leader lap
g = ggplot(lapTimes)
#Plot a marginal item on the left hand side naming the lap leader for each lap
g=g+geom_text(data=lapTimes[lapTimes['position']==1,],
               aes(x=-3000,y=leadlap,label=code),size=2)
#Plot markers for each car using colour to identify cars on lead lap
g = g + geom_point(aes(x=trackdiff,y=leadlap,col=(lap==leadlap)), pch=1)
#Track the progress of a particular driver
g = g + geom_point(data=lapTimes[lapTimes['driverRef']=='alonso',],
                    aes(x=trackdiff,y=leadlap), pch='+')
#Display a count of how many laps behind lapped drivers are
g = g + geom_text(data=lapTimes[lapTimes['lapsbehind']>0,],
                   aes(x=trackdiff,y=leadlap, label=lapsbehind),size=3)
#Show the position of the next lap leader along with an offset name label
g = g + geom_point(data=11,aes(x=t, y=n), pch='x')

```

```

g = g + geom_text(data=ll,aes(x=t+3000, y=n,label=c), size=2)
#Set an elapsed time amrker (19000ms) to indicate pit stop loss
g = g + geom_vline(aes(xintercept=19000), linetype=3)
g+ guides(colour=FALSE)+xlab('Gap to lap leader (ms)')+ylab('Lead lap number')

```



A track position map showing how cars are separated on track for each lead lap

The dotted line can be set at a value around the pit loss time to indicate where a lap leader may return to the race after pitting.

Battles can be identified through the close proximity of two or more drivers within a lap, across several laps. The "next-lap-leader"™ time at the far right shows how close the leader on the next lead lap is to the backmarker (on track) on the current lead lap.

By highlighting one or more specific drivers, we can compare how their races evolved, perhaps highlighting different strategies used within a race that eventually bring the drivers into a close competitive battle in the last few laps of a race.

Summary

In this chapter, we have demonstrated how to process the laptime data in order to identify track position as well as race position, generating standard GAP and DIFF values between cars, as well as the DIFF to the car in the race position behind, and the time to the cars ahead and behind *on track*.

A new type of chart, the *battle map*, was introduced that shows how close a selected driver is to the cars immediately ahead of, and behind them, on each lap, in terms of both race position and track position. The battle map can be used to show how a race evolved from the perspective of a particular driver.

A variant of the battle map was also introduced that showed the battle that took place around a particular race position. This still needs further work, however, in terms of identifying which driver was in the position being fought over on each lap.

Finally, the gap between cars on track was used as the basis of a second novel chart type, a *track position map*. This shows, for each lead lap, the distribution of other cars on track and can be used to give an overview of how the cars were separated *on track* throughout the race.

Pit Stop Analysis

With the F1 race regulations such as they are in 2015, cars are required to run two tyre compounds during a dry race. This means that drivers are forced to pit at least once during a race in order to change tyres. In addition, at the request of Formula One Administration, Pirelli have experimented in recent years with various tyre compounds in an attempt to add elements of “tyre jeopardy” to the race spectacle.

Whenever a car comes in to the pits, the lap time goes up. Associated with each pit stop there is a *pit loss time* compared to a flying lap time that accrues as a result of the car having to slow down as it makes its way through the pit lane. There are also time losses associated with slowing down to enter the pit lane on an in-lap, and time spent getting back up to speed and tyre temperature on the out-lap. These time losses, or costs, are all in addition to any time the car spends as it is stationary, along with the milliseconds spent actually entering and exiting a particular pit box.

At the current time, the FIA do not publicly publish the time a car is stationary, or explicit data about the reason a car has entered the pit lane. This means that we do not necessarily know whether a car has stopped for a procedural tyre change pit stop, or as part of a penalty such as a drive-through penalty or a stop/go penalty.

However, sources such as the *ergast* API do publish some information about the time each car spends within the pit lane each time it enters the pits. In this chapter, we will explore various ways of visualising pit behaviour and comparing the relative effectiveness of the teams at pitting. We will also explore several ways of sorting and reshaping the pit stop data as part of the process of generating charts that are capable of highlighting meaningful similarities and differences in both individual and grouped pit stop times.

Pit Stop Data

Information about pit stop times can be found in the *pitStops* table of the *ergast* database and are also published as an official FIA timing sheet.

The *pitStops* table provides information on a race by race basis of includes the driver involved with a stop (`driverRef` and `driverId`), the time spent in the pit lane in seconds (`duration`) and as milliseconds (`milliseconds`), the pit number for that driver (that is, a count of the

number of stops to date, including the current stop) and the time of day the pit event happened.

One way of using the pit stop data is to recognise the fact that a particular driver pitted on a particular lap, for example by annotating a session utilisation chart or lap chart with the fact that a pit stop occurred, or how long it took. Another way is to look to the data itself as a basis for comparison or analysis.

Let's grab some pit stop data for a particular race and preview the first few lines of the table that we obtain:

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

pitStops.df=dbGetQuery(ergastdb,
  'SELECT d.driverRef, p.stop, p.lap,
    p.time, p.duration, p.milliseconds
  FROM pitStops p, races r, drivers d
  WHERE r.name = "Australian Grand Prix"
    AND r.year=2012
    AND p.raceId=r.raceId
    AND p.driverId=d.driverId')
pitStops.df$duration=as.numeric(pitStops.df$duration)

#Preview the contents of the pitStops table
kable(head(pitStops.df,5),format='markdown')
```

driverRef	stop	lap	time	duration	milliseconds
bruno_senna	1	1	17:05:23	24.599	24599
ricciardo	1	1	17:05:35	32.319	32319
massa	1	11	17:21:08	22.313	22313
rosberg	1	12	17:22:31	23.203	23203
alonso	1	13	17:24:04	22.035	22035

It's not immediately obvious at what point of the pit procedure the timestamp is captured. If it is generated on entry to pit lane, and duration is calculated as the time between the pit entry and pit exit, we should be able to calculate whether there has been a change of position within the pits. For example, if the order of the entry time of day of two drivers pitting about the same time is different to the order of the entry time of day + pit duration, then a position

change will presumably have taken place?

If the time is the entry time and the duration is the time between entry and exit, we can also calculate the pit exit time, and from that try to identify whether one driver enters the pit before another but exits after them. Note that the time only states the time in seconds, whereas the duration includes tenths of a second, so there may be a significant amount of error in calculating the exit time.

$$t_{exit,i} = t_{entry,i} + t_{pit,i}$$

If $t_{entry,A} < t_{entry,B}$ and $t_{exit,A} > t_{exit,B}$, then A entered the pit before B but left after them.

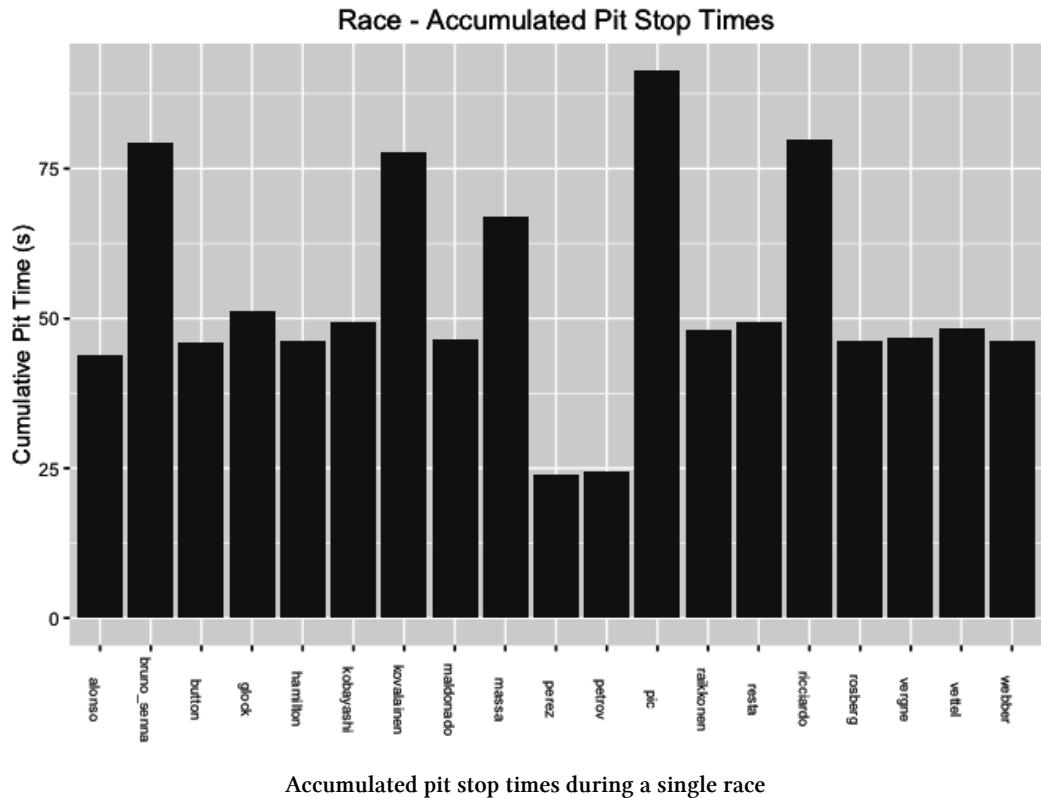
Total pit time per race

A typical report on pitstop behaviour will give the cumulative time spent by each driver whilst pitting. The simplest chart we might create totals the pit time for each driver and displays it by driver. For driver i who stops n times, we have:

$$t_{total,i} = \sum_{j=1}^n t_{stopTime,i,j}$$

```
library("ggplot2")
xRotn=function(s=7) theme(axis.text.x=element_text(angle=-90,size=s))

g=ggplot(pitStops.df,aes(x=driverRef,y=duration))
g=g+geom_bar(stat='identity')
g=g+ggtitle("Race - Accumulated Pit Stop Times")
g=g+ylab("Cumulative Pit Time (s)")
g+xRotn()+xlab(NULL)
```



To display each separate pitstop time independently, we need to create a dummy column that provides a unique (*driver*, *pitstop*) key.

```
#Create a new column by combining the driver identity with the pit stop number
pitStops.df$driverStop=paste(pitStops.df$driverRef,pitStops.df$stop)
```

```
head(pitStops.df$driverStop)
```

```
## [1] "bruno_senna 1" "ricciardo 1"    "massa 1"        "rosberg 1"
## [5] "alonso 1"      "kobayashi 1"
```

A common challenge presented by bar chart views is the order in which we should present the bars. Where a factor is used as the basis of the x-axis, the order in which the items are displayed corresponds to the order of the factor levels. By default, this is based on alphabetical

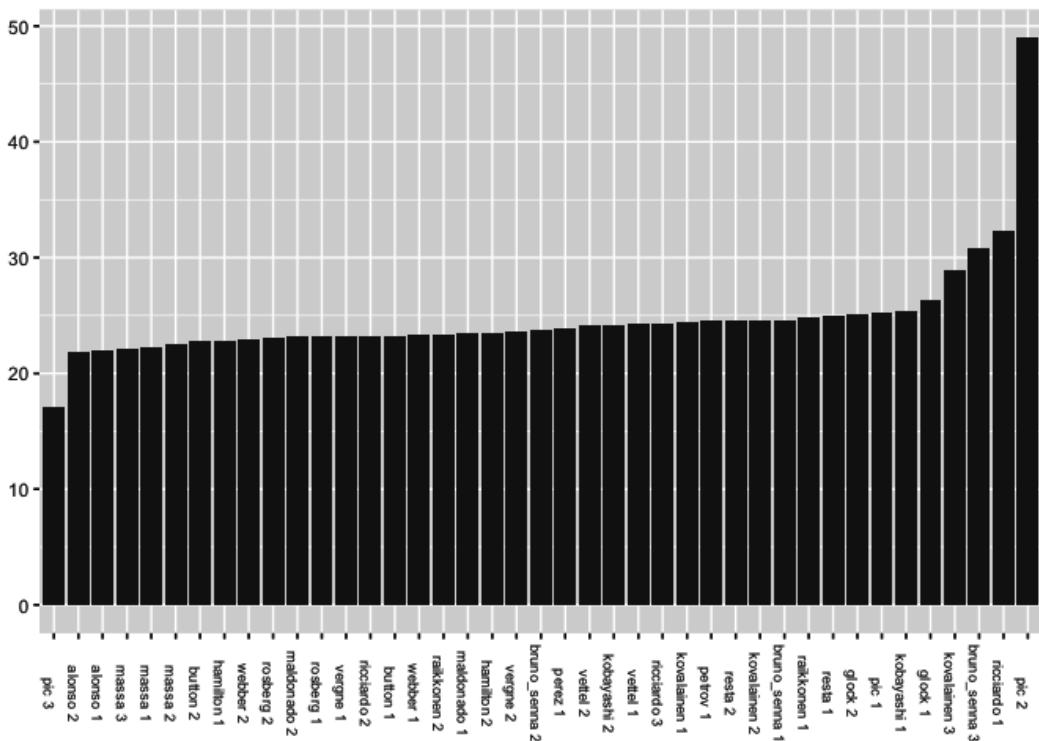
ordering. By choosing `driverRef` for the x-axis, we get an ordering (by default) of the drivers in alphabetical order of their surname. That is, by default, items are ordered based on the first character in surname (or whatever string/text field we are ordering on), then the second, then the third, and so on.

Ordering by surname is meaningful in many situations - we are familiar with the order of the alphabet, so we can use that knowledge to help us find a name that starts with a particular letter or combination letters quite quickly. But in Formula One, there are several other, rather more meaningful orderings, at least to people who follow the sport and are knowledgeable about the teams, drivers, the state of the championship, or the state of the current race weekend.

For the example of a simple bar chart of pit stop times, a sensible ordering might be to sort by pit stop durations.

```
#Create a simple base theme to declutter sketch plots
theme_base = function () theme(
  axis.title.x = element_blank(),
  axis.title.y = element_blank(),
  legend.position="none" )

#reorder() orders the driverStop axis according to the corresponding duration
g=ggplot(pitStops.df, aes(x=reorder(driverStop, duration), y=duration))
#Set the chart type and tweak the theme
g + geom_bar(stat='identity') + xRotn() + theme_base()
```



Bar chart showing pit stop times for each pit event in the Australia 2012, Grand Prix

Looking at the chart ordered in this way, any outliers immediately become obvious. We note that outliers may take an inordinately long time, perhaps because of a problem; or they may appear to be much shorter than a typical pit stop time, perhaps because the pit time actually represents a drive-through or stop-go penalty. One way of trying to identify outliers might be to identify significant step changes in pit stop time by charting the first difference:

```

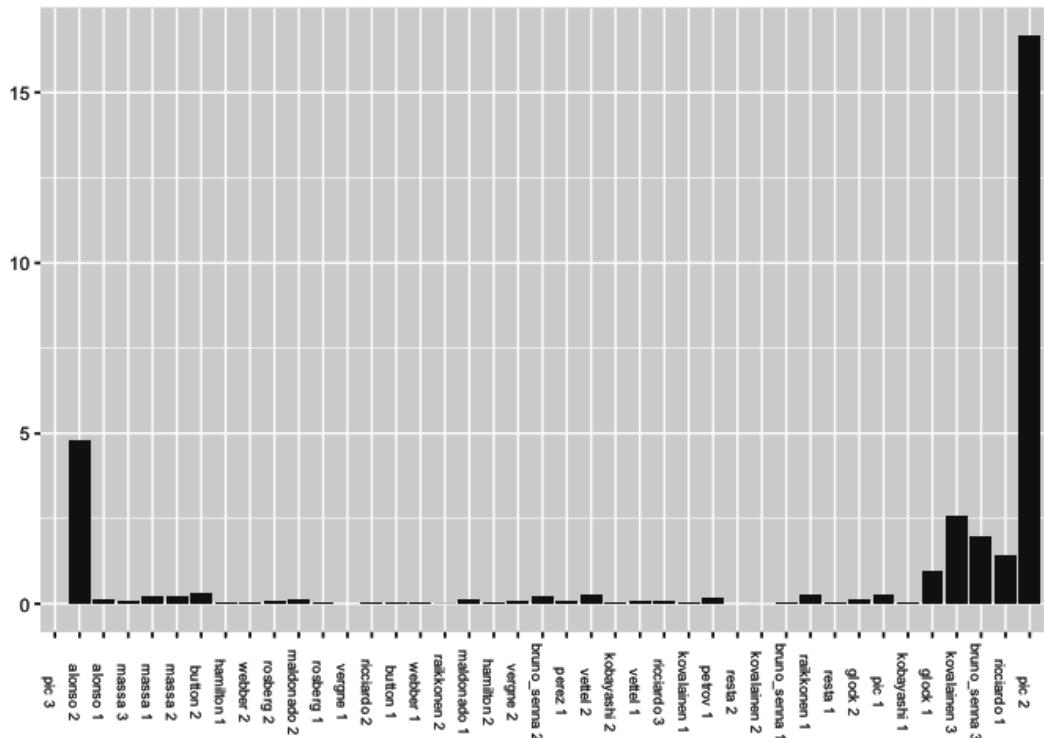
library(plyr)
#Order the pitstop times dataframe in order of increasing stop time
pitStops.df=arrange(pitStops.df, duration)

#Create a first difference column between consecutive, ordered pit times
pitStops.df$tmp_diff=c(0,diff(pitStops.df$duration))

#Plot the chart
g = ggplot(pitStops.df, aes(x=reorder(driverStop, duration), y=tmp_diff))
#Set the chart type and tweak the theme

```

```
g + geom_bar(stat='identity') + xRotn() + theme_base()
```



Bar chart showing the difference between consecutive, increasingly ordered, pit stop times for each pit event in the Australia 2012, Grand Prix

We can then use this information to identify just those pitstop times that appear to be competitive.

Other, more complex, orderings and groupings of the original chart are also possible. For example, we might order the drivers according to a driver's racing number and group the pitstops separately by driver; or we might choose to group the drivers by team and order the teams according to Constructors' Championship standing from the previous year.

Grouping Pit Stop Times

One problem presented by the pit time charts above is that we can't readily see how individual pit stops contributed to the overall (that is, accumulated) pit stop time for a driver, or the relative duration of them by driver.

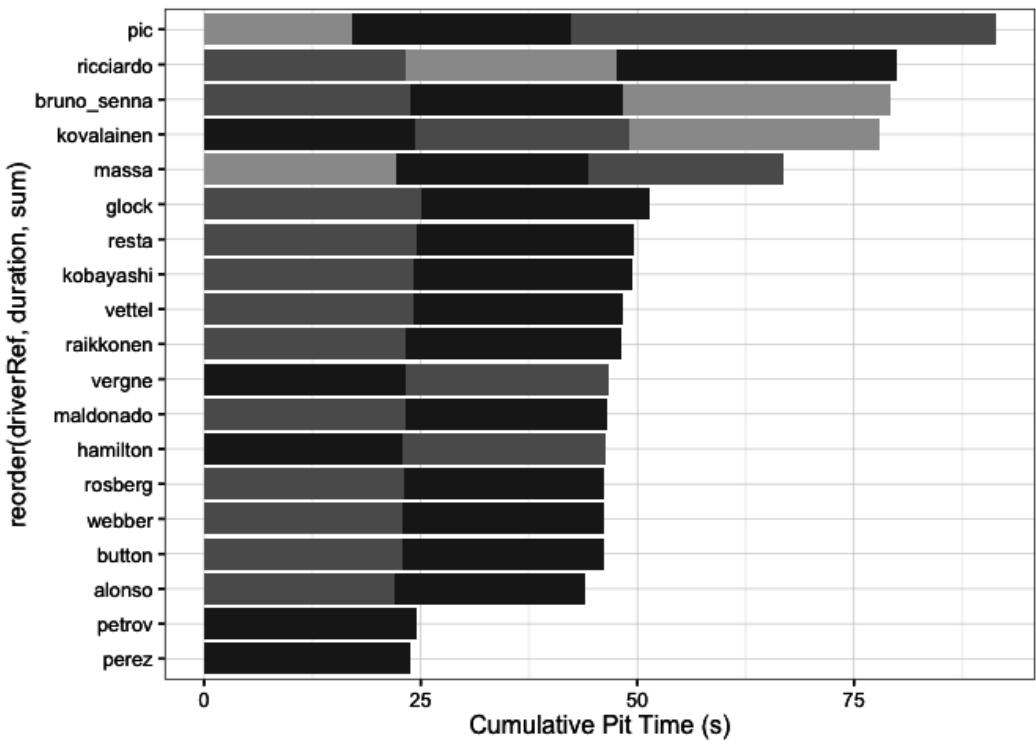
A *stacked bar chart* where the individual components receive their own colour fill is one way of breaking down the cumulative pit stop time into component parts whilst retaining an indication of that total time.

By using a *stacked bar chart* with each bar segment coloured according to the pit stop number for each driver, not only is the number of pit stops made visible, but we can also see how each pit stop contributed that particular driver's overall pit time. If we rotate the chart to give a horizontal layout, we get a more natural positioning of the driver identifiers.

We can further order the bars according to the total pit time, as given by the summed values of the individual pit stop times for each driver.

```
race.plot.pits.cumulativeTime=function(.racePits){
  #The reorder() function sums the durations for stops grouped by driverRef
  #The grouping operation results from setting the fill parameter
  g=ggplot(.racePits,aes(x=reorder(driverRef,duration,sum),y=duration,fill=factor(sto\
p)))
  g=g+geom_bar(stat='identity')
  #Do we want a legend identifying pitstop number or not?
  #g=g+guides(fill=guide_legend(title="Stop"))
  g=guides(fill=FALSE)
  #Add some styling - colour the bars and simplify the background
  g=g+scale_fill_grey(start=0.4,end=0.8)+theme_bw()
  #Work on the labeling
  g=g+ylab("Cumulative Pit Time (s)")
  #Flip the chart to a horizontal bar chart
  g+coord_flip()
}

race.plot.pits.cumulativeTime(pitStops.df)
```

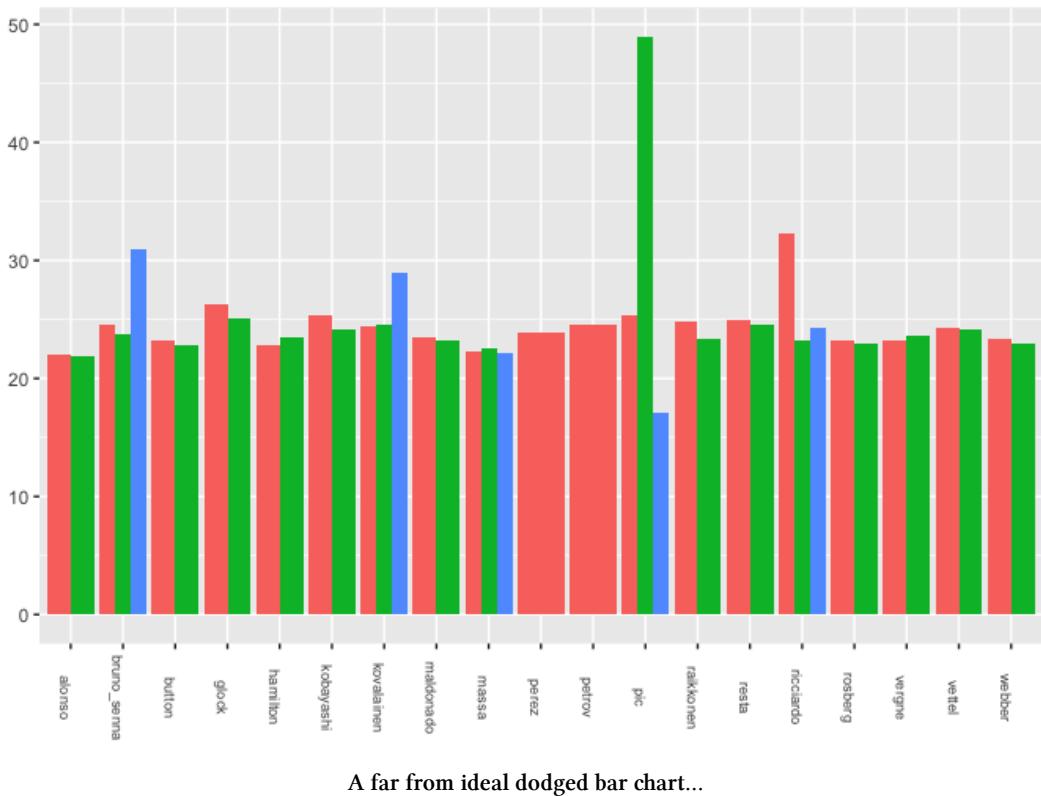


Cumulative pit stop times as a combination of distinct individual pit stop times

One problem with this sort of chart is that we can't easily compare each individual pit stop time. A *dodged bar chart* gets around this problem by allowing us to compare the time taken for each separate pitstop by driver whilst still grouping the stops by driver.

Generating this view presents two challenges as the following base chart demonstrates:

```
#For each driver, use fill colour to identify stop number
g=ggplot(pitStops.df,aes(x=driverRef,y=duration,fill=factor(stop)))
#Generate a dodged bar chart where bar height represents stop time
g+geom_bar(stat='identity',position='dodge') + xRotn() + theme_base()
```



Firstly, we need to find a more sensible way of ordering the drivers. Secondly, in the default chart, the width of the bar for drivers who have only one or two pitstops takes up the whole plot area for the driver (as for example in the case of Perez and Raikkonen in the chart above). What we would really like to see is equal width bars, with spaces for drivers that complete fewer than the maximum number of pit stops for a single driver for a particular race.

To force the plotting of the “empty” bars, we need to reshape the data. First, we generate a wide format data frame with columns for each pit stop number and then identify whether each driver had such a stop.

```

library(reshape2)
driverStop=pitStops.df[,c('driverRef','stop')]

driverStop.wide=dcast(driverStop, driverRef ~ stop,
                      fun.aggregate = length,
                      value.var="stop")

##      driverRef 1 2 3
## 1      alonso 1 1 0
## 2 bruno_senna 1 1 1

```

Then we melt this data frame to generate a long form dataset representing the same data.

```

driverStop.long=melt(driverStop.wide,
                     id.vars = "driverRef",
                     measure.vars = c("1","2","3"),
                     variable.name = "stop")

##      driverRef stop value
## 1      alonso    1     1
## 2 bruno_senna    1     1

```

We can now merge the duration of the corresponding stops back in; setting `all.x=TRUE` ensures that we retain the 0 values for pit stop numbers that a driver did not avail himself of.

```

driverStopDur=merge(driverStop.long,
                    pitStops.df[,c('driverRef','stop',"duration")],
                    by=c("driverRef","stop"),all.x=TRUE)

##      driverRef stop value duration
## 1      alonso    1     1   22.035
## 2      alonso    2     1   21.910

```

If we calculate the total pit stop time for each driver, we can use that as a the basis for ordering the drivers in the actual chart.

```
tot.pitstop=aggregate(duration~driverRef,driverStopDur,sum)
```

```
##      driverRef duration
## 1      alonso    43.945
## 2 bruno_senna   79.264
```

The ordering is achieved by order the factor levels of the `driverRef` attribute.

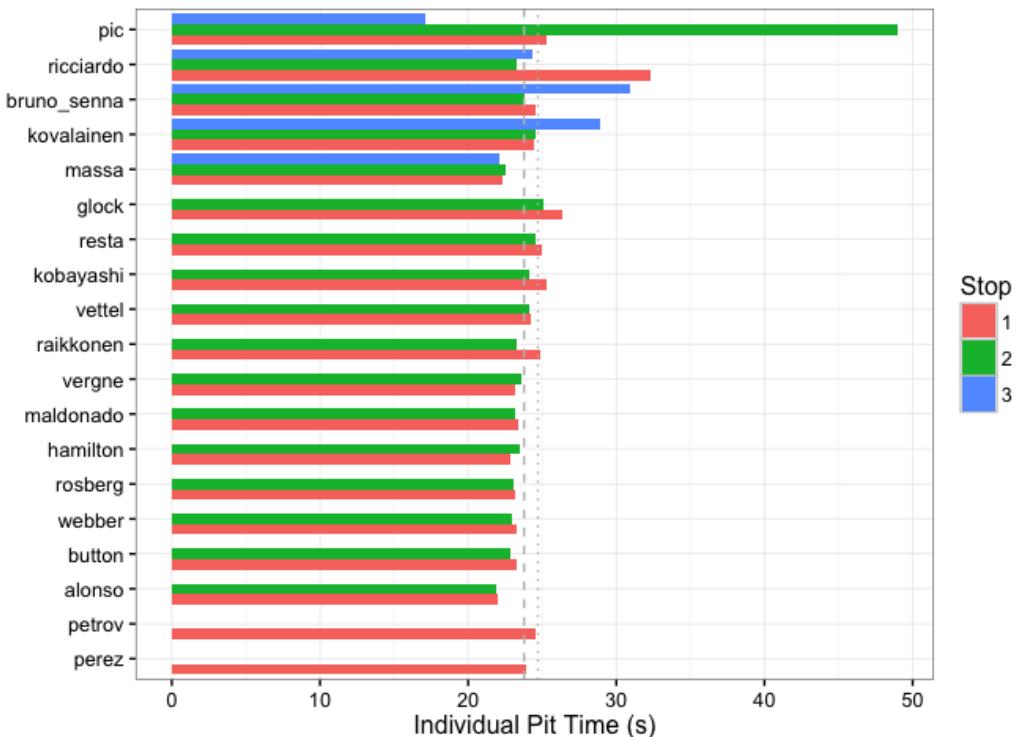
```
driverStopDur$driverRef=factor(driverStopDur$driverRef,
                                levels=tot.pitstop[order(tot.pitstop$duration),
                                "driverRef"])
```

We can now plot a dodged bar plot with empty bars where appropriate, and drivers ordered according to their total pit stop time.

The chart is further annotated with lines showing the median (dashed line) and mean (dotted line) pit stop times.

```
race.plot.pits.dodged=function(.racePits){
  g=ggplot(.racePits,aes(x=driverRef,y=duration))
  g=g+geom_bar(aes(fill=factor(stop)),stat='identity',position='dodge')
  #Annotate the chart with median and mean pit stop times, ignoring null values
  g=g+geom_hline(yintercept=mean(na.omit(.racePits$duration)),
                  linetype='dotted',colour='grey')
  g=g+geom_hline(yintercept=median(na.omit(.racePits$duration)),
                  linetype='dashed',colour='grey')
  g=g+guides(fill=guide_legend(title="Stop"))
  g=g+ylab("Individual Pit Time (s)")
  g+coord_flip()+xlab(NULL)+theme_bw()
}

race.plot.pits.dodged(driverStopDur)
```



Dodged bar chart with empty bars, with drivers ordered by summed pit stop time

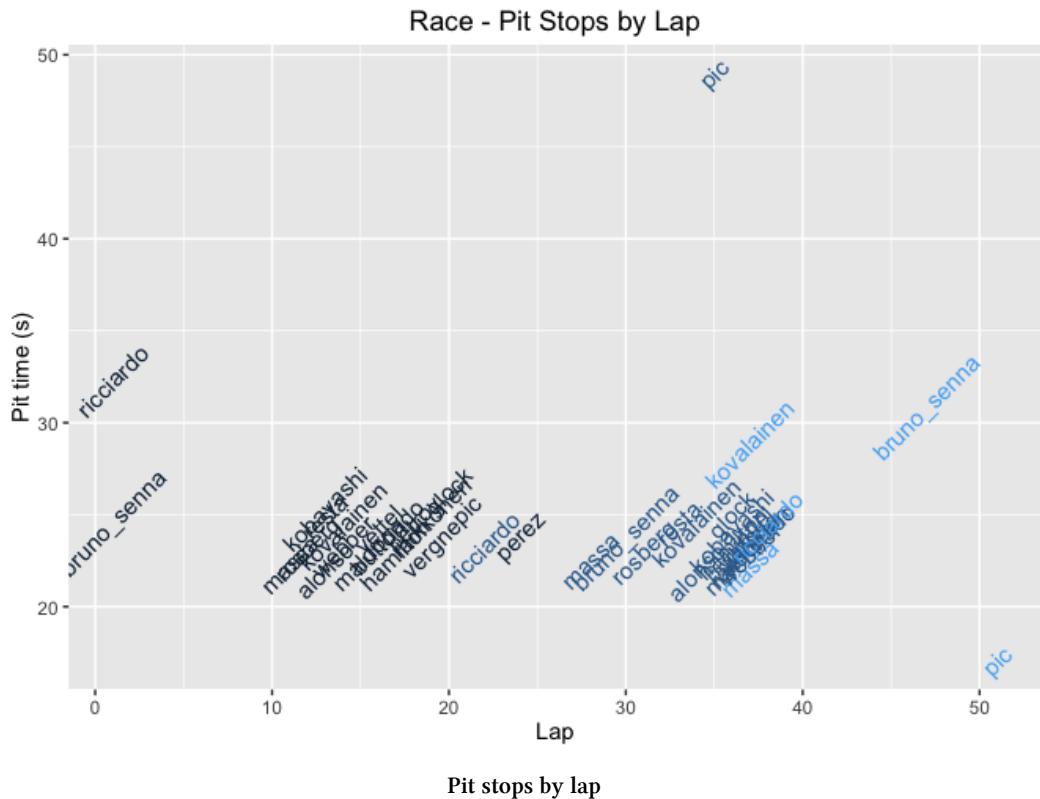
We can also generate a *tabular data report* to summarise this information, identifying the number of times each driver passed through the pit lane during the race, along with the total time spent there.

Pit Stops Over Time

One of the problems with the total pit time summary reports is that they mask information about *when* the drivers were pitting. One possible macroscopic view over all the pitstop data for a particular race would be to use a text plot to display the duration of each pit stop event for each named driver using lap number or time of day for the x-axis. (Time of day is a proxy for race time, and allows us to zoom in on the actual times that drivers pitted during the race. We can generate time of day from lap time data if we know the time at the start of the race, or at the end of the first lap, for example.)

```
#Need newer db - update driverRef to TLID
race.plot.pits.stopsByLap=function(.racePits){
  g=ggplot(.racePits)+geom_text(aes(x=lap,y=duration,label=driverRef,col=stop),
                                size=4,angle=45)
  g=g+ggtitle("Race - Pit Stops by Lap")+xlab("Lap")
  g=g+ylab("Pit time (s)")
  g=g+theme(legend.position="none")
  g
}

race.plot.pits.stopsByLap(pitStops.df)
```

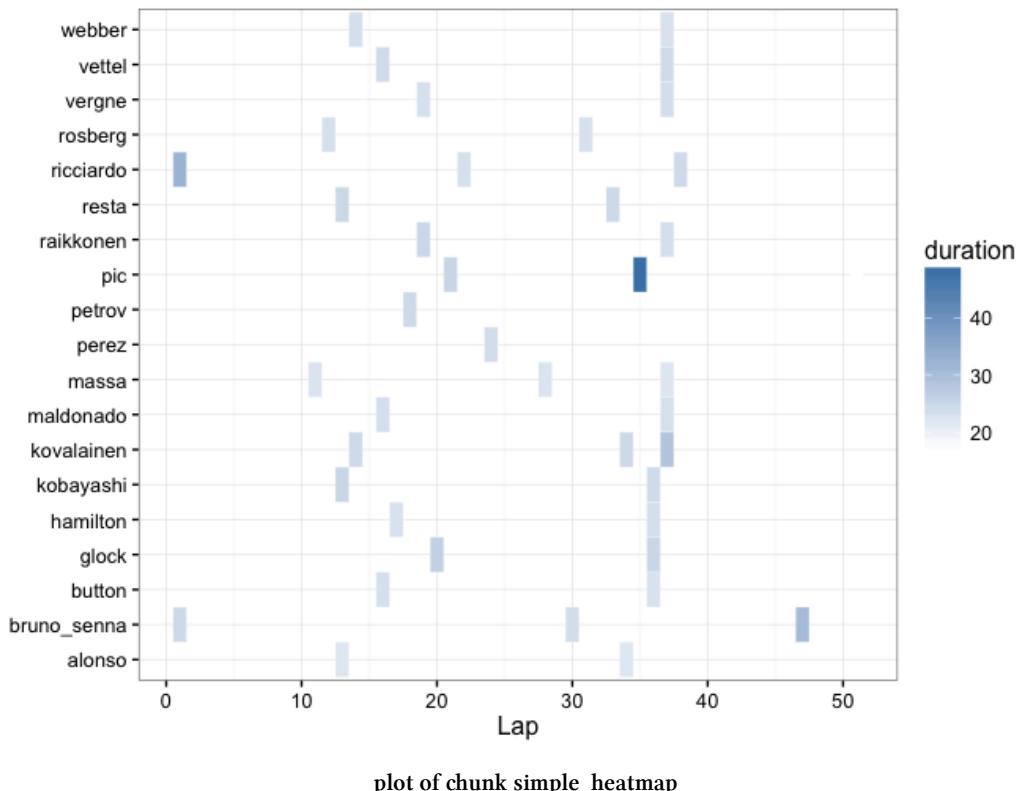


As we have seen in several charts previously, pit stop information may also be used to annotate a wide range of alternative chart types. One efficient way of doing this on a lap chart style display is to use a form of heatmap layer where driver/lap cells are coloured using a `geom_tile()` mapping according to the amount of time spent in the pits.

```

g=ggplot(pitStops.df, aes(lap,driverRef))
g=g+geom_tile(aes(fill=duration),colour = "white")
g=g+scale_fill_gradient(low="white",high="steelblue")
g+xlab('Lap')+ylab(NULL)+theme_bw()

```



Placing this layer on a lower layer, with additional symbols overprinted, provides an indication of the amount of time spent in the pit without adding textual clutter.

From Pitstops to Stints

During the course of a race in constant weather conditions, the two major influences on driver laptime are the mass of the car (which decreases over time as fuel is used up) and the state of the tyres. A *stint* is a sequence of continuous laps that are uninterrupted by an intervening pitstop that typically changes either of these variables (for example, a tyre change

or refueling) or other aspects of the car's set-up. A trivial way of marking a driver's race into a set of stints is to split their race into notional blocks based on the actual record of when the driver pitted from their pit stop report. This forms part of the approach that was taken to identify stints during practice sessions that were highlighted during the development of the practice session utilisation charts.

Stints may also be identified by lap time analysis, in that laps in which a pit stop takes place are likely to result in inlap and outlap lap times that are markedly different from typical racing laps. (That we may be able to reliably identify pit-behaviour from laptime data is mentioned to reinforce the idea that we should be able to identify pitting situations and hence separate stints if the *only* data we have to hand is lap time data.)

Both these approaches are likely to associate stint markers with drive through penalties or certain stop-go penalties that cannot be combined with a service related stops (that is, pit events that do not allow a tyre change; we typically associate the idea of a stint *as a stint on a particular set of tyres*). However, they still provide a good first approximation at how we can subdivide a race for each driver into a discrete set of phases.

Another avenue that may be worth exploring is a graphical approach that helps illustrate possible undercut strategies as described in the chapter on *Event Detection*.

Summary

In this chapter, we have started to explore some of the ways of displaying individual and aggregated pit stop time using a range of bar charts, as well as providing an overview of when different drivers plotted using a text plot. By stacking or dodging bars, we can generate different summary or breakdown views over the data. We also explored a variety of ways of manipulating the sort order of bars and groups of bars based on individual pit stop times and aggregated pit stop times.

Career Trajectory

*This chapter was originally inspired by the **Career Trajectories** chapter of **Analyzing Baseball Data with R**⁵⁵ (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 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 grid? 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. The data we need is spread across several tables in the local database.

```
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)) }

tables=dbGetQuery(ergastdb, 'SELECT name FROM sqlite_master WHERE type = "table"')
tables[['name']]

[1] "circuits" "constructorResults" "constructorStandings" [4] "constructors" "driverStandings" "drivers"
[7] "lapTimes" "pitStops" "qualifying"
[10] "races" "results" "seasons"
[13] "status"
```

⁵⁵<http://baseballwithr.wordpress.com/about/>

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 allows us to make comparisons with the nationality of any teams the driver competes 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/wiki/Lewis_Hamilton
2	heidfeld	HEI	Nick	Heidfeld	1977-05-10	German	http://en.wikipedia.org/wiki/Nick_Heidfeld
3	rosberg	ROS	Nico	Rosberg	1985-06-27	German	http://en.wikipedia.org/wiki/Nico_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 cross-reference a result from the *races* table with information from both the *dirvers* and the *constructors* tables.

If we access the *ergast* API directly, some of the linking and annotation work is done for us. For example, the *driverStandings* API call⁵⁶ includes information about the constructor a driver competed for, as well as their championship position, the number of their wins and the number of points they collected, at the end of each round.

```
#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)
```

⁵⁶<http://ergast.com/api/f1/drivers/alonso/driverStandings>

year	driverId	code	pos	points	wins	car
2001	alonso	ALO	23	0	0	minardi
2003	alonso	ALO	6	55	1	renault
2004	alonso	ALO	4	59	0	renault
2005	alonso	ALO	1	133	7	renault
2006	alonso	ALO	1	134	7	renault
2007	alonso	ALO	3	109	4	mclaren
2008	alonso	ALO	5	61	2	renault
2009	alonso	ALO	9	26	0	renault
2010	alonso	ALO	2	252	5	ferrari
2011	alonso	ALO	4	257	1	ferrari
2012	alonso	ALO	2	278	3	ferrari
2013	alonso	ALO	2	242	2	ferrari
2014	alonso	ALO	6	161	0	ferrari
2015	alonso	ALO	17	11	0	mclaren

For convenience, we can create a local temporary table within the local *ergast* database 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(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
2009	17	17	Abu Dhabi Grand Prix	17
2008	35	18	Brazilian Grand Prix	18
2007	52	17	Brazilian Grand Prix	17

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(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 at 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 a driver's 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 a driver's "F1 age", the number of years they have been competing at that level, or the number of races they have competed in, or even the number of races they have actually 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. For his age, will we use the number of years between the year of his birth and the year of each championship he has competed in?

We can get the driver data back from the *ergast* 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 extracted from the actual date of birth:

```
driverData=function (driverRef){
  q= paste('SELECT * FROM drivers WHERE driverRef== "',driverRef,'" ,sep=' ')
  df=dbGetQuery(ergastdb,q)
  df$yearOfBirth=getYearFromDate(as.Date(df$dob))
  df
}
#Omit the url column from the displayed results
kable(subset( driverData('alonso'), select = -url))
```

driverId	driverRef	code	forename	surname	dob	nationality	yearOfBirth
4	alonso	ALO	Fernando	Alonso	1981-07-29	Spanish	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*.

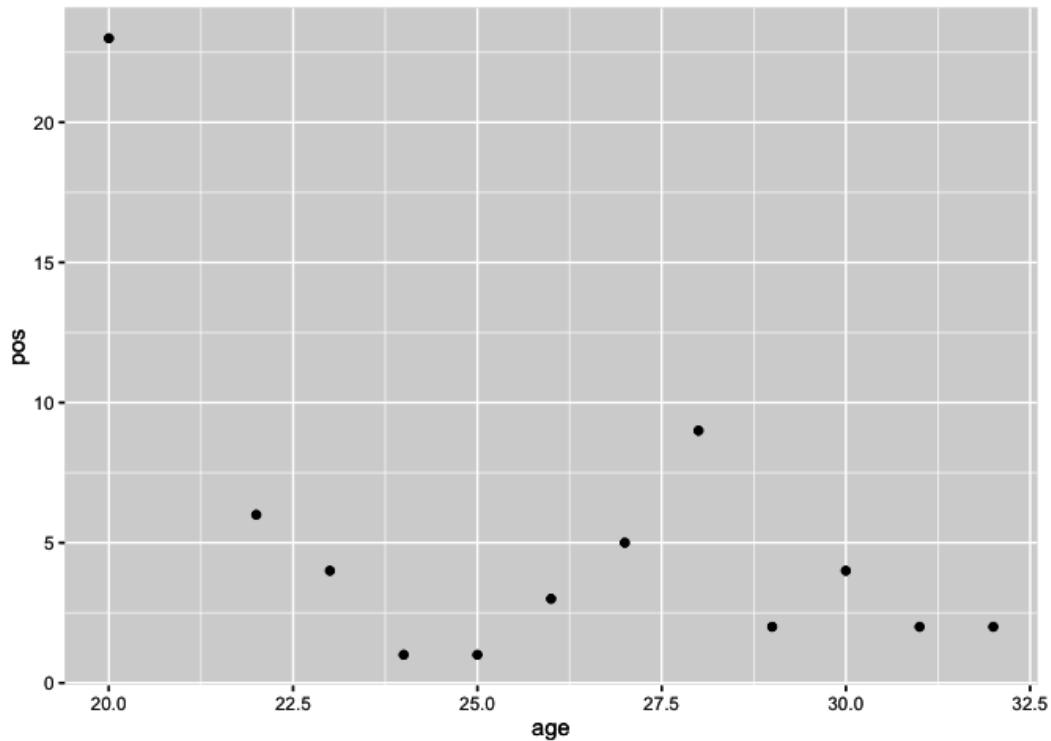
```
drivercareer.aug=function(driverRef){
  ##API equivalent calls:
  #drivercareer=driverCareerStandings.df(driverRef)
  #driverdata=driverData.list(driverRef)
  q=paste('SELECT * from driverChampionship WHERE driverRef="',driverRef,'" ,sep=' ')
  drivercareer=dbGetQuery(ergastdb, q)
  driverdata=driverData(driverRef)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

```
library(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

As we start to think about *modeling* a particular data set, the aim is to produce some sort of mathematical equation that describes how the value of one *dependent variable* changes in response to the values taken by one or more other *independent variables*.

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 modeled as a mathematical function of the age of the driver. We actually use the expression $(Age - 30)$ in the equation so that the value A is a prediction of the championship position for the driver aged 30.

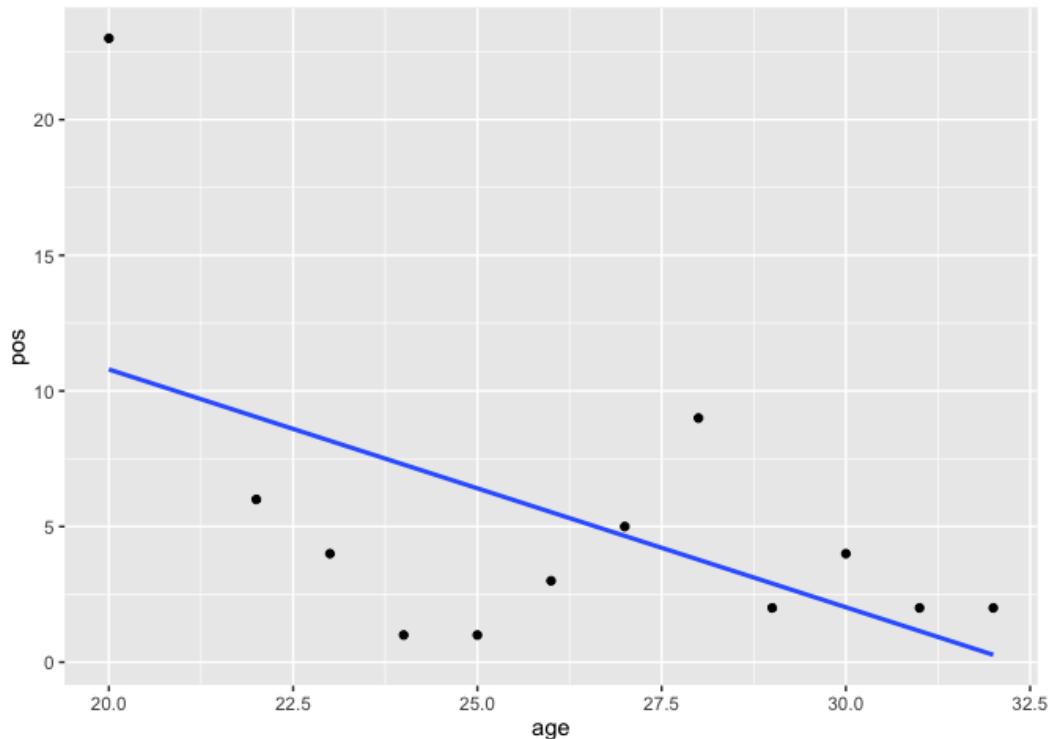
The equation itself defines a best fit curve through the data. The values of A , B and C are chosen so to minimise the difference or distance (also referred to as *error*, or *residual value*) between each data point and the line.

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. We can fit, and then plot, the model directly within `ggplot`:

```
g = ggplot(drivercareer,aes(x=age,y=pos))
#The method specifies what sort of smoothing model to use
#The formula describes the model to use in the smoothing function
#The se parameter identifies whether or not to display standard error bars
g = g + stat_smooth(method = "lm", formula = y ~ I(x-30) , se=FALSE)
g + geom_point()
```

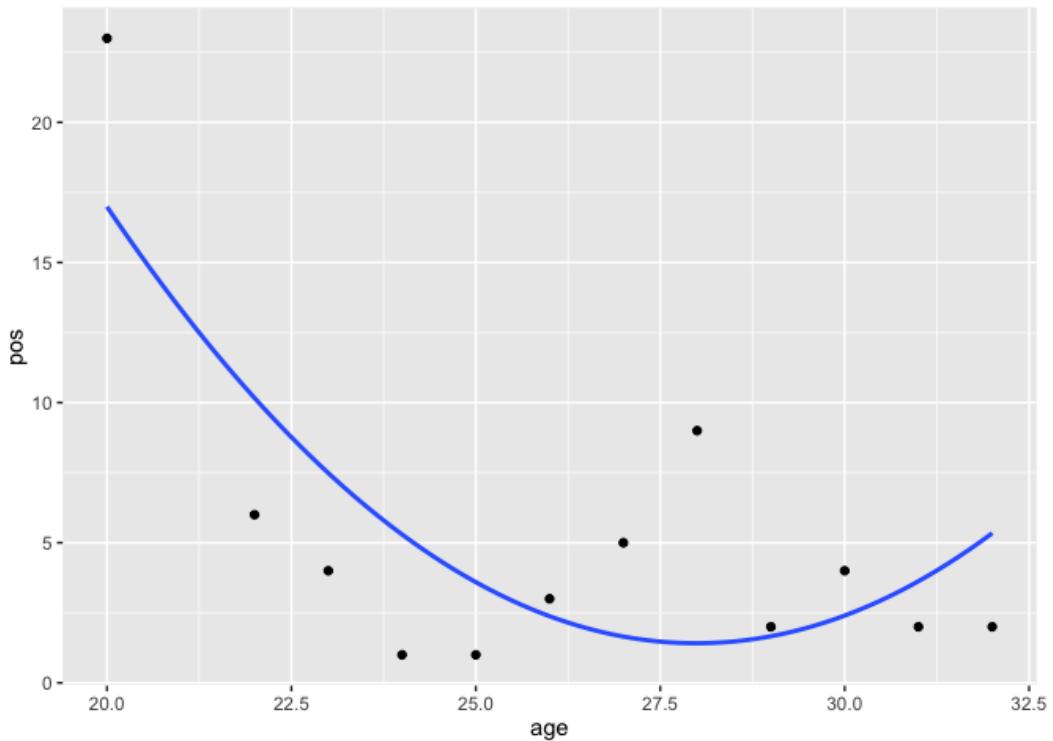


Fernando Alonso's championship positions versus his age in years, with a straight line best fit model

The problem with this sort of line is that it is a straight line, which suggests 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 Albert

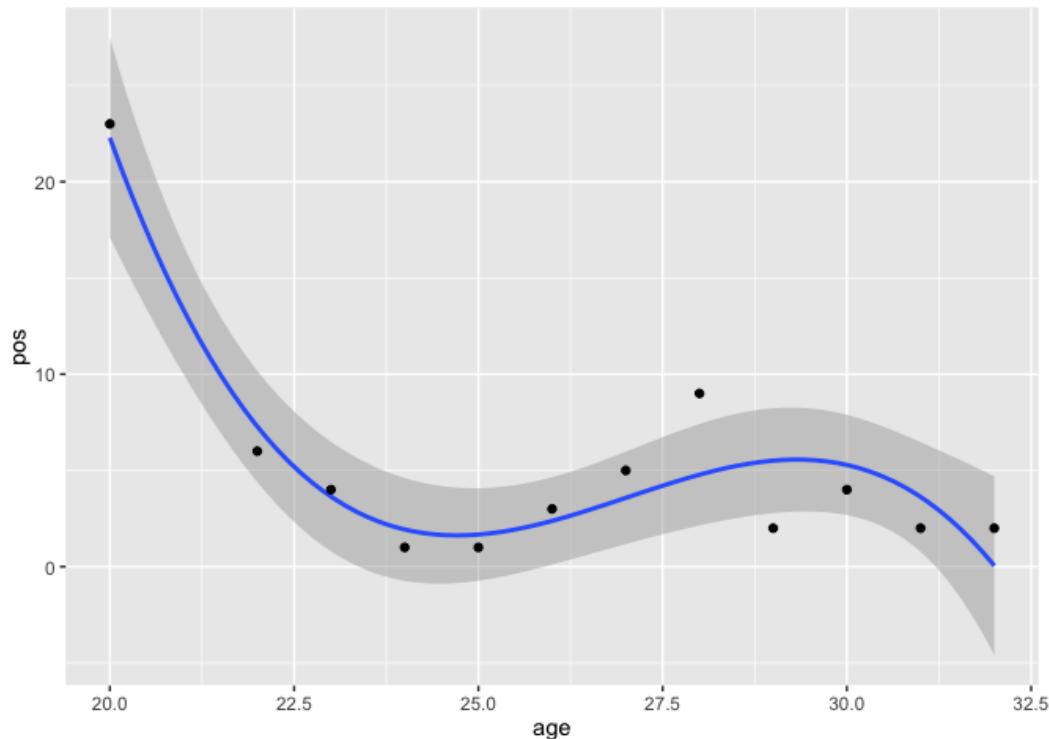
We can add increasing numbers of terms to the model, but the effect of these higher powered terms is often negligible. For example, if we add a cubic term (power 3), we allow the line to have a couple of wiggles (that is, changes of direction).

We can also add confidence limits to the line (by default) to show how confident we are that a point is modeled 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*)).

Although it may be hard for us to interpret these numbers directly, we can do some sums with them. In fact, we can do some *calculus* with them to find the age 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).

This point can be found on the graph as the point at which the curve changes direction, a so called *stationary point* because the gradient equals zero at that point. 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 in turn:

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

and hence:

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

This is then 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:

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

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

Interpreting these results, we see that the peak age for Alonso in terms of overall championship performance was when he was just under 28 years old, with the peak (maximum) position of about 1.4. If we look directly at the plotted curve, we see a 28 year old Alonso actually came 9th in the championship that year, although we also see that this was very much “off trend”. Indeed, the quadratic model shows a best fit line hitting a minimum at around age 28.

Followers of Alonso might well argue that, despite not winning multiple Driver’s Championships, Alonso career performance was in part thwarted by not being in the right team,

and that he repeatedly got more out of a car than one might have reasonably expected. For an analysis that reflects that sort performance, however, we need to rethink what sort of performance measure would help us tell that sort of story.

Confidence limits

TO DO - a note on reading confidence limits

Modeling the Performance of F1 Drivers In General

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 approach this in at least two ways - one way would be to use the age in years, another would be to use the number of years in F1.

The first part of the problem is to identify different drivers with longlasting F1 careers, along with 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 group the results by driver, counting how many final rounds of the season each driver competed in. However, one problem with this approach is that we would miss any drivers who didn't compete in the final round of the season. A better way would be to capture a list of all the F1 races a driver has competed in, generate the set of championship years in which those races took place, and then count how many years were in that set. To plot the career chart, let's get data for drivers who competed in at least 10 seasons.

```
longstanding=dbGetQuery(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

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

```
dbGetQuery(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(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 of F1 drivers 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 Ca' Foscari University of Venice, Department of Economics, Working Paper No. 16/WP/2009⁵⁷).

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 performance. 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.

⁵⁷http://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**⁵⁸ (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 consecutive 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')
```

Let’s start by grabbing some data for a particular driver, specifically the *year*, *round*, *race name*, *grid position*, *final classification* and *points scored* for each race the driver participated in over the course of their career, or (optionally) during a specific championship season:

⁵⁸<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 will 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 paragraph of this chapter, 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 as “*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 it 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 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 between 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 to 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	8	11	0	0	0	0
2013	17	Circuit 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 identifying (using a positive or negative sign) whether a run applied to a streak of 1s or a streak of 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)

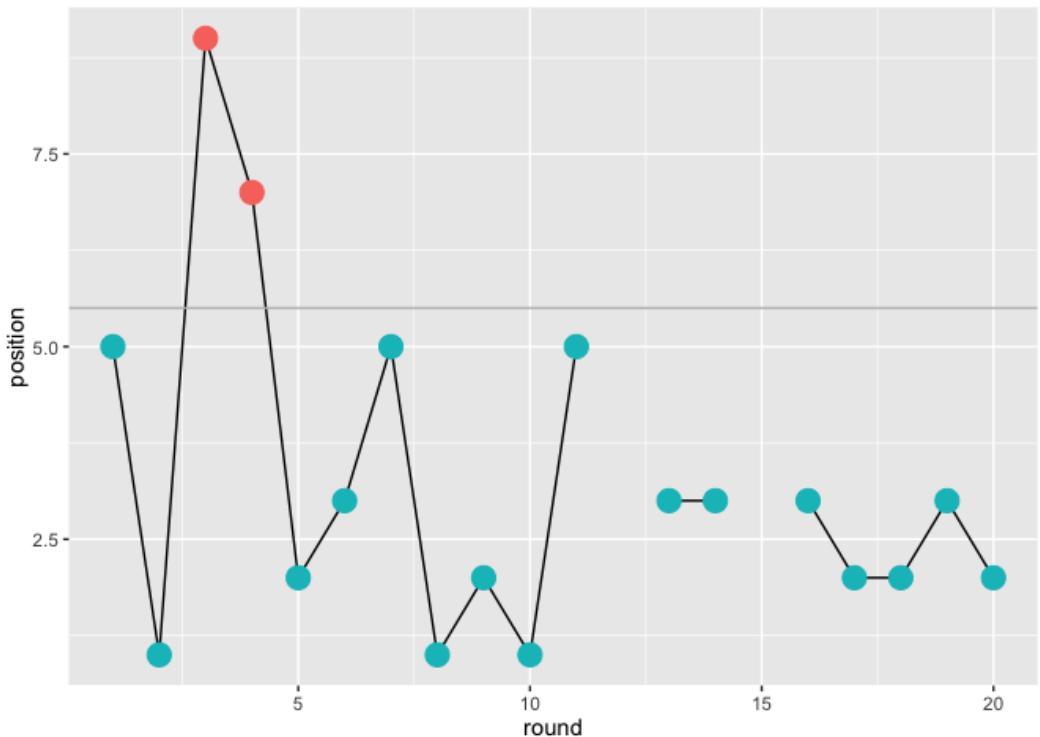
kable( streaks(alonso$top5) )
```

	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 = g + geom_point(aes(colour=factor(top5)), size=5, show_guide = FALSE)
g + geom_hline(yintercept=5.5, colour ="grey")
```



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["1"])>=length,]
  else if (typ==1) streak[abs(streak["1"])>=length & streak["1"]>0,]
  else streak[abs(streak["1"])>=length & streak["1"]<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 Sepang International	Albert Park Grand Prix Circuit Autódromo José Carlos Pace	2001	2003	2003	Sepang International Circuit Autodromo Enzo e Dino Ferrari Circuit de Catalunya
19	20	2	Autodromo Enzo e Dino	Enzo e Dino	2003	2003	2003	
21	21	-1	Ferrari Circuit de Catalunya	Ferrari Circuit de Catalunya	2003	2003	2003	A1-Ring
22	22	1	A1-Ring	Hockenheimring	2003	2003	2003	Hungaroring

The following table shows long streak runs of podium finishes (positive l) and periods without making the podium (negative l) for Fernando Alonso:

```
kable( head(alonso[order(-abs(alonso$l)),]), row.names=FALSE )
```

start	end	l	startc	endc	starty	endy	brokenbyy	brokenbyc
1	18	-18	Albert Park Grand Prix Circuit Sepang International	Albert Park Grand Prix Circuit Circuit Gilles Villeneuve	2001	2003	2003	Sepang International Circuit Indianapolis Motor Speedway
65	79	15	Istanbul Park	Circuit Gilles Villeneuve	2005	2006	2006	Marina Bay Street
106	119	-14	Albert Park Grand Prix Circuit	Autodromo Nazionale di Monza	2008	2008	2008	Circuit Mariana Bay Street
124	136	-13	Albert Park Grand Prix Circuit	Autodromo Nazionale di Monza	2009	2009	2009	Circuit Mariana Bay Street
35	42	-8	Sepang International Circuit	Indianapolis Motor Speedway	2004	2004	2004	Circuit de Nevers Magny-Cours
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 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*⁵⁹ or Wikipedia's *list of Formula One driver records*⁶⁰.

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" (Guardian, "Futures of Force India and Sauber become subject of speculation"⁶¹).

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

⁵⁹<http://www.enterf1.com/blog/166-the-greatest-winning-streaks-in-f1-history>

⁶⁰http://en.wikipedia.org/wiki/List_of_Formula_One_driver_records

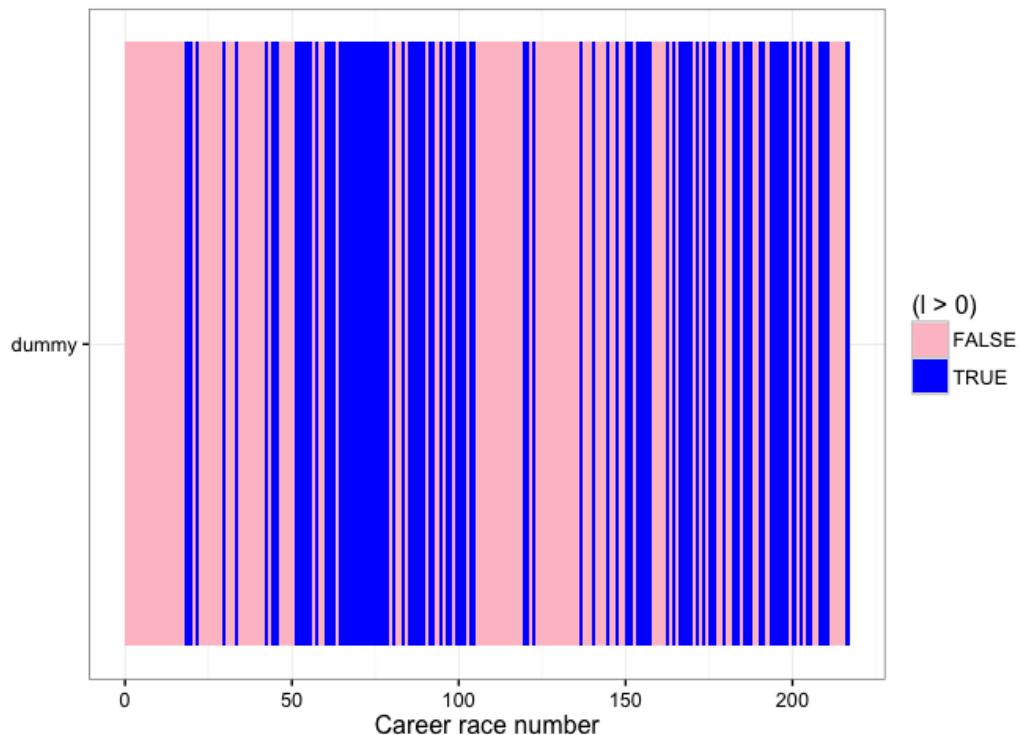
Streak Maps

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 note that the *l* value in the dataframe returned from the `streakReview()` function provides all the information we need: its magnitude helps set the element size, and its sign the fill colour.

```
g = ggplot(alonso, aes(x="dummy", y=abs(1), fill=(1>0)))
g=g+geom_bar(stat='identity')+xlab(NULL)+ylab('Career race number')
g+coord_flip()+scale_fill_manual(values=c('pink','blue'))+theme_bw()
```

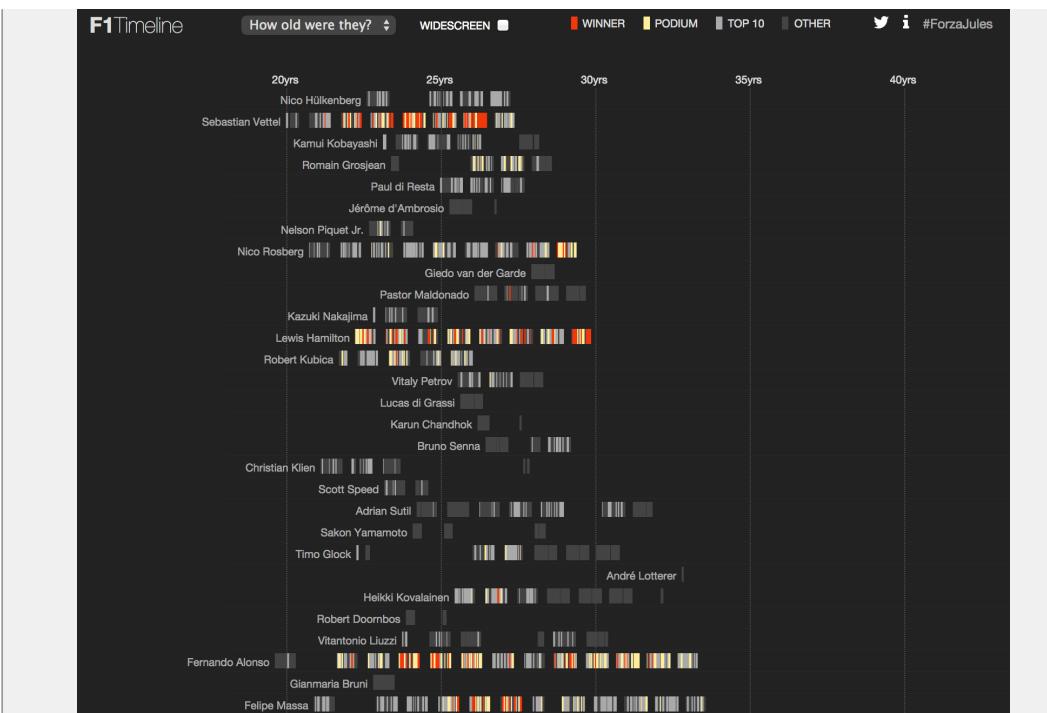


Visualising Alonso's podium streaks with a horizontal filled bar chart

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*⁶¹ 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 Cook's initial sketch 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.

⁶¹<http://animateddata.co.uk/articles/f1-timeline-design/>

Team Streaks

Searching for streaks in performance 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 *driverIds* 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 finished 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 also 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 than 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 END 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 modify 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

Having set the `lockoutFlag` indicator, we then generate a streak report, remembering to sort the dataframe appropriately before attempting to detect any streaks.

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 calculate the time to the first win, we don't necessarily need to calculate the length of an initial not-winning streak. For example, the first win can be identified quite straightforwardly using a query that searches for all wins by a particular driver, orders them by ascending date and then limits the result to just the top row:

```
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 use a customisation of this query to find the number of races a driver had competed in before his first win? Or N^{th} win?

How would you generalise it to find the number of races he had competed in before gaining his first win?



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.

Looking for Streaks Elsewhere

The method described in this chapter for discovering streaks - generate an indicator flag for a feature of interest, look for runs in it - is a general one and can be applied to datasets other than just results.

For example, the method might equally be applied to discovering the number of *consecutive laps led* by a driver in a particular race, over the course of a season, or even across the whole of their F1 career.

Summary

We have seen one way of calculating streaky behaviour in a driver's or team's performance by setting a results flag that identifies 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 complete 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.

Keeping an Eye on Competitiveness - Tracking Churn

If every race was a processional race, and every season saw the same teams finishing in the same order year on year, we might start to doubt that Formula One was in any way competitive, at least in the sense of being a points based competition between teams or drivers (we might still argue there is competition for a team to better its own laptimes, for example, or reduce its pit stop times).

One way of trying to measure competitiveness within a sport is to look at position *churn*, or the change in relative standings or rankings of competitors over a period of time. Churn was developed as a measure of competitiveness within professional sports leagues across seasons by Mizak et al. (2007) (Mizak, D, Neral, J & Stair, A (2007) *The adjusted churn: an index of competitive balance for sports leagues based on changes in team standings over time*. Economics Bulletin, Vol. 26, No. 3 pp. 1-7⁶²), originally using the context of Major League Baseball.

One of the ways in which the churn indicator has been used is to model the extent to which “competitiveness” influences audience interest: organised competitive sports are businesses and as such economists are interested in how competitiveness translates to - or detracts from - an economic take on the sport. From a sports reporting or data journalism point of view, we are perhaps more interested in the extent to which it might signal some element of “interestingness” or “newsworthiness”, or help us identify particularly noteworthy races.

According to Mizak et al. (2007), *churn* is defined as follows:

$$C_t = \frac{\sum_{i=1}^N |f_{i,t} - f_{i,t-1}|}{N}$$

where C_t is the churn in team standings for year t , $|f_{i,t} - f_{i,t-1}|$ is the absolute value of the i -th team’s change in finishing position going from season $t - 1$ to season t , and N is the number of teams.

⁶²<http://core.kmi.open.ac.uk/download/pdf/6420748.pdf>

The *adjusted churn* is defined as an indicator with the range 0..1, calculated by dividing the churn, C_t , by the maximum churn, C_{max} . The value of the maximum churn depends on whether there is an even or odd number of competitors:

$$C_{max} = N/2, \text{ for even } N$$

$$C_{max} = (N^2 - 1)/2N, \text{ for odd } N$$

As Berkowitz et al. (2011) describe in their paper on race outcome uncertainty in NASCAR races, “[a]t least three levels of outcome uncertainty have been defined in the literature: event-level, season-level, and inter-season uncertainty.” That is, there may be competition *in the context of a particular race*, competition across races *within a season*, and competition *across seasons* (Berkowitz, J. P., Depken, C. A., & Wilson, D. P. (2011). *When going in circles is going backward: Outcome uncertainty in NASCAR*. Journal of Sports Economics, 12(3), 253-283).

In recognition of these different aspects of competitiveness, Berkowitz et al. reconsidered churn as applied to an individual NASCAR race (that is, at the event level). In this case, $f_{i,t}$ is the position of driver i at the end of race t , $f_{i,t-1}$ is the starting position of driver i at the beginning of that race (that is, race t) and N is the number of drivers participating in the race. Once again, the authors recognise the utility of normalising the churn value to give an *adjusted churn* in the range 0..1 by dividing through by the maximum churn value.

Taking a similar line of reasoning, we might also define the lap-on-lap churn by treating $f_{i,t}$ as the position of driver i at the end of lap t , $f_{i,t-1}$ as their position at the start of that lap (that is, at the end of lap $t - 1$) and N is the number of drivers completing that lap. A time series view of this indicator across all laps in a race may help us identify turbulent periods within a race when there significant position changes occurring (such as during pit stop windows). This approach might also be compared to the one briefly introduced in the chapter on *Lapcharts*, where we generated a table of *position change counts* that could act as an estimate of how much position change activity took place within a race.

Across a season, we might denote churn with reference to position in the championship standings at the start and end of a particular championship round. Within a race, we might additionally consider *first lap churn* (churn between the grid positions and the positions at the end of lap 1) or churn between qualifying position and final race position.

Calculating Adjusted Churn - Event Level

Following Berkowitz et al. (2011), let's begin by considering the adjusted churn across races for a single F1 season, in particular the 2013 season. We'll measure the churn at the level of individual drivers. To do this, we need the starting (grid) and finishing positions of each driver for each round.

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

q=paste('SELECT round, name, driverRef, code, grid,
         position, positionText, positionOrder
      FROM results rs JOIN drivers d JOIN races r
      ON rs.driverId=d.driverId AND rs.raceId=r.raceId
      WHERE r.year=2013',sep='')

results=dbGetQuery(ergastdb,q)
```

We can define functions to calculate the churn and adjusted churn values from a set of rank positions.

```
#The modulus function (%%) finds the remainder following a division
is.even = function(x) x %% 2 == 0
churnmax=function(N)
  if (is.even(N)) return(N/2) else return(((N*N)-1)/(2*N))

churn=function(d) sum(d)/length(d)
adjchurn = function(d) churn(d)/churnmax(length(d))
```

Using these functions, it's straightforward enough to calculate the churn for each race based on the absolute difference (*delta*) between the grid position and final position of each driver.

```

library(plyr)
results['delta'] = abs(results['grid']-results['positionOrder'])

churn.df = ddply(results[,c('round','name','delta)], .(round,name), summarise,
                 churn = churn(delta),
                 adjchurn = adjchurn(delta)
)

```

round	name	churn	adjchurn
1	Australian Grand Prix	4.000000	0.3636364
2	Malaysian Grand Prix	5.272727	0.4793388
3	Chinese Grand Prix	3.090909	0.2809917
4	Bahrain Grand Prix	3.727273	0.3388430
5	Spanish Grand Prix	4.000000	0.3636364
6	Monaco Grand Prix	3.181818	0.2892562
7	Canadian Grand Prix	3.636364	0.3305785
8	British Grand Prix	5.181818	0.4710744
9	German Grand Prix	3.181818	0.2892562
10	Hungarian Grand Prix	4.000000	0.3636364
11	Belgian Grand Prix	3.909091	0.3553719
12	Italian Grand Prix	2.363636	0.2148760
13	Singapore Grand Prix	4.636364	0.4214876
14	Korean Grand Prix	3.909091	0.3553719
15	Japanese Grand Prix	3.454546	0.3140496
16	Indian Grand Prix	3.818182	0.3471074
17	Abu Dhabi Grand Prix	2.454546	0.2231405
18	United States Grand Prix	2.090909	0.1900826
19	Brazilian Grand Prix	4.181818	0.3801653

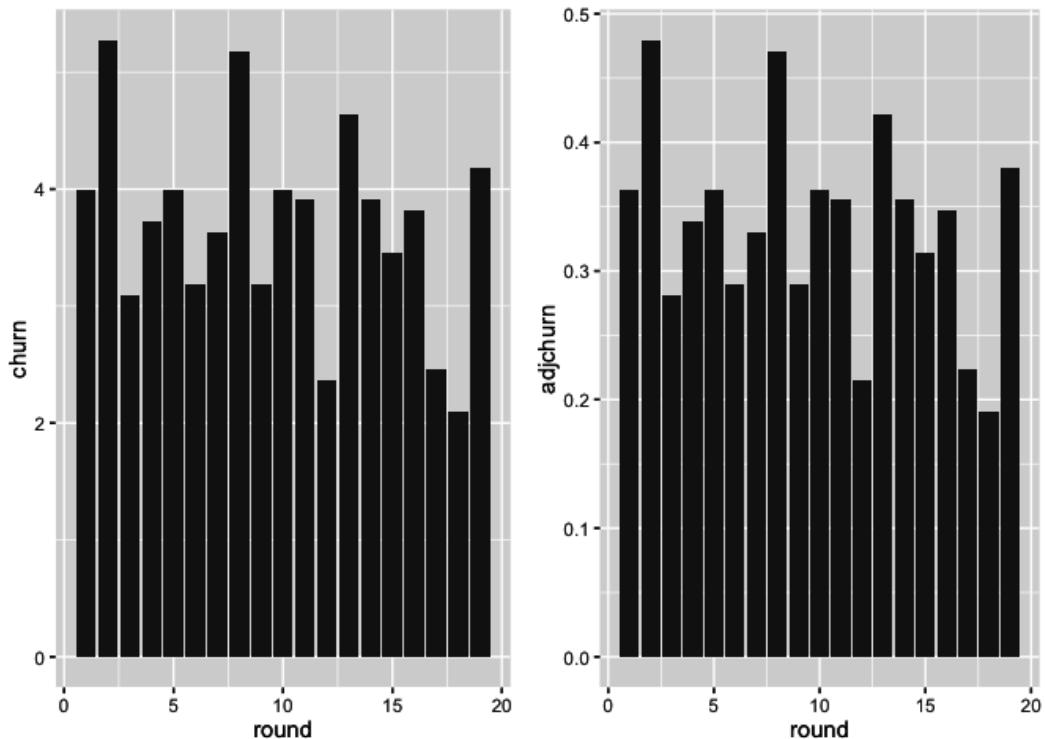
One way of displaying this tabular information graphically is to use bar charts.

```

library(ggplot2)
library(gridExtra)

g1 = ggplot(churn.df)+geom_bar(aes(x=round,y=churn),stat='identity')
g2 = ggplot(churn.df)+geom_bar(aes(x=round,y=adjchurn),stat='identity')
grid.arrange(g1, g2, ncol=2)

```

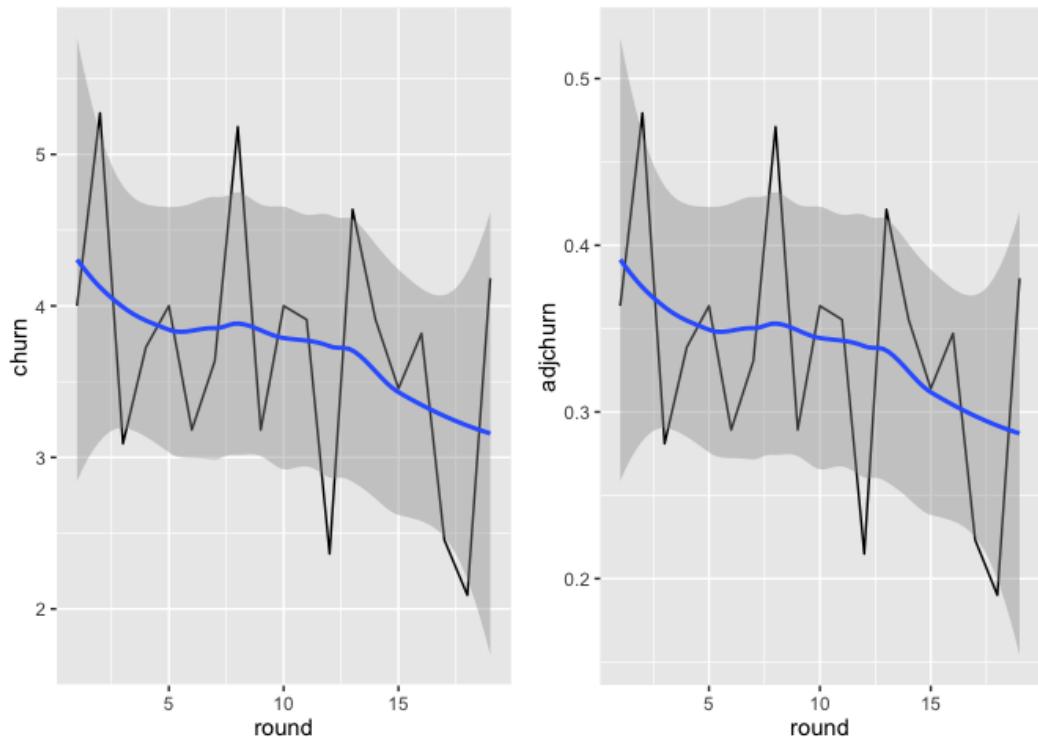


Bar charts showing churn and adjusted churn values for races in the 2013 season

So what sorts of things might we learn from these results? By inspection, we notice that rounds 2 and 8, the Malaysian and British Grand Prix rounds had a significant change in rankings from the original grid positions to the final result, whereas the Italian, Abu Dhabi and United States Grands Prix rounds saw far less difference between the starting and finishing positions. (*We really should confirm this graphically, for example using start/finish slopegraphs to compare those two races by eye.*)

We can also display the time series using line charts. In this case, we get a slight visual indication of a possible trend in decreasing churn across the course of the season although the effect (if any) looks to be highly uncertain.

```
g1 = ggplot(churn.df,aes(x=round,y=churn))+geom_line() + geom_smooth()
g2 = ggplot(churn.df,aes(x=round,y=adjchurn))+geom_line() + geom_smooth()
grid.arrange(g1, g2, ncol=2)
```



Line charts showing churn and adjusted churn values for races in the 2013 season. A loess model fit line suggests a highly uncertain/weak decreasing trend in churn (that is, loss of competitiveness) over the course of the season



Are Some Circuits Less Competitive than Others?

By comparing average churn values for each circuit over several seasons, we may get some indication of the likelihood with which significant churn may occur, compared to circuits that appear to encourage rather less competition. Facet a set of year-on-year churn data by circuit to explore whether different circuits appear to have churn values that are markedly different from the churn values of other circuits. How reliable an indicator do you think churn might be as a way of identifying circuits that appear to offer races whose results do not appear to be predetermined by the initial grid positions?



Does the Weather Affect Competitiveness?

Another comparison we might make is to consider churn relative to the prevalent weather conditions, for example to explore whether or not a wet race is likely to have and influence on race competitiveness. See if you can identify a source of prevailing weather information for each Grand Prix in a given period, and then explore the extent to which particular weather types either do or do not appear to influence churn. It may be worth further grouping results based on both the circuit *and* the prevailing weather conditions.

Churn Indicators that Reflect Drivers Retiring from the Race

One of the problems with the approach to calculating churn described above is that we calculate position changes for all drivers rather than just those based on drivers that are classified in a particular race. That is, the *positionOrder* results value used as the final position value returns an integer value for each driver that is used to rank all drivers irrespective of whether they were actually classified in a race. However, we might also wish to calculate churn based solely on drivers that do make it far enough into the race to guarantee that they *are* classified. In this case, the size of the competitive field is naturally given by the number of classified drivers; but how should we calculate the magnitude of the change in position? If a car that starts 22nd on the grid is classified in 15th position because seven cars that started ahead of it on the grid drop out, is the position change seven, or zero?

In this section, we'll sketch out what difference, if any, this might make, although for now we will avoid any consideration about the rationale for *not* using the simplistic approach originally described, and avoid any detailed exploration about how we might interpret any differences in the results of each model.

We will consider three different scenarios:

- the original model (*churn*, *adjchurn*), described above, where position changes are calculated as the absolute difference between the grid position and the final rank position, as given in news reports. This includes cars that were not classified;
- a modification of the original model (*finchurn*, *finadjchurn*) using a limited set of driver results, specifically including just results from drivers that were classified, but using their original grid position to classify the position changes. (This may break the adjusted churn calculation, because a position changes may be calculated from a grid position that is greater than the number of cars that appear in the final classification. In addition, the size of the field that is awarded a final classification may vary significantly race on race.);

- a further modified model (*rfinchurn*, *rfinadjchurn*) that considers just those drivers that were classified in a race, but adjusts the grid position to the rank of the original grid position amongst the classified drivers to give a new gridrank position. So for example, in a field in which two cars are not classified, the driver at the back of the grid (for example, in grid position 22) that is classified will have a re-ranked grid position of 20 when calculating the position change.

```
#Start by filtering the dataframe to contain only classified drivers
#Then calculate the ranked grid position equivalent for the classified drivers
classresults=ddply(results[!(is.na(results['position']))],,
  .(round), mutate,
  delta=abs(grid-positionOrder),
  gridrank=rank(grid),
  rankdelta=abs(gridrank-positionOrder)
  )

#Calculate the churn values for the classified and gridranked drivers
churn2.df=ddply(classresults[,c('round','delta','rankdelta')],,
  .(round), summarise,
  finchurn=churn(delta),
  finadjchurn=adjchurn(delta),
  rfinchurn=churn(rankdelta),
  rfinadjchurn=adjchurn(rankdelta)
  )

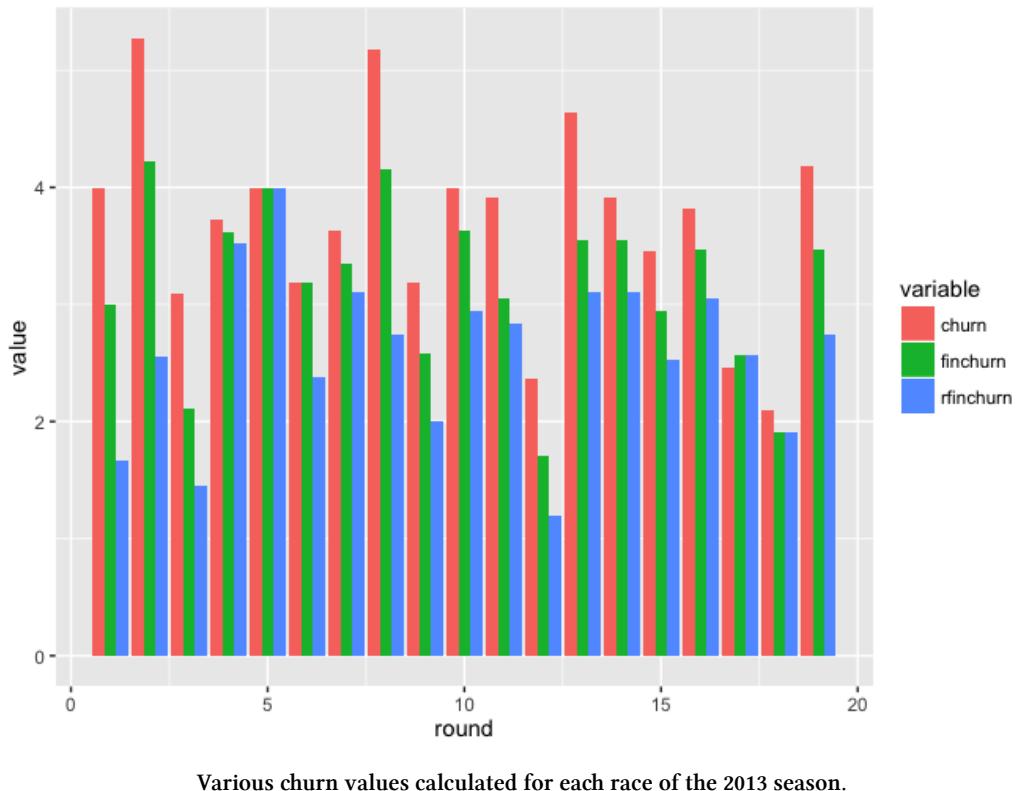
#Merge the results with the original results
churn3=df=merge(churn.df,churn2.df,by='round')
```

To plot the data, we will reshape it into a long form that allows us to readily group the data based on which model we are applying.

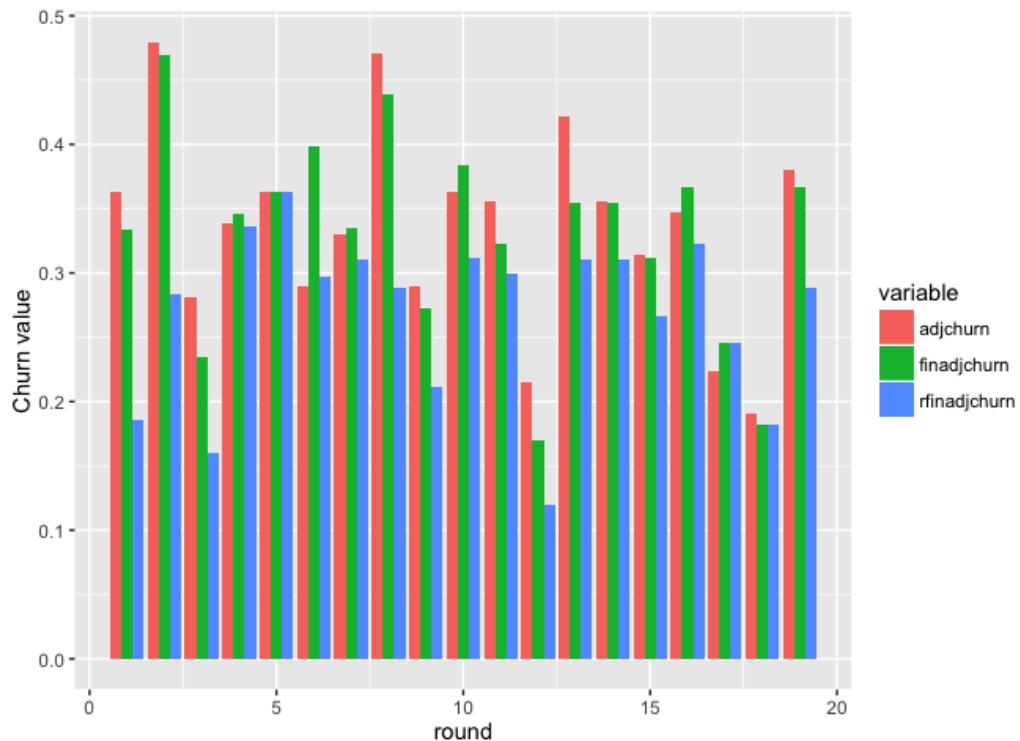
```
library(reshape2)
churnData=melt(churn3.df, id.vars='round', measure.vars=c('churn','finchurn','rfinchurn'))
adjChurnData=melt(churn3.df, id.vars='round', measure.vars=c('adjchurn','finadjchurn',
,'rfinadjchurn'))
```

By inspection of the following charts, we note that the rank order of the churn and adjusted churn values for the races differs for the different population and grid/gridrank value combinations.

```
ggplot(churnData) + geom_bar(aes(x=round, y=value, fill=variable),
                             stat='identity', position='dodge')
```



```
g=ggplot(adjChurnData) + geom_bar(aes(x=round, y=value, fill=variable),
                                   stat='identity', position='dodge')
g+ ylab("Churn value")
```



Various adjusted churn values calculated for each race of the 2013 season.

Note that the different measures behave differently with respect to each other in different races. For example, in round 1, *adjchurn* > *finadjchurn* > *rfinadjchurn*, whereas in rounds 4, 5 and 19 all values are the same or more or less the same, and in rounds 10 and 16 *finadjchurn* > *adjchurn* > *rfinadjchurn*.

It may be worth exploring whether any of the patterns in how the different measures compare with each other in a given race are indicative of any particular features about the race. *This is left for future exploration.*

Calculating Adjusted Churn - Across Seasons

So far, we have been focusing on churn in the context of position changes in the standings of the Drivers' Championship. In this section, where we consider churn from one season to the next over the lifetime of the F1 World Championship, we will additionally consider churn insofar as it relates to standings in the Constructors' Championship.

But first, let's see how churn affected the Drivers' Championship.

Year-on-year Churn - Drivers' Championship

The data source we will use for this exploration is the online *ergast* API. We can pull down the results for championship standings for each year we require, combining the results into a single data frame.

```
source('ergastR-core.R')

seasons=data.frame()
#Earliest 1950
for (year in seq(1950,2014,1)){
  seasons=rbind(seasons,seasonStandings(year))
}
```

Note that this routine hits the ergast API 65 times in constructing the final data frame.

In calculating the churn in standings from one championship year to the next, we need to draw on the difference between a driver's standing in one year and their standing in the previous year.

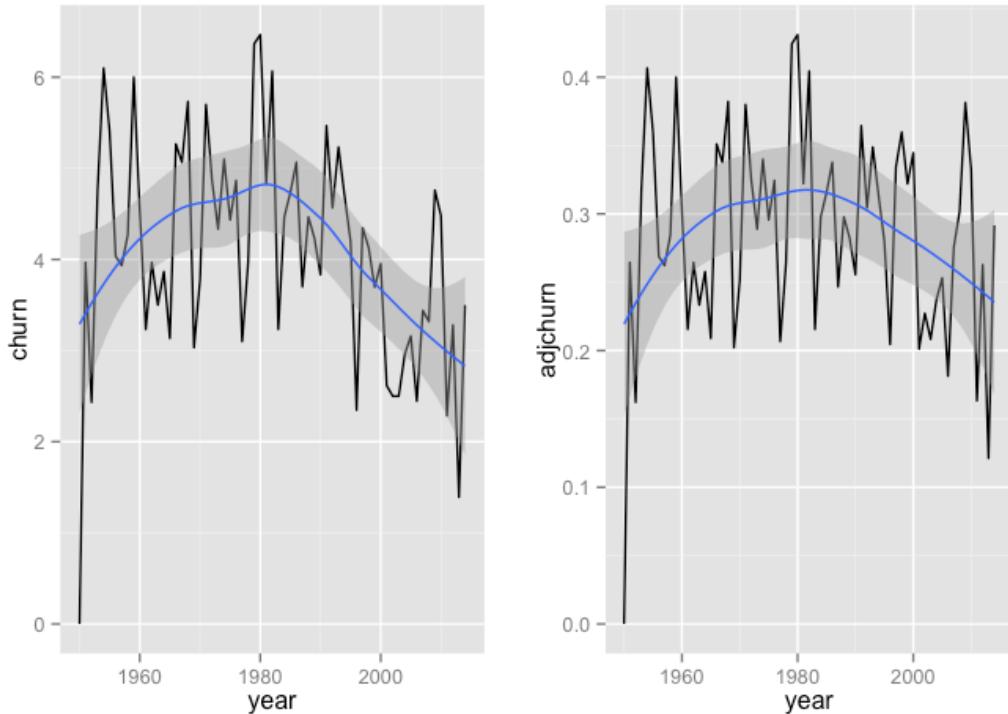
```
seasons=ddply(seasons, .(driverId), transform, delta=c(0, abs(diff(pos))))
```

We can then apply the churn function to these differences, summarising the data for each year of interest.

```

g1 = ggplot(season.churn.df,aes(x=year,y=churn))+geom_line() + geom_smooth()
g2 = ggplot(season.churn.df,aes(x=year,y=adjchurn))+geom_line() + geom_smooth()
grid.arrange(g1, g2, ncol=2)

```



Churn in the Drivers' Championship standings, 1950-2014

As well as calculating the churn and adjusted churn values as calculated across all drivers, we can also limit the results to show the churn amongst just the points scoring drivers, or amongst the top 10 drivers as classified at the end of each year.

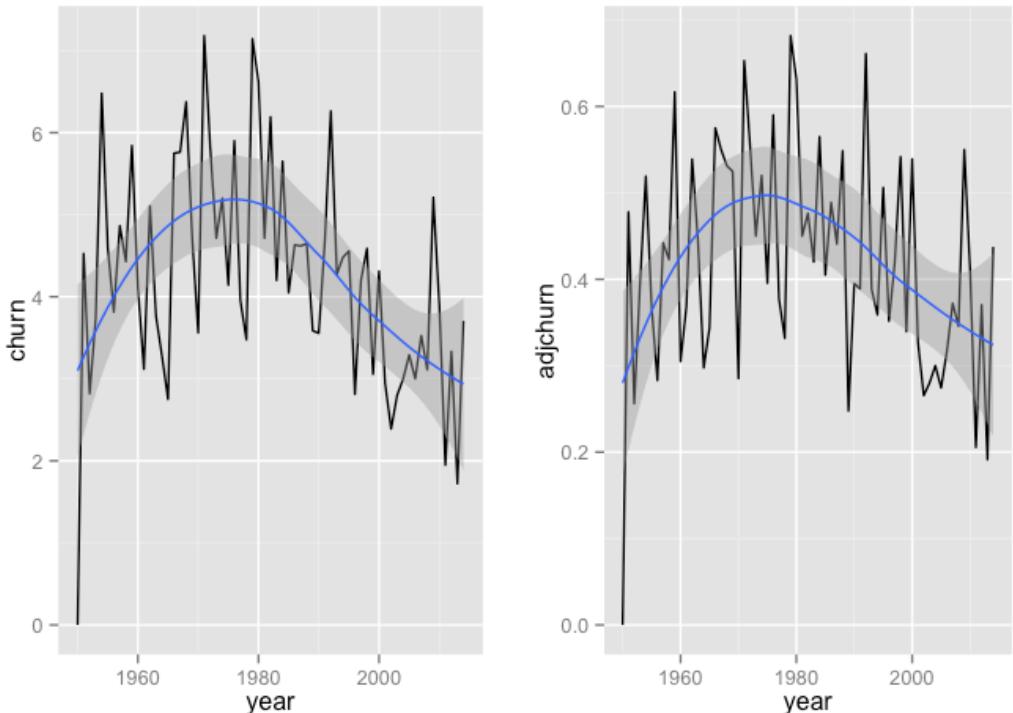
The trendlines suggest that competitiveness - at least as measured by positional churn in the Drivers' Championship - peaked around 1980 and has been in decline ever since.

```

season.churn.points.df = churner( seasons[seasons['points']>0,] )

g1 = ggplot(season.churn.points.df,aes(x=year,y=churn))+geom_line()+geom_smooth()
g2 = ggplot(season.churn.points.df,aes(x=year,y=adjchurn))+geom_line()+geom_smooth()
grid.arrange(g1, g2, ncol=2)

```



Churn in the Drivers' Championship standings for points scorers, 1950–2014

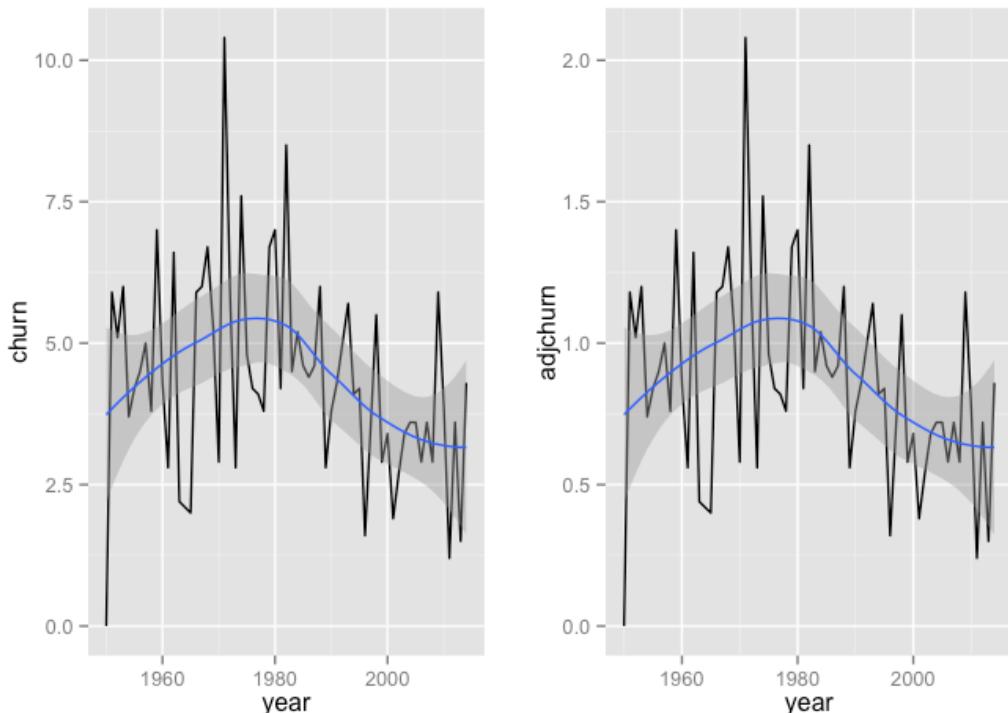
A similar effect is suggested when we consider the churn amongst points scoring drivers (above) and top 10 placed drivers at the end of each year (below).

```

season.churn.top10.df = churner( seasons[seasons['pos']<11,] )

g1 = ggplot(season.churn.top10.df,aes(x=year,y=churn))+geom_line()+geom_smooth()
g2 = ggplot(season.churn.top10.df,aes(x=year,y=adjchurn))+geom_line()+geom_smooth()
grid.arrange(g1, g2, ncol=2)

```



Churn in the top 10 placed drivers of the Drivers' Championship standings, 1950-2014

Note how the adjusted churn measure is very obviously broken in this model, with the maximum value exceeding the desired upper limit for this indicator of 1.0. This arises because the absolute position change values, calculated as a difference from a championship position possibly outside the top 10 from a previous year, may be greater than the number of competitors (10) used to calculate the adjusted churn value. It is left to further work or explore whether these filtered population measures are useful, and if so how to modify the adjusted churn metric so that it stays within the desired 0..1 bounding limits.

Despite possible issues with certain filtered variants of the adjusted churn measure, all the charts would seem to suggest that competitiveness has been decreasing amongst the drivers. But does the same story hold true of competitiveness amongst the teams? In the next subsection we'll look at churn across the Constructors Championship standings.

Year-on-year Churn - Constructors' Championship

The method for calculating the churn across seasons based on the Constructors' Championship closely follows that used for the Drivers' Championship.

For example, start by grabbing down the end-of-season constructor standings and then calculate the difference year-on-year for each team.

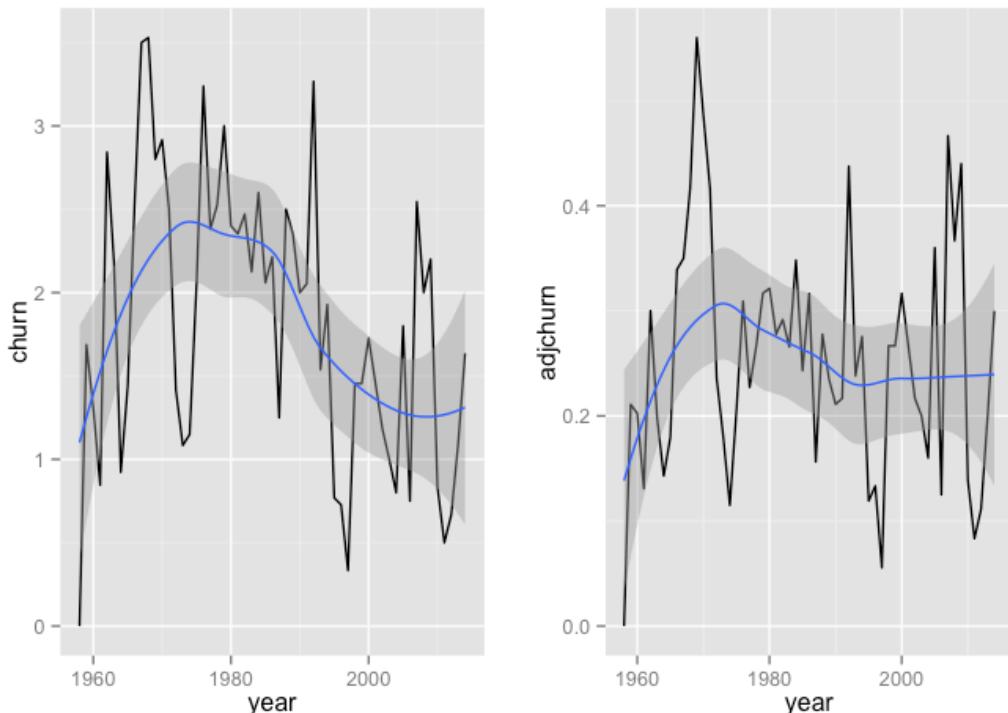
```
constructorSeasons=data.frame()
#Earliest 1958
for (year in seq(1958,2014,1)) {
  constructorSeasons=rbind(constructorSeasons,constructorStandings.df(year))
}

constructorSeasons=ddply(constructorSeasons,.(constructorId),transform,
                        delta=c(0,abs(diff(pos))))
```

Once again, take note that this approach calls the ergast API quite hard in a short period of time.

```
constructor.churn.df = churner( constructorSeasons )

g1 = ggplot(constructor.churn.df,aes(x=year,y=churn))+geom_line()+geom_smooth()
g2 = ggplot(constructor.churn.df,aes(x=year,y=adjchurn))+geom_line()+geom_smooth()
grid.arrange(g1, g2, ncol=2)
```



Churn in Constructors' Championship standings, 1958–2014

In contrast to the Drivers' Championship, competitiveness as measured by churn appears to have peaked in the mid-1970s, several years earlier than the peak in driver competitiveness, declining until the mid-1990s and then (on trend at least) remaining relatively consistent ever since. However, the wild swings in churn values in recent years suggests the story is not so simple...



Exercise - Do Rule Changes Have Any Influence on Competitiveness?

Try to find a source of information about major technical rule changes in Formula One over the year. (A useful starting point may be the Wikipedia page History of Formula One regulations⁶³). Do periods of significant change appear to have an influence on competitiveness as measured by churn in championship standings?

⁶³http://en.wikipedia.org/wiki/History_of_Formula_One_regulations

One final thing to note is that teams evolve, occasionally changing name whilst still retaining the same personnel, factory, design philosophy and racing DNA. However, such changes are likely to *increase* the churn indicator value, since they force change into rankings even if there has been no real change...

Taking it Further

The following ideas may be used to develop the notion of churn in the F1 context a bit further.

Calculating Adjusted Churn Within a Race

Although originally defined by Mizak et al. (2007) as a measure of competitive balance moving from one season to the next, and appropriated by Berkowitz et al. (2011) to track the evolution of competitive balance across a season, we might also explore the extent to which churn and adjusted churn might be used to provide an indication of the competitive evolution of a race itself, based on the position of each driver at the end of each lap.

Churn In Drivers' Careers

To what extent might we apply the churn measure to the standings achieved by a particular driver during their career? Does the churn indicator tend to suggest changes in fortune when a driver changes teams or acquires a new team mate?

Summary

In this chapter, we have explored the notion of churn in standings at the individual and constructor level in a variety of contexts, including within a season and across seasons. Churn is often used as an indicator within models of audience interest (not explored here), although it remains to be seen whether we might also use it as an indicator of *interestingness* when it comes to identifying stories about various forms of competitiveness within F1.

Laps Completed and Laps Led

One informal metric of success used to rank drivers' race performances is a tally of the number of laps on which they have been race leader, either given as a raw count or as a percentage of the number of racing laps they have completed either within a season or across their career.

Using the `lapTimes` table in the `ergast` database, which records race position, as well as lap time, for each driver on each lap of each race from 2012 onwards, we can easily generate several different sorts of "laps led" count.

To begin with, let's consider the number of laps on which each driver has been in the lead *at any stage of a race*. We'll calculate these lead lap counts for all the seasons for which we have laptime data, and also report the total number of lead laps for each driver as a percentage of the number of laps they completed (that is, the number of laps we have lap times for).

- the `Laps` count is a count of the total number of laps completed by each driver;
- the `lapsled` value is effectively a count of the the number of laps where the driver was in first position;
- the `lapsled_completed_pc` column represents the percentage of laps completed where the driver was in first position. That is, $\text{lapsled_completed_pc} = 100 * \text{lapsled} / \text{Laps}$ given as a percentage.

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

q=paste('SELECT d.driverRef,d.code,
          COUNT(l.lap) Laps,
          SUM(l.position==1) lapsled,
          (100*SUM(l.position==1))/COUNT(l.lap) lapsled_completed_pc
       FROM drivers d JOIN lapTimes l JOIN races r
      WHERE year="2012"
        AND r.raceId=l.raceId
        AND d.driverId=l.driverId
        GROUP BY d.driverId ORDER BY lapsled DESC')
lapsled=dbGetQuery(ergastdb,q)
```

driverRef	code	Laps	lapsled	lapsled_completed_pc
vettel	VET	1162	368	31
hamilton	HAM	1045	229	21
alonso	ALO	1095	216	19
button	BUT	1105	136	12
webber	WEB	1131	66	5
rosberg	ROS	1036	48	4

The effective count of laps led works as follows: if a driver is in first position at the end of any lap, `1.position==1` evaluates TRUE. This logical value also evaluates as a numerical value of 1. For other positions, the equivalence test fails, returning a logical FALSE value, which has a numerical value of 0. For each driver (GROUP BY `d.driverId`), we can thus count the number of laps they were leading on as the sum of the laps in which a test of their position equal to 1 was true: `SUM(1.position==1)`. The total number of laps they completed is simply a count of the number of laps recorded for that driver (`COUNT(1.lap)`).

We have already calculated one percentage value for each driver in the form of the percentage of laps completed that he was also leading on. But we could also calculate the percentage of the number of laps summed over the full race distances of the races he competed in that he was also leading on; that is, rather than dividing through by a count of the number of laps a driver completed, we divide through by the total race distance for each of the races the driver competed in.

This second interpretation suggests the following useful measure: *percentage of race laps completed = laps completed / total race distance of races competed in*. (A further possible metric is *average number of laps led per race*, although this is likely to be a highly variable amount.)

We shall see in the following section how to calculate the percentages based on the total race lap counts.

The lack of historical laptime data in the *ergast* dataset, in which laptime data is only available from the start of the 2011 Championship season, makes it impossible to calculate career history counts of drivers who started their Formula One career *before* 2011 using this resource. However, given the data we *do* have, we can explore leader lap counts for particular races or across a particular season, for example, as an indicator of competitive balance, or across the last few seasons. We can also compare leader lap counts across the years for particular circuits, for example to get a crude indication of whether or not a track seems to result in processional races or to get a feel for whether one particular driver or another has dominated on a particular track. In the case of a specific race, by supplementing

a simple laps led counts by a churn analysis or streak length comparison, we may be able to get a much richer view of the race dynamics.

Calculating Laps Completed and Laps Led Percentages

To calculate percentages relative to race distances, we need access to the race distance (in laps) of each race. We can get this information from the `results` table, by looking up the number of laps completed by the winner of each race. In particular, we generate an “as if” datatable containing two columns that pair a `raceId` with the number of race laps, given as the number of laps completed by the race winner:

```
q='SELECT raceId, laps AS racelaps FROM results WHERE position=1 LIMIT 3'
dbGetQuery(ergastdb, q)
```

```
##   raceId racelaps
## 1      18      58
## 2      19      56
## 3      20      57
```

We can use this value in our calculation of the percentage rates for the number of laps a driver completed, as well as the number of laps they led, taken over the total race distances of the races they competed in.

Putting all those elements together, we get a single query of the form:

```
q='SELECT circuitRef, d.code Code, year,
       racelaps AS Racelaps,
       COUNT(l.lap) Laps,
       SUM(l.position==1) AS lapsled,
       (100*SUM(l.position==1))/COUNT(l.lap) lapsled_comp_pc,
       (100*SUM(l.position==1))/RaceLaps AS lapsled_tot_pc
  FROM drivers d, lapTimes l, races r, circuits c,
       (SELECT raceId, laps AS racelaps FROM results WHERE position=1) rs
 WHERE r.raceId=l.raceId
       AND d.driverId=l.driverId
       AND r.raceId=rs.raceId AND c.circuitId=r.circuitId
       GROUP BY r.circuitId,d.driverId,year ORDER BY lapsled DESC'
cct_driver_lapsled=dbGetQuery(ergastdb, q)
```

This query dynamically creates an “as if” table, `rs`, that describes the `raceId` and number of race laps recorded for the winner of each race (`position=1`). This is the *de facto* race length for each race.

circuitRef	Code	year	Racelaps	Laps	lapsled	lapsled_comp_pc	lapsled_tot_pc
suzuka	VET	2013	53	53	22	41	41
sepang	VET	2013	56	56	21	37	37
buddh	WEB	2013	60	39	21	53	35
marina_bay	HAM	2012	59	22	19	86	32
yas_marina	HAM	2012	55	19	19	100	34
shanghai	VET	2011	56	56	18	32	32
catalunya	ALO	2011	66	65	17	26	25

In the table above, we see in the case of Hamilton and Webber how a driver may have led on a high percentage of completed laps, but a far lower percentage when the comparison is made relative to the total number of race laps for the races each driver competed in. For example, at Marina Bay in 2012, Lewis Hamilton completed 22 laps of a 59 lap race, leading for 19 of them, to give an 86% laps led percentage for the laps he completed, and 32% for the whole race.

We can also calculate the percentage over the whole of a season (that is, the ratio of the number of laps led by a driver in a particular season relative to the total number of race laps in that season). Again, we can calculate these percentages based on the total number of race laps (the sum of race distances), or the number of laps actually completed by the driver.

```
q=paste('SELECT d.code Code, rt.year as Year,
          racelapstot,
          COUNT(1.lap) AS Laps,
          (100*COUNT(1.lap))/racelapstot AS laps_pc,
          SUM(1.position==1) AS lapsled,
          (100*SUM(1.position==1))/COUNT(1.lap) lapsled_comp_pc,
          (100*SUM(1.position==1))/racelapstot AS lapsled_tot_pc
        FROM drivers d, lapTimes l, races r,
              (SELECT raceId, laps AS racelaps FROM results WHERE position=1) rs,
              (SELECT year, SUM(laps) AS racelapstot FROM results, races
                WHERE results.raceId=races.raceId AND position=1 GROUP BY year) rt
        WHERE r.raceId=l.raceId
          AND d.driverId=l.driverId
          AND r.raceId=rs.raceId
          AND rt.year=r.year')
```

```
GROUP BY d.driverId,r.year ORDER BY lapsled DESC')
cct_driver_lapsled_season=dbGetQuery(ergastdb,q)
```

Code	Year	racelapstot	Laps	laps_pc	lapsled	lapsled_comp_pc	lapsled_tot_pc
VET	2011	1133	1079	95	739	68	65
VET	2013	1131	1120	99	684	61	60
VET	2012	1192	1162	97	368	31	30
HAM	2012	1192	1045	87	229	21	19
ALO	2012	1192	1095	91	216	19	18
HAM	2011	1133	1013	89	150	14	13
BUT	2012	1192	1105	92	136	12	11
ROS	2013	1131	1058	93	104	9	9
ALO	2013	1131	1076	95	89	8	7
BUT	2011	1133	1095	96	88	8	7

Sorting by the percentages, we get a feel for the extent to which a particular season was dominated by any particular driver: in 2011 and 2013 Sebastian Vettel was in the lead *for over 60% of the total number of racing laps*. He also showed great reliability, completing 95% of all race laps in 2011 and 99% in 2012.



Exercise - Reliability Charts

Another way of using lap completed counts is to use them as a measure of *reliability*.

Use the lap data to generate a bar chart showing the number of race laps completed by driver or driver grouped by team, as a percentage of race distance, to visualise many racing laps were completed by each of them across a season.

Comparing laps led counts over seasons

If we view the laps led counts as indicators of competitiveness, we might want to explore how these values are distributed across seasons (to see whether the races in one season appear more or less competitive than another) or across circuits (for example, to see whether particular circuits appear to support races where there are likely to be several drivers with a significant lead lap count in each race, howsoever distributed, or whether the races appear to have a single leader throughout most of the race).

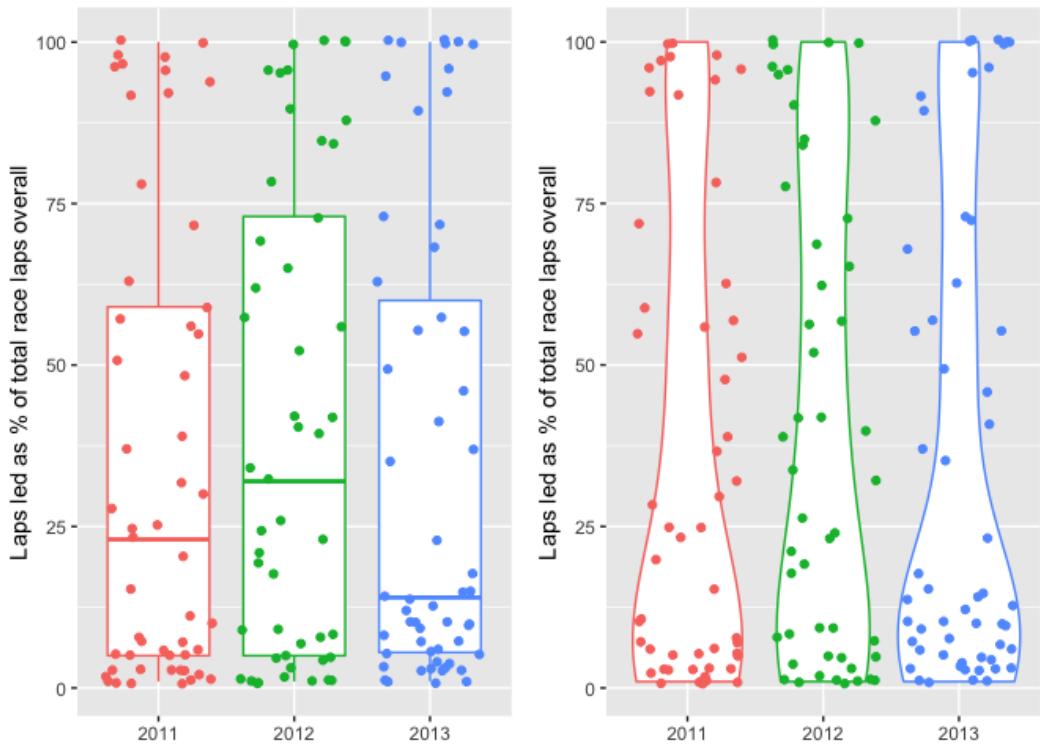
In the first case, we can plot a distribution of all laps led percentages for each of the races over the course of a season, ignoring the zero percentage values for people who led no laps in any particular race.

```
library(ggplot2)
library(gridExtra)

g1=ggplot(cct_driver_lapsled[cct_driver_lapsled['lapsled']>0,],
          aes(x=factor(year), y=lapsled_tot_pc, col=factor(year)))
g1=g1+geom_boxplot()+geom_jitter()+guides(colour=FALSE)
g1=g1+xlab(NULL)+ylab('Laps led as % of total race laps overall')

g2=ggplot(cct_driver_lapsled[cct_driver_lapsled['lapsled']>0,],
          aes(x=factor(year), y=lapsled_tot_pc, col=factor(year)))
g2=g2+geom_violin()+geom_jitter()+guides(colour=FALSE)
g2=g2+xlab(NULL)+ylab('Laps led as % of total race laps overall')

grid.arrange(g1, g2, ncol=2)
```



Distribution of laps led count percentages for all races over several seasons as a box plot and a violin plot

The chart on the left is a box plot, overlaid by a scatter plot of laps led counts. The central bar in the box plot is the median laps led count. The chart on the right is a *violin plot*, a shaped box plot whose contours reflect the actual shape of the distribution of individual points. (The width of the violin plot at a given value on the y-axis reflects the extent to which the original data values have similar values of y. That is, where there is a large number of points with a particular y-value or value thereabouts, the plot will be wide; where there are few values, it will be narrow.)

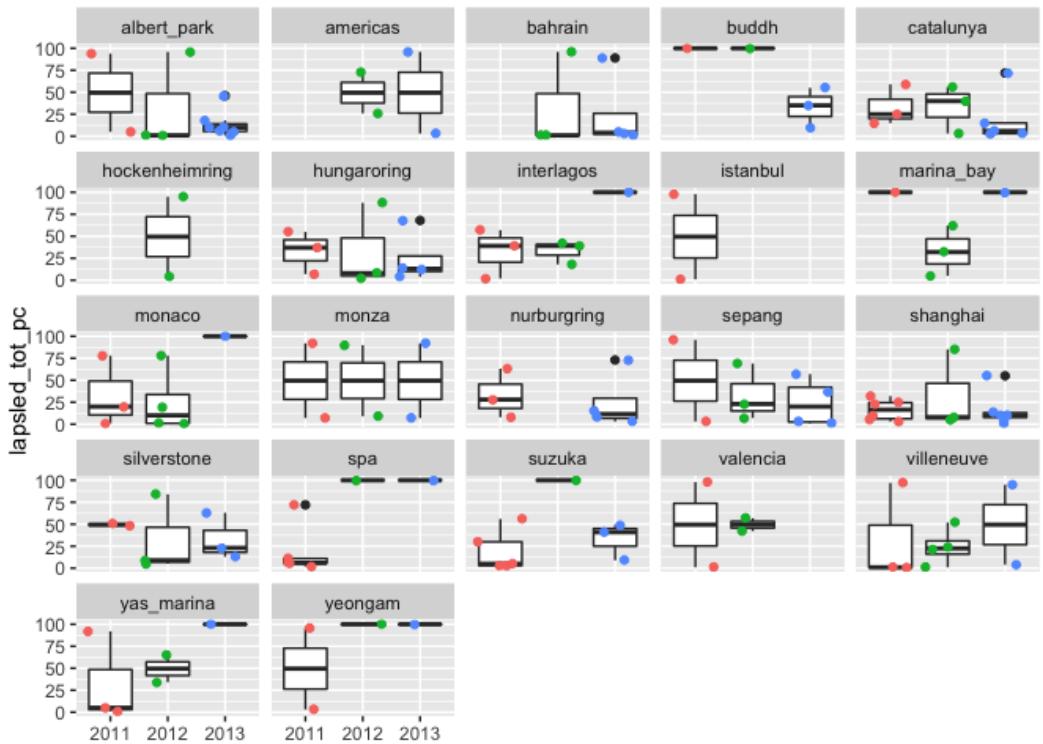
For the current example, the resulting charts are not particularly informative - it is difficult to get even a gut feeling about the relative competitiveness of the different years, if any, over this period, albeit a period in which we know that Red Bull dominated.

Comparing Laps Led Counts for Specified Circuits Across Several Years

An alternative approach is to further segment the data to see whether we spot any structure at a more refined level. For example, can we detect any differences in the likelihood of races being dominated by a single driver across circuits? That is, can we identify circuits where the race lead is historically unlikely to change over the course of a race, compared to circuits where the leadership does seem to be subject to change?

Let's start by looking at the distribution of lead lap counts for drivers who led any particular race for at least one lap. Although there is only a small number of points per track per year, a boxplot view emphasises the distribution of the points.

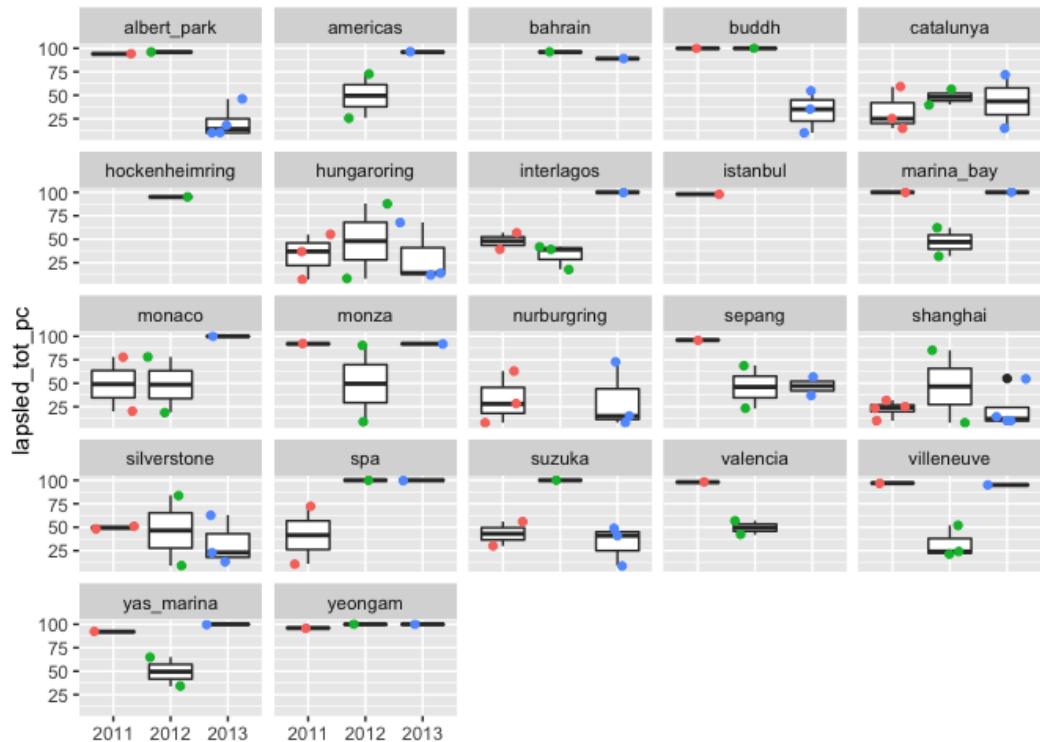
```
g = ggplot(cct_driver_lapsled[cct_driver_lapsled['lapsled']>0,],  
           aes(x=factor(year), y=lapsled_tot_pc))  
g = g + geom_boxplot() + geom_jitter(aes(col=factor(year)))  
#Facet the chart to show the distributions for separate races  
g = g + facet_wrap(~circuitRef)  
g + xlab(NULL) + guides(colour=FALSE)
```



Faceted scatterplot view of laps led counts by circuit over several years.

We can tighten up this graphic by limiting the display of drivers to show only those who led for at least five laps in any particular race.

```
g = ggplot(cct_driver_lapsled[cct_driver_lapsled['lapsled'] >= 5, ],
            aes(x=factor(year), y=lapsled_tot_pc))
g = g + geom_boxplot() + geom_jitter(aes(col=factor(year)))
g + facet_wrap(~circuitRef) + xlab(NULL) + guides(colour=FALSE)
```



Filtering the drivers to those who led for at least five laps, we get a more powerful graphic.

Statistically, this is not really an appropriate way to use the box plot because the number of scatter points used to calculate the box limits is too small. However, *pragmatically*, the boxes do provide a glanceable summary of how the laps led counts are distributed for each track in each particular year.

In particular, the charts suggest that if we want to see a race where the lead is not likely to be pre-determined for much of the race, Silverstone, Catalunya, the Hungaroring and Shanghai may be places to go, and Yeongam one to avoid.

One thing these charts do not indicate are how processional these races may or may not have been. For example:

- *in terms of race leader*: if two cars each led for 50% of a race, how was that distributed? Lap leader streakiness would help us identify that;
- *in terms of the general processional nature of a race*: an analysis of race position churn over the course of a race could start reveal how positions had changed throughout the

course of a race.

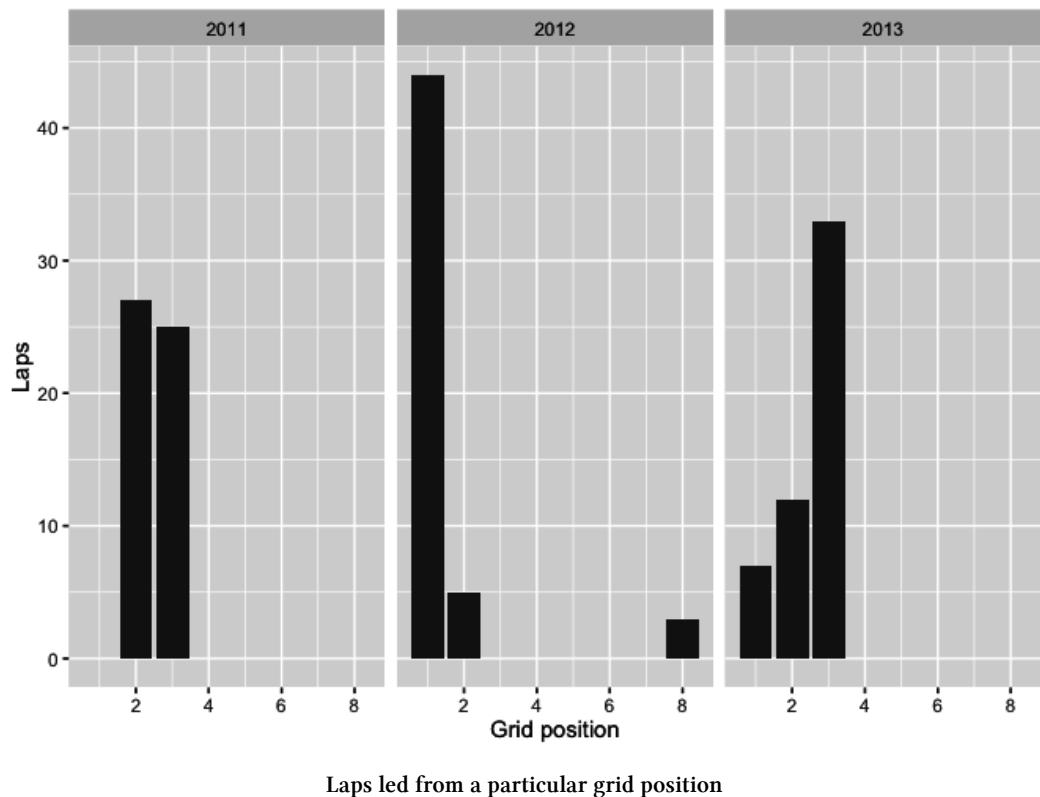
Laps Led From Race Position Start

One final analysis we might run is to look at the distribution of counts of laps led against the original grid position, again splitting these down by circuit. To do this, we need to identify drivers who led on at least one lap, and also identify their grid position.

Finding the counts is straightforward enough:

```
q='SELECT circuitRef, grid, year, COUNT(l.lap) AS Laps
    FROM (SELECT grid, raceId, driverId from results) rg,
        lapTimes l, circuits c, races r
    WHERE c.circuitId=r.circuitId AND rg.raceId=l.raceId
        AND rg.driverId=l.driverId AND l.position=1 AND r.raceId=l.raceId
    GROUP BY grid, circuitRef, year
    ORDER BY year'
lapsledfromgridposition=dbGetQuery(ergastdb,q)

g = ggplot(lapsledfromgridposition[lapsledfromgridposition['circuitRef']=='silverston\\
e',])
g = g + geom_bar(aes(x=grid,y=Laps,group=factor(year)),
                  stat='identity',position='dodge')
g + facet_wrap(~year) +xlab("Grid position") + ylab("Laps")
```



What this chart shows is how many laps were led by a driver starting in a particular grid position at Silverstone for the years 2011-2013. In 2011, the cars starting from 2nd or 3rd position dominated the laps led counts. In 2012, the car on pole took most of the led laps. In 2013, the cars on the front row of the grid both led for part of the race, but the driver starting from third led most laps.

Exercise - Laps Led By Grid Position Time Series



One of the disadvantages of laps led *counts* is that they don't provide any information about the way the leader laps are *distributed* by driver across the course of a race: a race in which drivers are frequently changing the lead may have the same laps led profile as one in which a different driver dominates different parts of the race.

How might you visualise the laps led by driver distribution over the course of a race?

One way might be to set a lap leader flag (set to 1, or TRUE when a driver led a lap, 0 or FALSE otherwise) for each driver, by lap number, and then plot this against lap.

The driver axis could arrange drivers sorted by grid position and offer two values (lap led / not led), tracing out a binary "timing diagram" style trace for each driver/grid position, showing the laps they led.

To make fair comparisons across races, we should use the percentage of race laps led by dividing though by the number of laps from each race (and making life easier for ourselves by reusing parts of the queries we have already developed, such as reusing the 'as if' table that reported the number of race laps):

```
q='SELECT circuitRef, grid, year, COUNT(1.lap) AS Laps,
       (100*COUNT(1.lap))/racelaps AS laps_pc
    FROM (SELECT grid, raceId, driverId from results) rg,
         (SELECT raceId, laps AS racelaps FROM results WHERE position=1) rs,
         lapTimes l, circuits c, races r
   WHERE c.circuitId=r.circuitId AND rg.raceId=l.raceId AND rs.raceId=rg.raceId
         AND rg.driverId=l.driverId AND l.position=1 AND r.raceId=l.raceId
  GROUP BY grid, circuitRef, year
  ORDER BY year, round'
lapsledfromgridpositionpc=dbGetQuery(ergastdb,q)
```

```
g = ggplot(lapsledfromgridpositionpc)
g = g + geom_text(aes(x=grid, y=circuitRef, label=laps_pc, size=laps_pc))
g + facet_wrap(~year) + xlab(NULL) + ylab(NULL) + guides(size=FALSE) + xlim(0,20)
```

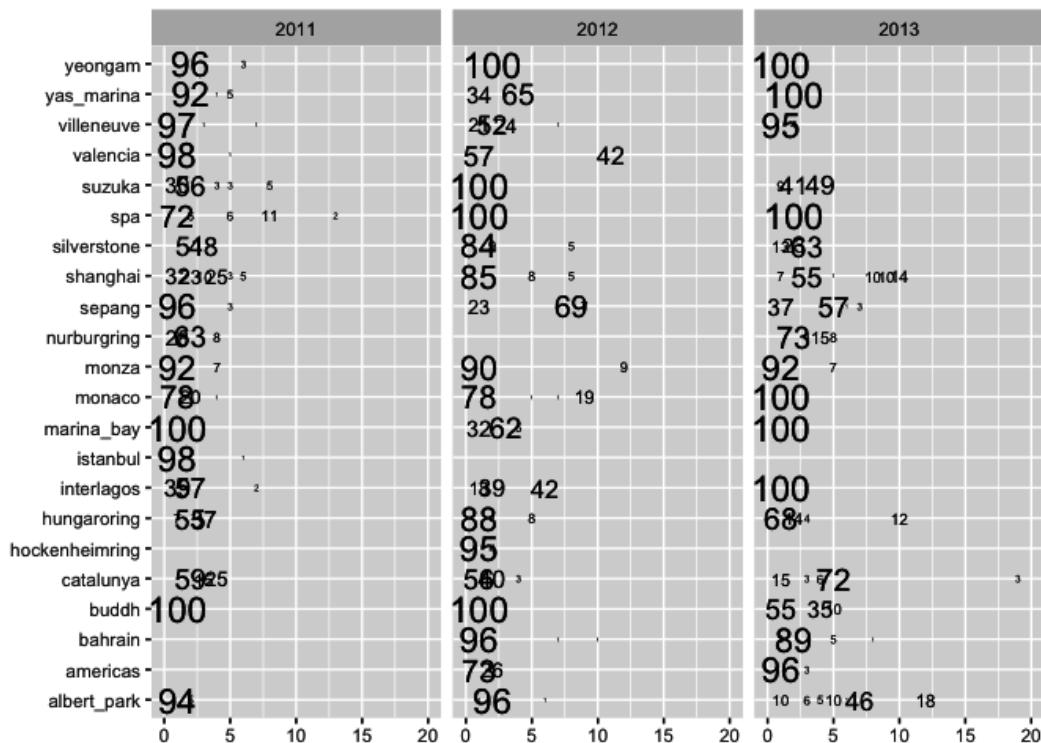


Table showing percentage of laps led per grid position by circuit as a percentage of total race laps

This chart is far from ideal, not least because it is difficult to see race position dominated because of the compressed nature of the x-axis. It might also be improved by sorting the y-axis in a more natural way, rather than the default reverse alphabetical ordering. For example, it is not clear whether races come to be more or less uncertain over the course of each season. If we ordered the circuits by round in each season, it would be easier to see whether competitiveness changed over the season. This is easy enough to do if the circuits maintain their round number each year, because we can compare both round and circuit as the same thing. However, if circuits change which round they represent, organising the y-axis ordering appropriately so that we can compare circuit behaviour with season progress may become more challenging.



Improving the chart

The “laps led per grid position” text plot is rather scruffy. What problems in particular can you identify with it? How would you start to tidy it up?

Laps Led By Driver from Particular Grid Position Starts

We can also generate a report to show the simple count of laps led by each driver from a particular grid position:

```
q='SELECT code, grid, year, COUNT(l.lap) AS Laps
    FROM (SELECT grid, raceId, driverId from results) rg,
        lapTimes l, races r, drivers d
    WHERE rg.raceId=l.raceId AND d.driverId=l.driverId
        AND rg.driverId=l.driverId AND l.position=1 AND r.raceId=l.raceId
    GROUP BY grid, driverRef, year
    ORDER BY year, round'
driverlapsledfromgridposition=dbGetQuery(ergastdb, q)
```

To try to make it a little clearer to see what's going on from the front of the grid, we can use a *logarithmic* scale to provide a little more room at the left hand side of the chart to display the labels. Rotating the labels also helps to reduce overlap between the labels. To prevent the text from being pushed off the left hand side of the chart, we can *expand* the x-axis to prevent the labels overflowing the plot area. A faint dashed grey line at the "2.5" position on the logarithmic x-axis identifies marks out the cars starting out from the front row of the grid. Finally, the black and white theme makes the chart much cleaner.

```
g = ggplot(driverlapsledfromgridposition)
g = g + geom_vline(xintercept = 2.5, colour='lightgrey', linetype='dashed')
g = g + geom_text(aes(x=grid, y=code, label=Laps, size=log(Laps), angle=45))
g = g + facet_wrap(~year) + xlab(NULL) + ylab(NULL) + guides(size=FALSE)
g + scale_x_log10(expand=c(0,0.3)) + theme_bw()
```

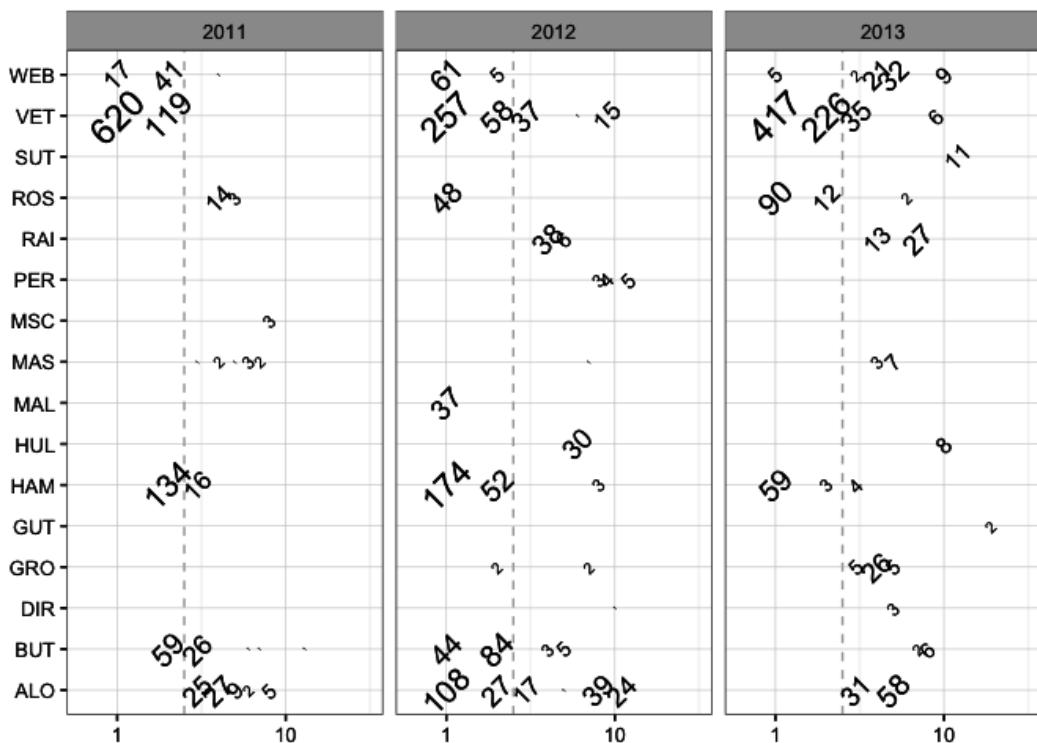


Table showing count of laps led per grid position by driver

Once again, we could probably improve the ordering of the y-axis, such as by ordering according to the percentage of laps led, compared to total racing laps, completed across all displayed years. And although using the “stretchy” logarithmic x-axis does not really “make sense” in statistical terms, (we aren’t trying to plot growth rates, for example), it does rather conveniently space out the text labels for the higher placed grid positions, which are likely to be the starting positions from which lap leaders are likely to come.



Exercise - Laps Led per Team

How would you display counts for the laps led by team and display it in an informative way? Remember, the results table contains details of the constructorId associated with each result, and the constructors table information about each constructor.

Summary

In this chapter, we have explored the distribution of laps led over a season, across particular circuits, and based on grid position. It is not immediately clear (to me at least) what the most appropriate way of displaying the information in a graphical form might be. The distribution and amount of data available meant that box plots and violin plots, combined with a scatterplot, did not really produce much insight. However, the text plot combined with the use of a logarithmic scale to “stretch out” the distance between counts from the head of the grid did seem to provide quite a rich, if rather contrived, way of presenting information.

Event Detection

When we look at a chart such as a lap chart, it is very clear to us when there is a change in position and which drivers were involved. But can we *automatically* detect which drivers have been involved in a series of position changes between each other over a particular series or laps?

Or consider the more complex race history chart: it's easy enough for us to see from the chart when drivers are close to each other in terms of their total race time up to any particular lap because their race lines are close to each other on the chart. But once again, can we *automatically* detect groups of drivers who are perhaps within a second or so of each other, increasing the chance of an overtake because of the likelihood that DRS will be enabled?

Detecting Position Change Groupings

Let's grab some example data to explore how we might automatically identify such groupings or clusters amongst the drivers:

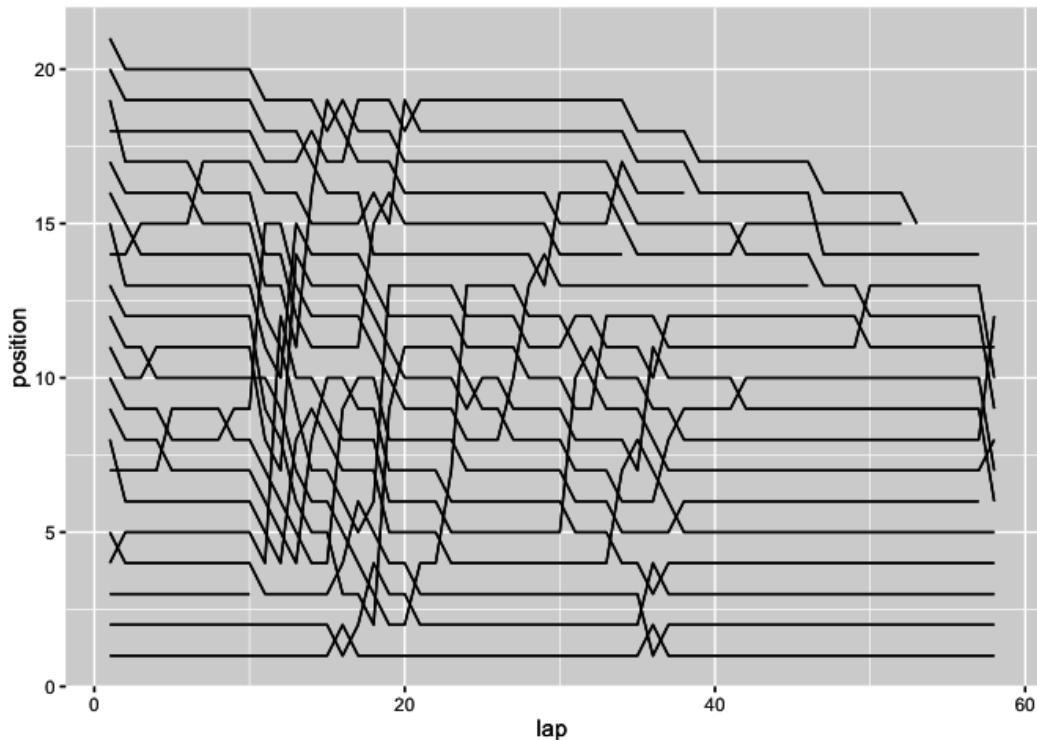
```
library(DBI)
ergastdb =dbConnect(RSQLite::SQLite(), './ergastdb13.sqlite')

#Get a race identifier for a specific race
lapTimes=dbGetQuery(ergastdb,
  'SELECT driverId, position, lap, milliseconds
   FROM lapTimes l JOIN races r
   WHERE l.raceId=r.raceId AND year="2012" AND round="1" ')
```

In the following lap chart sketch, each line represents the evolution of positions held by a particular driver over the course of a race. Changes in position are identified by crossed lines.

```
library(ggplot2)

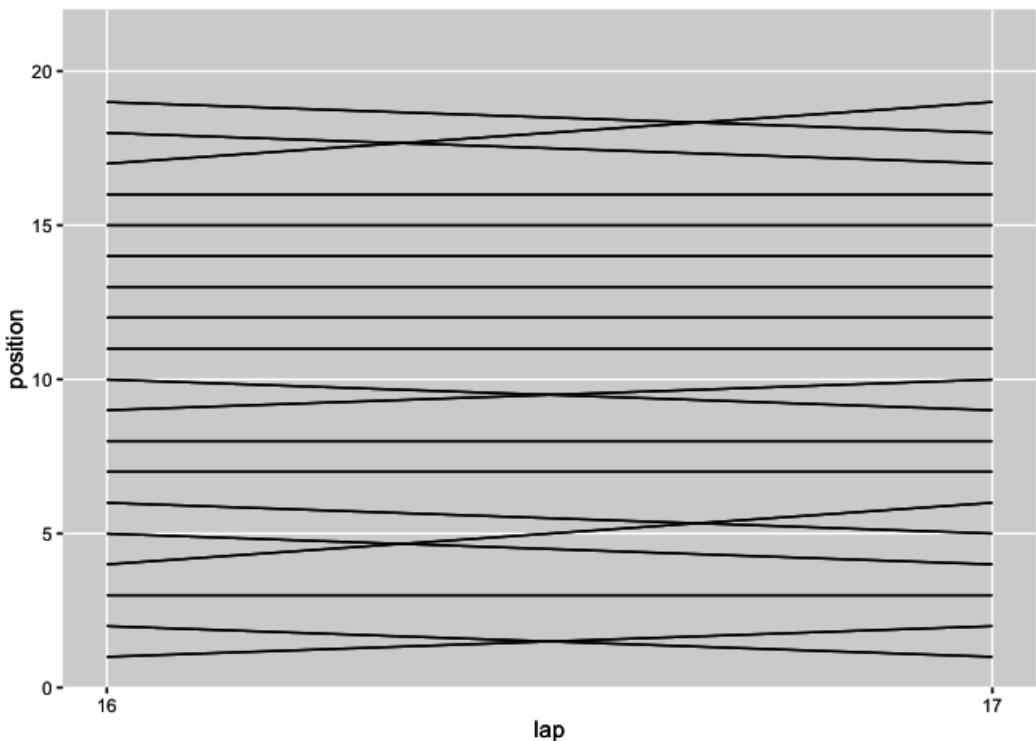
g=ggplot(lapTimes)+geom_line(aes(x=lap,y=position,group=driverId))
g
```



Example lap chart

If we zoom in, we can look in detail at the positions held - and that changed (the crossed lines) - over a couple of consecutive laps:

```
g=g+ scale_x_continuous(breaks = c(16,17),limits=c(16,17))
g+theme( panel.grid.minor = element_blank() )
```



Zooming in to a particular area of the lap chart - laps 16 and 17

Many of the drivers do not change position at all, but there are position changes between four distinct groups of drivers: those in 1st and 2nd; those in 4th, 5th and 6th; those in 9th and 10th; and those in 17th, 18th and 19th.

How can we identify the positions that are being contested during a particular window of time, where a contested position means that the particular position was held by more than one person in that particular window?

Let's create a couple of new data columns to help us work through how we might identify position changes for each driver between each consecutive lap: a column that identifies the driver's position in the *previous* lap (*pre*), and a column that describes their change in position for the current lap given the previous lap position (*ch*). (Note that we will get an end effect at the start of the race; we really should use grid position as the first entry rather than the dummy 0 value.)

Filter the augmented data set to show just those rows where there is a difference between a driver's current position and previous position (the first column in the result just shows row

numbers and can be ignored).

```
library(plyr)

#Sort by lap first just in case
lapTimes=arrange(lapTimes,driverId,lap)

#Create a couple of new columns
#pre is previous lap position held by a driver given their current lap
#ch is position change between the current and previous lap
tlx=ddply(lapTimes,
           .(driverId),
           transform,
           pre=(c(0,position[-length(position)])),
           ch=diff(c(0,position)))

#Find rows where there is a change between a given lap and its previous lap
llx=tlx[tlx['ch']!=0 & tlx['lap']==17,c("pre","position")]

arrange(llx,pre,position)

##      pre position
## 1      1        2
## 2      2        1
## 3      4        6
## 4      5        4
## 5      6        5
## 6      9       10
## 7     10        9
## 8     17       19
## 9     18        17
## 10    19       18
```

From the data table, we see distinct groups of individuals who swap positions with each other between those two consecutive steps. So how can we automatically detect the positions that these drivers are fighting over?

An answer⁶⁴ provided in response to a Stack Overflow question on how to get disjoint sets from a list⁶⁵ gives us a nice solution: represent the connections between previous and current

⁶⁴<http://stackoverflow.com/a/25130575/454773>

⁶⁵<http://stackoverflow.com/questions/25130462/get-disjoint-sets-from-a-list-in-r>

lap positions as connected nodes in a graph (that is, a network), and then find the connected components. These components will identify the positions where there was a position change.

Graph Theory

Graph Theory is a branch of mathematics that focuses on ways of describing and analysing networks. A great example is the map of the London Underground network: tube stations are represented as *nodes* in the network, and are connected together by lines referred to as *edges*.

Graphs (that is, *networks*) can be either *directed* or *undirected*. In a *directed graph*, the edges are represented using arrows and as such have a direction, going *from* one node *to* another. In an *undirected graph*, we represent edges using a simple line that has no direction - it just shows that two nodes are connected to each other.

As well as direction, edges may also have a *weight* associated with them, often represented visually by the thickness of the line. The weight identifies the strength of the connection between the two nodes. Graph theory provides a very powerful and elegant way of representing relations between things in the world, and, as in the example described here, can be used in many not immediately obvious, yet still surprisingly useful, ways.

Colour may also be applied to nodes and edges in order to highlight additional channels of information.

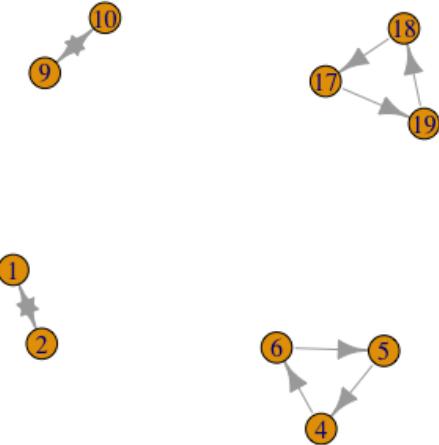
The *igraph* R package contains a wide variety of tools for constructing, analysing and visualising networks and building up your own graph based representations.

Here's my working of the solution. Each row of the previously shown list of results that shows how a particular driver held different positions going from one lap to the next for a specific lap.

We can now create a graph in which nodes represent positions (that is, current lap (*position*) or previous lap (*pre*) values) and edges connect current and previous positions. If we plot the resulting graph, we notice that it is split into several components:

```
#install.packages("igraph")
#http://stackoverflow.com/a/25130575/454773
library(igraph)
```

```
##  
## Attaching package: 'igraph'  
  
## The following objects are masked from 'package:stats':  
##  
##     decompose, spectrum  
  
## The following object is masked from 'package:base':  
##  
##     union  
  
par(bg="white")  
plot(graph.data.frame(l1x))
```



Network diagram showing position changes across two laps.

Notice how the nodes - representing positions - are connected to each other by arrows, showing how a car placed in one position moved to another position going from one particular lap to the next. So for example, we see that the cars in positions 9 and 10 changed place with each other, as did those in positions 1 and 2. The car in 19th went to 18th, the one in 18th to 17th, and the one in 17th fell back to 19th. And so on.

Visually, it's clear to us that the connected nodes form different groups, or clusters. The question we are now faced with is how to identify these distinct components of connected nodes *automatically*? Fortunately, there are well known methods for calculating such things, and the `igraph` package makes it easy for us to call on them:

```
posclusters = function(edg) {
  g = graph.data.frame(edg)
  split(V(g)$name, clusters(g)$membership)
}

posGraph=posclusters(l1x)
```

This gives the following clusters, and their corresponding members:

```
#Find the position change battles
for (i in 1:length(posGraph))
  print(posGraph[[i]])

## [1] "1" "2"
## [1] "18" "19" "17"
## [1] "10" "9"
## [1] "4" "5" "6"
```

How might we generalise this approach? Here are some things that came to my mind:

- allow a wider window within which to identify battles (this means we should be able to look for clusters over groups of three or more consecutive laps);
- simplify the way we detect position changes for a particular driver; for example, if we take the set of positions held by a driver within the desired window (laps M to $M+3$, for example), if the size of the set (that is, its *cardinality*) is greater than one, then we have had at least one position change for that driver within that window. Each set of unique positions held by different drivers that has more than one member can be used

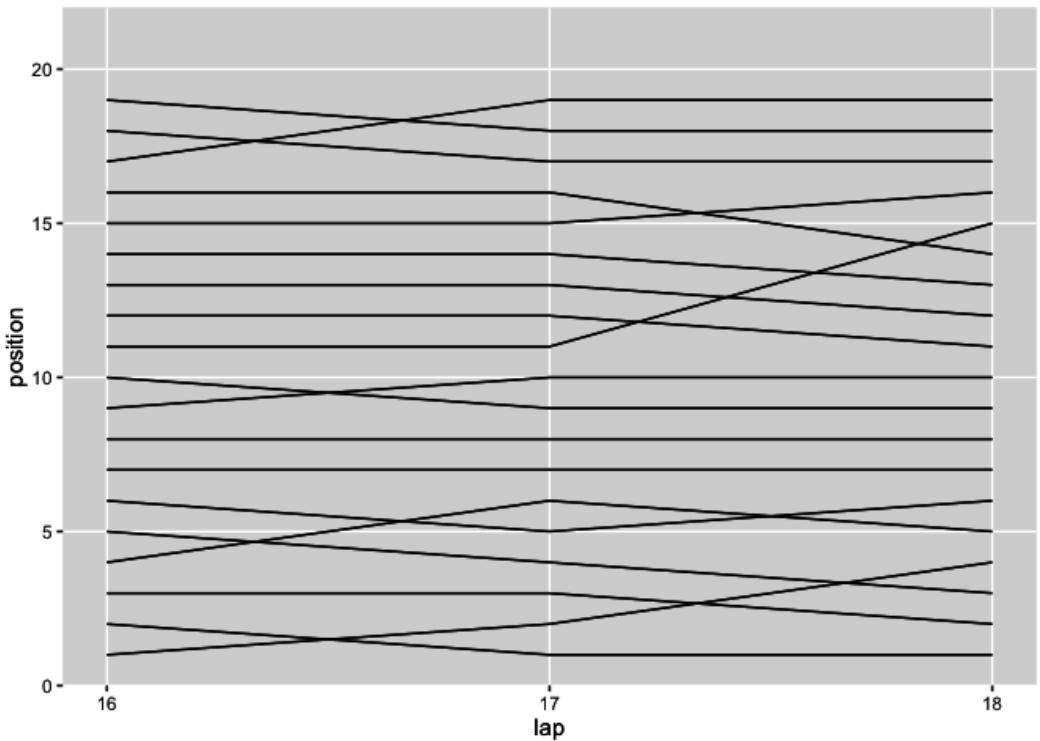
to generate a set of distinct, unordered pairs that connect the positions (I think it only matters that they are connected, not that a driver specifically went from position x to position y going from one lap to the next?). If we generate the graph from the set of distinct unordered pairs taken across all drivers, we should then be able to identify the contested/driver change position clusters.

One downside of this approach, as applied over several laps, would be that we might lose information about each particular position change, or lose resolution about repeated changes of position back and forth.

Extending the Approach to Larger Lap Windows

Rather than try to extend the approach as suggested above, let's just see how well the original method fares when we consider position changes over three laps rather than two. Here's an example of several sorts of position change that are possible within that time frame:

```
g=g+ scale_x_continuous(breaks = c(16,17,18),limits=c(16,18))  
g+theme( panel.grid.minor = element_blank() )
```



A view of position changes over laps

As before, we identify rows corresponding to drivers who experienced a position change going from one lap to another, in this case using data for laps 17 and 18 (thus including lap 16, the “pre” lap of column 17):

```
l1x3=tlx[tlx['ch']!=0 & (tlx['lap']==17 | tlx['lap']==18),c("pre","position")]
arrange(l1x3,pre,position)
```

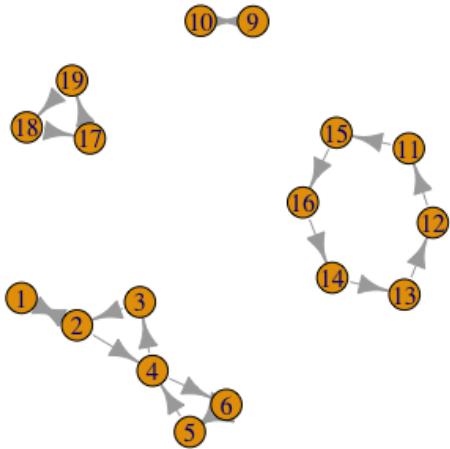
```
##      pre position
## 1      1      2
## 2      2      1
## 3      2      4
## 4      3      2
## 5      4      3
## 6      4      6
## 7      5      4
## 8      5      6
## 9      6      5
## 10     6      5
## 11     9     10
## 12    10      9
## 13    11     15
## 14    12     11
## 15    13     12
## 16    14     13
## 17    15     16
## 18    16     14
## 19    17     19
## 20    18     17
## 21    19     18
```

To generalise this approach, it is tempting to create a function that uses data from with a lap window specified by minimum and maximum lap values that can be used as a the basis of a sliding window applied across the race and from which “eventful” laps might be automatically identified.

The method might also be applied around laps that are likely to be eventful, such as the laps around which cars of interest enter the pits.

Visualising this data again as a *directed graph*, with edges (arrows) going *from* the previous lap position *to* the current lap position, we notice how the various graph components describe what position changes took place within the window:

```
par(bg="white")
plot(graph.data.frame(11x3))
```



Network diagram showing position changes across three laps.

So for example, at some point in the window, we see that the cars in positions 9 and 10 swapped places at least once; we also see how the cars in 1st and 2nd swapped places, and then the car in 2nd fell back to 4th; at some point a car in 4th place car moved to 3rd and the 3rd placed car to 2nd; at possibly other points the car in 4th fell back, a car in 6th moved up to 5th and a car in 5th moved up to 4th. Just *when* these changes took place, or even the order in which they took place, we can't necessarily tell from the diagram.

Unlike the simple 2 lap case then, where we can read off the actual position changes, when we try to use this technique to visualise positions changes across 3 or more laps, we quickly lose track of what actually happened when. What we do know, though, is which positions were being contested and which changed hands in the course of the selected laps.

```
posGraph3=posclusters(11x3)
for (i in 1:length(posGraph3))
  print(posGraph3[[i]])

## [1] "1" "2" "3" "4" "6" "5"
## [1] "18" "19" "17"
## [1] "15" "12" "13" "11" "16" "14"
## [1] "10" "9"
```

What actually happened?

By visual inspection of the position change network charts, particular the single lap example, (that is, where we compare the current position with the position from the previous lap) we see there may be several sorts of pattern of behaviour that we can quickly read off: two drivers changing position with each other for example, or two or more cars moving up a position as another car falls out of position.

Based on these observations, we might be able to probe actual events further by looking at other sources of data. For example, if we have access to pit stop data and we see that a car falls back several positions, if we notice that it had pitted but the cars that moved up position in the cluster did not, the chances are that it was the pit stop that was responsible for the position change. It might also be worth exploring ways of trying to detect what strategy a driver is likely to be on (in terms of numbers of stops, for example), or whether a driver appears to have changed strategy because of some event, from the online laptime data.

As well as detecting groupings based on position changes using the `ch` column derived from comparing current and previous lap positions, we can also identify a range of other flags to detect particular features. For example, if we order the drivers by lap and total cumulative lap time, we can calculate whether or not two drivers are within one second of each other at any particular lap, and as such whether DRS is likely to be available to the chasing (lower placed) driver. This can give us a set of graphs showing cars within likely DRS range of each other, and as such help us identify where possible battles may be about to take place.

Detecting Undercuts

One of the events that race commentators have taken to mentioning in recent years is an *undercut*, a tactical ploy described by F1 journalist James Allen in *The secret of undercut and offset*⁶⁶.

⁶⁶<http://www.ubs.com/microsites/formula1/en/james-allen/the-secret-of-undercut-and-offset.html>

An undercut occurs during a race when a Driver A, who is leading Driver B, is jumped by Driver B taking an early pit stop and a fresh set of tyres. As Driver A is ahead, he is unaware that the move is coming until it is too late to react and he has passed the pit lane entry.

On fresh tyres, Driver B drives a very fast outlap from the pits. Driver A will react to the stop and pit on the next lap, but his inlap time will have been set on old tyres, and will be correspondingly slower. As Driver A emerges from the pit lane after his stop, Driver B may be narrowly ahead of him into the first corner and an undercut has been achieved.

In logical terms, we might characterise this as follows:

- two drivers, d_1 and d_2 : $d_1 \neq d_2$;
- d_1 pits on lap X , and drives an outlap on lap $X+1$;
- d_1 's position on their pitlap (lap X) is greater than d_2 's position on the same lap X ; d_2 pits on lap $X+1$, with an outlap on lap $X+2$;
- d_2 's position on their outlap (lap $X+2$) is greater than d_1 's position on the same lap $X+2$.

We can generalise this formulation, and try to make it more robust, by comparing positions on the lap prior to d_1 's stop (lap A) with the positions on d_2 's outlap (lap B):

- two drivers, d_1 and d_2 : $d_1 \neq d_2$;
- d_1 pits on lap $A+1$;
- d_1 's position on their “prelap” (lap A), the lap prior to their pitlap (lap $A+1$), is greater than d_2 's position on lap A ; this condition tries to ensure that d_1 is behind d_2 as they enter the pit stop phase but it misses the effect on any first lap stops (unless we add a lap \emptyset containing the grid positions);
- d_1 's outlap is on lap $A+2$;
- d_2 pits on lap $B-1$ within the inclusive range $[1ap\ A+2, 1ap\ A+1+N]: N \geq 1$, (that is, within N laps of d_1 's stop) with an outlap on lap B ; the parameter, N , allows us to test for changes of position within a pit stop window, rather than requiring that d_2 pits on the lap immediately following d_1 's stop;
- d_2 's position on their outlap (lap B , in the inclusive range $[1ap\ A+3, 1ap\ A+2+N]$) is greater than d_1 's position on the same lap B .

One way of implementing these constraints is to write a declarative style query that specifies the conditions we want the solution to meet, rather than writing a procedural programme to

find such an answer. Using the `sqldf` package, we can use a SQL query to achieve just this result.

One way of writing the query is to create two situations, *a* and *b*, where situation *a* corresponds to a lap on which d1 stops, and situation *b* corresponds to the driver d2's stop. We then capture the data for each driver in each situation, to give four data states: d1a, d1b, d2a, d2b. These states are then subjected to the conditions specified above (using N=5).

```

source('ergastR-core.R')
#First get laptime data from the ergast API
lapTimes=lapsData.df(2015,9)

#Now find pit times
p=pitsData.df(2015,9)

#merge pitdata with lapsdata
lapTimesp=merge(lapTimes, p, by = c('lap','driverId'), all.x=T)

#flag pit laps
lapTimesp$ps = ifelse(is.na(lapTimesp$milliseconds), F, T)

#Ensure laps for each driver are sorted
library(plyr)
lapTimesp=arrange(lapTimesp, driverId, lap)

#Create an offset on the laps that are pitstops for each driver
#to set outlap flags for each driver
lapTimesp=ddply(lapTimesp, .(driverId), transform, outlap=c(FALSE, head(ps,-1)))

#Identify the lap before a pit lap by reverse sorting
lapTimesp=arrange(lapTimesp, driverId, -lap)
#So we can do an offset going the other way
lapTimesp=ddply(lapTimesp, .(driverId), transform, prelap=c(FALSE, head(ps,-1)))

#Now we can run the SQL query to find the undercuts
#Use a 5 lap window to try to identify undercut opportunities
library(sqldf)
ss=sqldf('SELECT d1a.driverId AS d1, d2a.driverId AS d2,
          d1a.lap AS A, d1a.position AS d1posA, d1b.position AS d1posB,
          d2b.lap AS B, d2a.position AS d2posA, d2b.position AS d2posB
     FROM lapTimesp d1a, lapTimesp d1b, lapTimesp d2a, lapTimesp d2b
    WHERE d1a.driverId=d1b.driverId AND d2a.driverId=d2b.driverId
      AND d1a.driverId!=d2a.driverId
')

```

```

AND d1a.prelap AND d1a.lap=d2a.lap AND d2b.outlap AND d2b.lap=d1b.lap
AND (d1a.lap+3<=d1b.lap AND d1b.lap<=d1a.lap+2+5)
AND d1a.position>d2a.position AND d1b.position < d2b.position')
)

```

d1	d2	A	d1posA	d1posB	B	d2posA	d2posB
hamilton	massa	18	3	2	21	1	3
hamilton	bottas	18	3	1	22	2	3
kvyat	hulkenberg	17	6	9	20	5	10
vettel	bottas	42	5	3	45	4	5
vettel	massa	42	5	3	45	3	4
vettel	kvyat	13	8	8	19	7	10
vettel	hulkenberg	13	8	7	20	6	10
alonso	ericsson	36	11	10	42	10	11
alonso	ericsson	36	11	10	43	10	11
ricciardo	sainz	10	11	12	13	10	13
merhi	stevens	43	13	12	46	12	13

With a five lap window we have evidence that supports successful undercuts in several cases, including VET taking KVY and HUL with his early stop at lap 13+1 (KVY pitting on lap 19-1 and HUL on lap 20-1), and MAS and BOT both being taken first by HAM's stop at lap 18+1 and then by VET's stop at lap 42+1. (To make things easier to read, we may instead define $d1a.lap+1$ AS $d1Pitlap$ and $d2b.lap-1$ AS $d2Pitlap$.)

The query doesn't guarantee that the pit stop was responsible for change in order, but it does at least gives us some prompts as to where we might look for further evidence that an undercut occurred.

Summary

In this chapter, we have started to explore the notion of detecting events within the data, focusing initially on the detection of position change events and the identification of groups of drivers (or at least, race positions) that were involved in a position change. By representing the position changes using a graph based representation, we were able to cluster race positions that were successfully contested and which changed hands between drivers.

It is possible that additional information - such as pit information - might be brought to bear to help start explain why a particular event, such as a position change, might have occurred.

The possibility of detecting other signals that could be used to help us identify events, or situations, within the race, such as groups of cars where each consecutive car is likely within

DRS range of the car ahead, was also identified.

Being able to identify tactical moves, such as undercuts, may be of some use when putting together race reports. As such, a method was explored for trying to identify potential undercuts within a rolling 5 lap window. The method itself was based on an SQL implementation, applied using the `sqldf` library to an R dataframe, derived from a logical description of an undercut situation.

Comparing Intra-Team Driver Performances

One of the strongest comparisons we can make about driver performances within F1, as within many areas of motorsport, is between drivers on the same team. Teams run cars of a similar specification and design, and performance data may be shared between the team members and their engineering teams. Where drivers within the same team are free to race, we can therefore compare their performances in order to gain some idea of relative ranking between them.

Over the years, drivers tend to move between teams, and join up with different team partners along the way. This provides us with an opportunity to start to piece together a ranking that can rate drivers relative to other specific drivers at particular stages of their respective careers to generate a more complete ranking. Such data may also be used in the compilation of “best driver ever” style statistics, such as John Burn-Murdoch’s Who are the best ever Formula One drivers?⁶⁷ rankings on the FT interactive graphics site, based on team and driver dominance statistics within a particular season, or Andrew Phillips’ Reconstructed History of Formula 1⁶⁸ on his *F1Metrics* blog. (*It is left as an exercise for the reader to implement those analyses using R!*)

In this chapter, we will start to explore how drivers within a particular team compare with each other over the course of a single season.

Intra-Team League Tables

Trivially, we can put together simple tables that compare the performances of drivers within a particular team on what we might refer to as pointwise mini-competitions between the drivers each race weekend. For example, we might compare:

- the number of times that one driver beat the other in qualifying;
- the average (mean) gap between drivers in qualifying (for example, in the last session both drivers made it in to);

⁶⁷<https://ig.ft.com/sites/the-best-f1-drivers-ever/>

⁶⁸<https://f1metrics.wordpress.com/2015/08/21/a-reconstructed-history-of-formula-1/>

- the number of times one driver beat the other in the race; we might additionally restrict this comparison to races where both drivers were classified;
- the number of times one driver started behind the other on the grid but went on to rank higher in the final classification, perhaps limited to situations where both drivers were classified;
- the number of races in which each driver had a fastest racing lap faster than their team mate;
- the average (mean) gap between drivers' fastest laps in the race, perhaps limited to situations where both drivers completed the race;
- the total number of laps on which one driver led the other, or had a faster laptime than the other.

Qualifying Performance

Let's start off by pulling some qualifying session results data directly from the *ergast* online API for a specific team in a specific year.

```
source('ergastR-core.R')
qualiResults=qualiResults.df(2014,constructorRef='mercedes')
```

We can reshape the data to generate a dataframe with two columns - one for each driver in the team - that gives their qualifying position in a given race. We shall call such a dataframe a *faceoff* dataframe.

```
library(reshape2)

#The dcast function can be used to "unmelt" a data frame
faceoff=dcast(qualiResults[, c('code','position','round')],
               round ~ code, value.var='position')

##   round HAM ROS
## 1     1   1   3
## 2     2   1   3
## 3     3   2   1
```

We can directly count how many times each driver beat the other based on summing the results of an inequality test applied across the driver columns, and then labeling the result

appropriately. This approach relies on being able to count a TRUE result as a 1, and a FALSE result as a 0. We index into the columns of the *faceoff* dataframe to compare the driver results, before grabbing the driver names from the appropriate columns in the original dataframe as the labels for the corresponding columns in the new dataframe.

```
df=data.frame(sum(faceoff[,2] < faceoff[,3]),
              sum(faceoff[,2] > faceoff[,3]))
names(df)=c(names(faceoff)[2],names(faceoff)[3])
df

##      HAM ROS
## 1    7 12
```

In order to report on the battles across all the teams, we need to take a slightly different approach. Let's pull down all the qualifying results from the 2014 season from the *ergast* API, and then see if we can generate a report that counts the number of times a particular driver beat their team mate during qualifying.

```
qualiResults2=qualiResults.df(2014)

#The advantage function decides who took the advantage in a particular faceoff
#It takes a list of rounds and for each driver returns a count of the rounds
# in which they had the advantage
adv=function(faceoff){
  dfx=data.frame(code=character(),advantages=numeric())
  #The first faceoff column is the round; then we have the drivers
  #In this example a driver has the advantage if their position is the same as
  # the minumum position across the driver columns for that round.
  #We need to generate advantage reports for each driver in the round
  for (i in 2:ncol(faceoff)){
    dfx=rbind(dfx,
               data.frame(code=names(faceoff)[i],
                           advantages=sum((i-1)==apply(faceoff[,-1],1,which.min))))
  }
  dfx
}

#Generate a faceoff dataframe that has rows as rounds and one column for each driver
#The cell value gives the position for a driver in a given round
```

```
#Note that some teams fielded more than two different drivers over the season,
# which means that for some teams there will be more than two driver columns
advrep=function(x){
  faceoff=dcast(x[,c('code','position','round')],
                round~code,
                value.var='position')
  adv(faceoff)
}

#We do the counts for each team
teamcounts=ddply(qualiResults2,.(constructorId),function(x) advrep(x))
teamcounts

##   constructorId code advantages
## 1      mercedes  HAM        7
## 2      mercedes  ROS       12
## 3     red_bull   RIC       11
## 4     red_bull   VET        7
## 5      mclaren   MAG        9
## 6      mclaren   BUT       10
## 7      ferrari   ALO       16
## 8      ferrari   RAI        3
## 9    toro_rosso  VER        7
## 10   toro_rosso  KVY       12
## 11  force_india  HUL       12
## 12  force_india  PER        7
## 13     williams  MAS        6
## 14     williams  BOT       13
## 15      sauber   SUT       10
## 16      sauber   GUT        9
## 17    caterham   KOB       12
## 18    caterham   ERI        4
## 19    caterham   LOT        1
## 20    caterham   STE        0
## 21    marussia   CHI        4
## 22    marussia   BIA       12
## 23   lotus_f1   GRO       15
## 24   lotus_f1   MAL        4
```

The approach taken to generate the above table is rather laboured, and in part reflects the way the data is tabulated.

Sometimes, it can be easier to construct a query using an alternative approach. For example, we can load the `qualiResults2` data from the `ergast` API into a SQLite database and then generate a similar report to the one above with the following SQL query:

```
library(DBI)
con = dbConnect(RSQLite::SQLite(), ":memory:")
tmp=dbWriteTable(con, "qualiResults2", qualiResults2, row.names = FALSE)

dbGetQuery(con,
  'SELECT q1.constructorId,q1.code,COUNT(*)
   FROM qualireturns2 q1 JOIN qualireturns2 q2
   WHERE q1.constructorId=q2.constructorId
   AND q1.round=q2.round
   AND q1.position<q2.position
   AND q1.code!=q2.code
   GROUP BY q1.constructorId, q1.code')

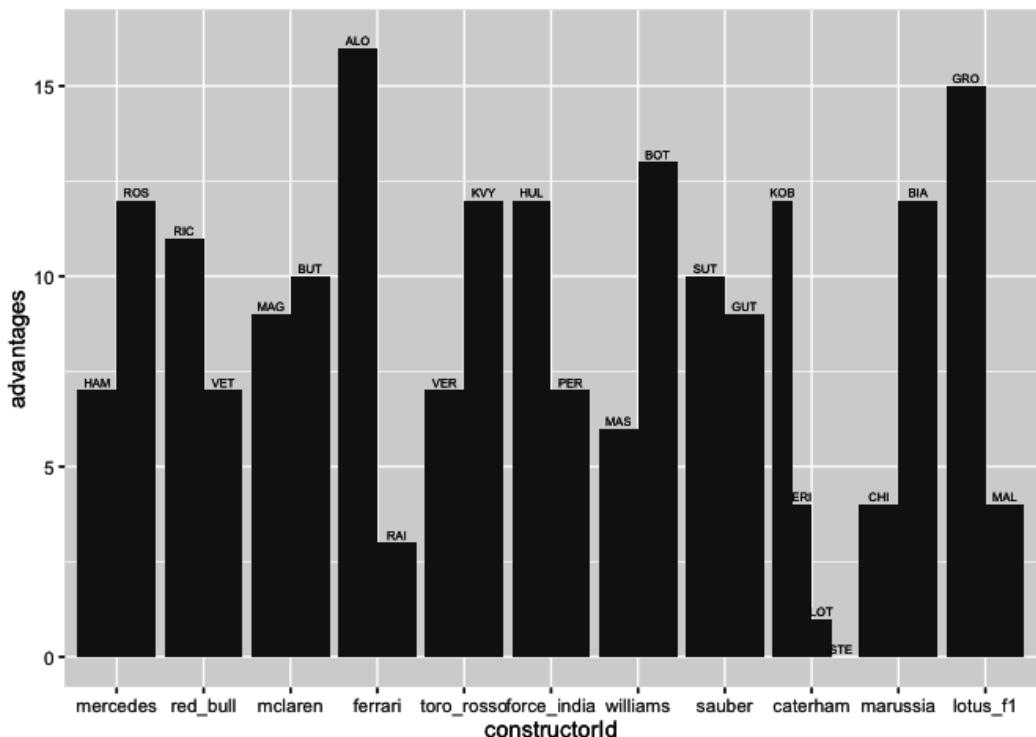
##      constructorId code COUNT(*)
## 1      caterham    ERI      4
## 2      caterham    KOB     12
## 3      caterham    LOT      1
## 4      ferrari     ALO     16
## 5      ferrari     RAI      3
## 6  force_india    HUL     12
## 7  force_india    PER      7
## 8    lotus_f1    GRO     15
## 9    lotus_f1    MAL      4
## 10   marussia     BIA     12
## 11   marussia     CHI      3
## 12    mclaren     BUT     10
## 13    mclaren     MAG      9
## 14    mercedes    HAM      7
## 15    mercedes    ROS     12
## 16   red_bull     RIC     11
## 17   red_bull     VET      7
## 18     sauber     GUT      9
## 19     sauber     SUT     10
## 20  toro_rosso    KVY     12
## 21  toro_rosso    VER      7
## 22    williams    BOT     13
## 23    williams    MAS      6
```

To my mind, this is a much more convenient way of getting the result, although the tabular form of the report is still hard to digest.

We can generate a quick to skim graphical review of the results using a dodged bar plot, grouped by team, with distinct bars representing the different drivers in each team.

```
library(ggplot2)

g=ggplot(teamcounts,aes(x=constructorId,group=code,y=advantages))
g=g+geom_bar(stat="identity",position="dodge",width=0.9)
g=g+geom_text(aes(y=advantages+0.2,label=code),
              position=position_dodge(width=0.9),
              size=2)
g
```



Bar chart showing how many times drivers within a team beat their team mate in qualifying during the 2014 season

This is actually quite a brutal chart, made worse by the lack of colour definition to distinguish each driver within the team. (But then, how might we generate a binary classification scheme to color each driver in a team?) Surely there must be a better way of representing this information?

Race Performance

Before trying to ask very particular questions about how drivers across different teams compared in terms of their race performances during a season, let's have a look at how the drivers compare over such a period *within* a team. We can pull the race results down for a particular driver directly from the *ergast* API.

```
button=driverResults.df('2014','button')
magnussen=driverResults.df('2014','kevin_magnussen')
```

code	constructor	grid	laps	position	positionText	points	status	season	round
BUT	mclaren	10	57	3	3	15	Finished	2014	1
BUT	mclaren	10	56	6	6	8	Finished	2014	2
BUT	mclaren	6	55	17	17	0	Clutch	2014	3

We can now generate a *raceoff* dataframe that contains the results for the two drivers so we can compare them directly.

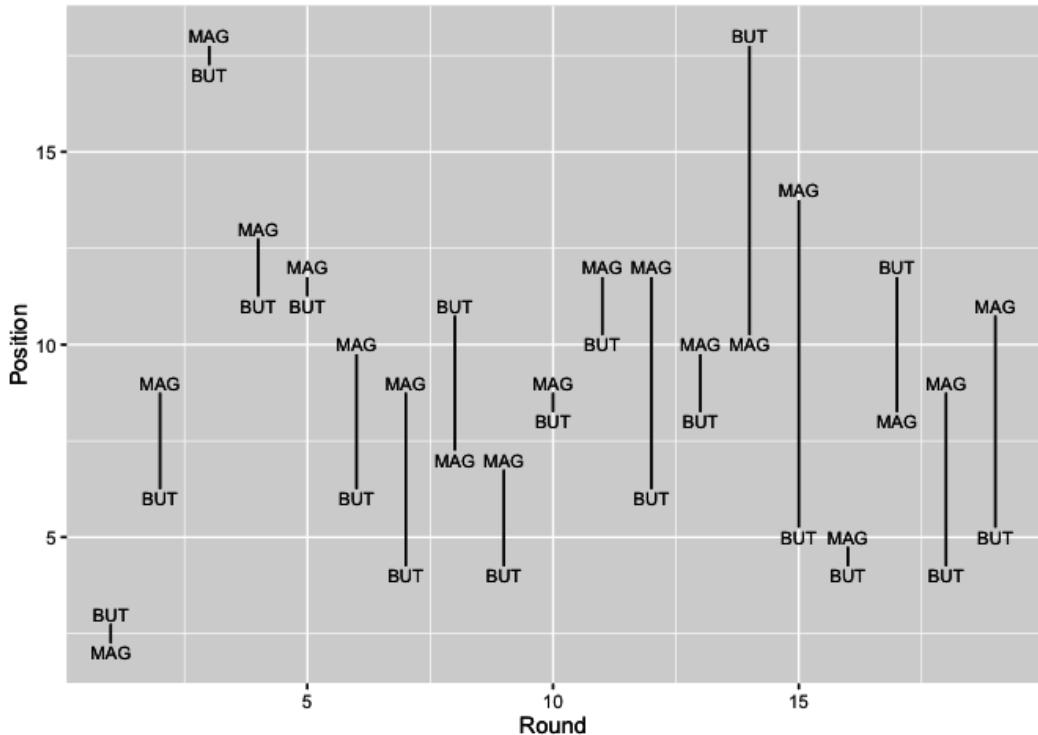
```
raceoff=df=function(d1,d2){
  raceoff=merge(d1[,c('code','position','positionText','round')],
                d2[,c('code','position','positionText','round')],
                by='round')
  raceoff$topd2=(raceoff$position.x>raceoff$position.y)

  #The dNpos arguments identify the max and min positions
  raceoff=ddply(raceoff, .(round),transform,
                d1pos=max(position.x,position.y),
                d2pos=min(position.x,position.y))
  raceoff
}

raceoff=raceoff.df(button,magnussen)
```

We can generate a quick sketch of how the drivers fared relative to each other in each round by plotting their respective positions in each round, connected by a line to highlight the distance between them.

```
#Create the base plot
g = ggplot(raceoff, aes(x=round))
#Add text labels showing the position of each driver in each round
g = g + geom_text(aes(y=position.x, label=code.x), size=3)
g = g + geom_text(aes(y=position.y, label=code.y), size=3)
#Add a line between the two labels for each round to connect them together
g = g + geom_segment(aes(x=round, xend=round, y=d1pos-0.25, yend=d2pos+0.25))
g + xlab('Round')+ylab('Position')
```



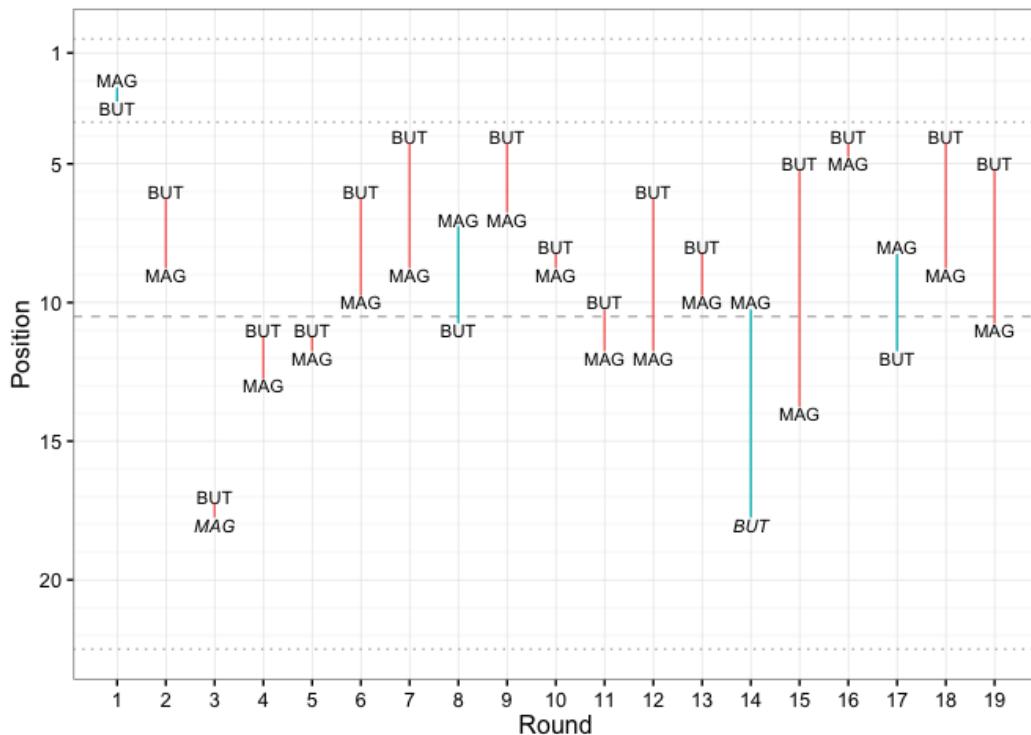
Sketch comparing final race positions of two drivers in the same team, by round, in the 2014 Drivers' Championship

Working from this sketch, we can start to elaborate on the design. For example, we might add in guides that distinguish podium positions or points gaining positions; we can colour the

line to highlight even further which driver was higher placed; and we might italicise driver labels where their position went unclassified. Finally, adding a clear theme and inverting the y-axis scale so the higher placed driver is at the top of the chart produces a far more stylised and stylish chart with several additional channels of information over and above the original sketch.

```
driverPos=function(raceoff){
  #Base chart
  g=ggplot(raceoff,aes(x=round))
  #Guides to highlight podium and points positions, and the back of the field
  g=g+geom_hline(yintercept=3.5,colour='grey',linetype='dotted')
  g=g+geom_hline(yintercept=10.5,colour='grey',linetype='dashed')
  g=g+geom_hline(yintercept=0.5,colour='grey',linetype='dotted')
  #We should really calculate the following intercept value based on size of the field
  g=g+geom_hline(yintercept=22.5,colour='grey',linetype='dotted')
  #Add the driver labels.
  #Set the fontface according to whether the driver retired or not
  g=g+geom_text(aes(y=position.x,
                     fontface=ifelse((positionText.x=='R'), 'italic' , 'plain'),
                     label=code.x),size=3)
  g=g+geom_text(aes(y=position.y,
                     fontface=ifelse((positionText.y=='R'), 'italic' , 'plain'),
                     label=code.y),size=3)
  g=g+xlab('Round')+ylab('Position')
  #Add in the lines, colouring them by whcih driver was higher placed
  g=g+geom_segment(aes(x=round,xend=round,y=d1pos-0.25,yend=d2pos+0.25,col=topd2))
  #Tidy up the scales and grid lines
  #The breaks range should really be calculated based on the size of the field
  g=g+scale_x_continuous(breaks = 1:22,minor_breaks=NULL)
  g=g+scale_y_reverse(breaks = c(1,5,10,15,20),minor_breaks=1:22)
  #Tidy up the theme
  g=g+guides(colour=FALSE)+theme_bw()
  g
}

driverPos(raceoff)
```



Race position comparison chart for the McLaren drives in the 2014 season.

In this form, the chart clearly demonstrates how closely (or not) the drivers are placed in each race, along with information about whether they were on the podium or in the points, and whether or not they retired from the race or achieved a fully classified result. We might also use bold emphasis to distinguish another channel of information, such as which driver started the race higher on the grid.

If we can find a way to further annotate these charts with a text label that identifies the qualifying and grid position of each driver, we could provide an even more complete summary of how the drivers compared over the qualifying and race sessions.

To maximise the use of space, we might even drop the name labels and instead just rely on the colour field to identify which driver was higher placed at the end of the race. We could also use line styles with associated semantics for particular situations, such as using dashed lines to highlight when one driver was out of the points, or dotted lines to show emphasis when both drivers were out of the points.

Summary

In this chapter we have started to explore various ways of comparing the performances of drivers within a team. Such charts may have a useful role to play in a season report, for example.

The *race position comparison chart* style may also be appropriate for summarising the qualifying achievements of the two drivers in a team.

Points Performance Charts

Knowing the likely payoff for different grid positions in terms of the number of points a car starting in that position is likely to attain is one-way of putting points-and-positions data to work. Another way in which we can look at points data is to use it to compare the performance of two different drivers, such as two drivers in the same team.

In this chapter, we'll explore some preliminary sketches of *points performance charts*, a novel graphical technique for comparing not just driver performances. The charts are also intended to demonstrate the extent to which a team maximises its points haul when the lead driver attains a particular points scoring position, although in their current form they are not as successful as hoped for in demonstrating this: the charts are hard to interpret and don't clearly communicate how teams compare in their points haul efficiency - the race position comparison charts are as equally effective at revealing this, if not more so.

This approach is very much a work in progress whose usefulness - or otherwise - is yet to be proven. What it will demonstrate, nevertheless, is how we can construct our own graphical chart types as tools that help us probe our understanding of the data in novel ways.

The origins of the *points performance chart* begin with a question surrounding the extent to which teams maximise their points haul given the finishing position of their highest placed driver in each race. But first, let's quickly have a look at another points related chart: *grid points productivity*.

Grid Points Productivity

Since the 2010 season, the top ten classified positions in each race are awarded an associated number of championship points depending on classified position (25 for first, 18 for second, 15 for third, then 12, 10, 8, 6, 4, 2 and finally 1 point for 10th) (FIA F1 Regulations⁶⁹). A tweak to the 2014 regulations made double points available in the final race of the season of that year.

Grid points productivity describes the number of points a driver might be expected to earn from a given grid position. That is, for each grid position over a given period, find the mean number of points awarded.

⁶⁹http://www.fia.com/sport/regulations?f%5B0%5D=field_regulation_category%3A82

```

library(DBI)
library(plyr)

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

q='SELECT rs.grid, rs.points, rs.position, rs.grid-rs.position gridposdelta
  FROM results rs JOIN races r
 WHERE rs.raceId=r.raceId AND r.year>=2010'

rr=dbGetQuery(ergastdb, q)

#Make a factor from the grid positions
rr['gr']=factor(rr$grid)

#Find the mean number of points associated with each grid position
rrp=ddply(rr,.(gr),summarise,m=mean(points))

```

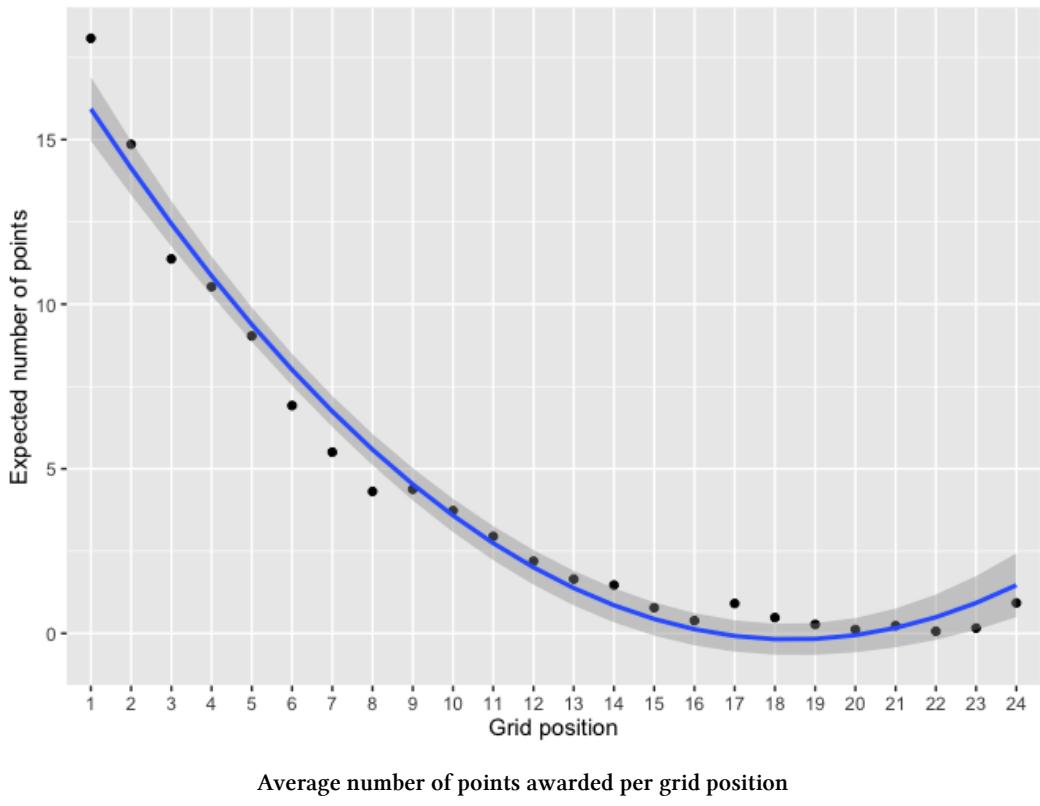
Charting these shows how we might expect points to be awarded for a given grid position, *on average*:

```

library(ggplot2)

g=ggplot(rrp[-1,],aes(x=gr,y=m,group=1))
g=g+geom_point()+stat_smooth(method = "lm", formula = y ~ x+I(x^2))
g + xlab('Grid position') + ylab('Expected number of points')

```



So for example, from first on the grid, we might expect a driver to earn 18 points, from second, 15 points, and so on.

We can also generate a chart that shows the mean *gridposdelta*, that is, the average number of places made (or lost) from a given grid position.

```
#Find the average displacement in terms of positions gained or lost from a given start
rrg=ddply(rr[!is.na(rr$position),], .(gr), summarise, m=mean(gridposdelta))

g=ggplot(rrg,aes(x=gr,y=m,group=1))+geom_point()
g=g+stat_smooth(method = "lm", formula = y ~ x+I(x^2))
g + xlab('Grid position') + ylab('Expected change in position')
```

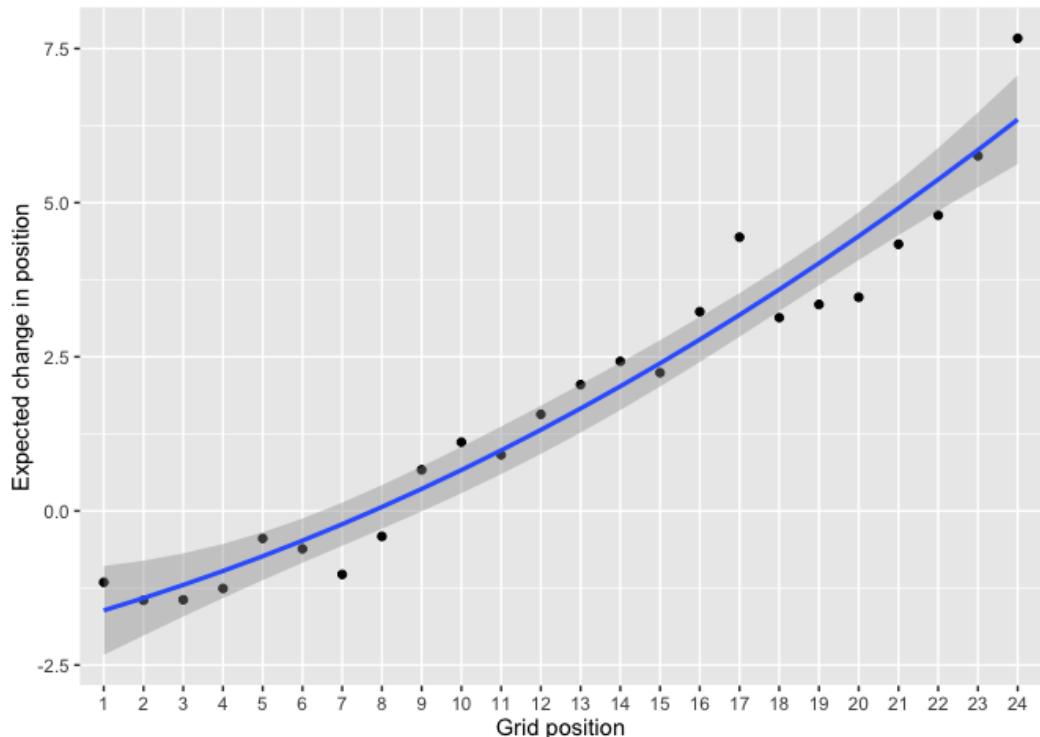


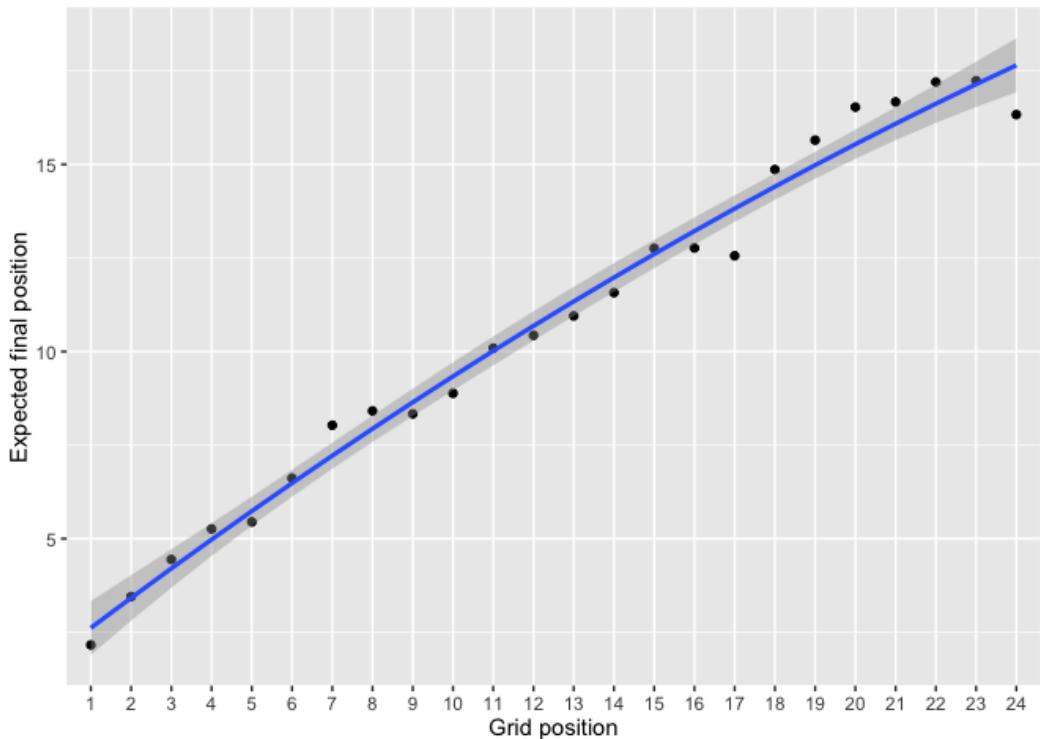
Chart showing the number of positions you might expect to gain - or lose - form a given grid position

From this chart, it looks as if cars starting from 8th or above on the grid are likely to lose a position, whereas cars starting 9th or lower look set to gain places, *on average*, with 2nd and 3rd being susceptible to most change at the front of the grid.

A complementary chart might plot the average final classification against the starting grid position:

```
#Find the average final position for a given grid position
rrg=ddply(rr[!is.na(rr$position),], .(gr), summarise, f=mean(position))

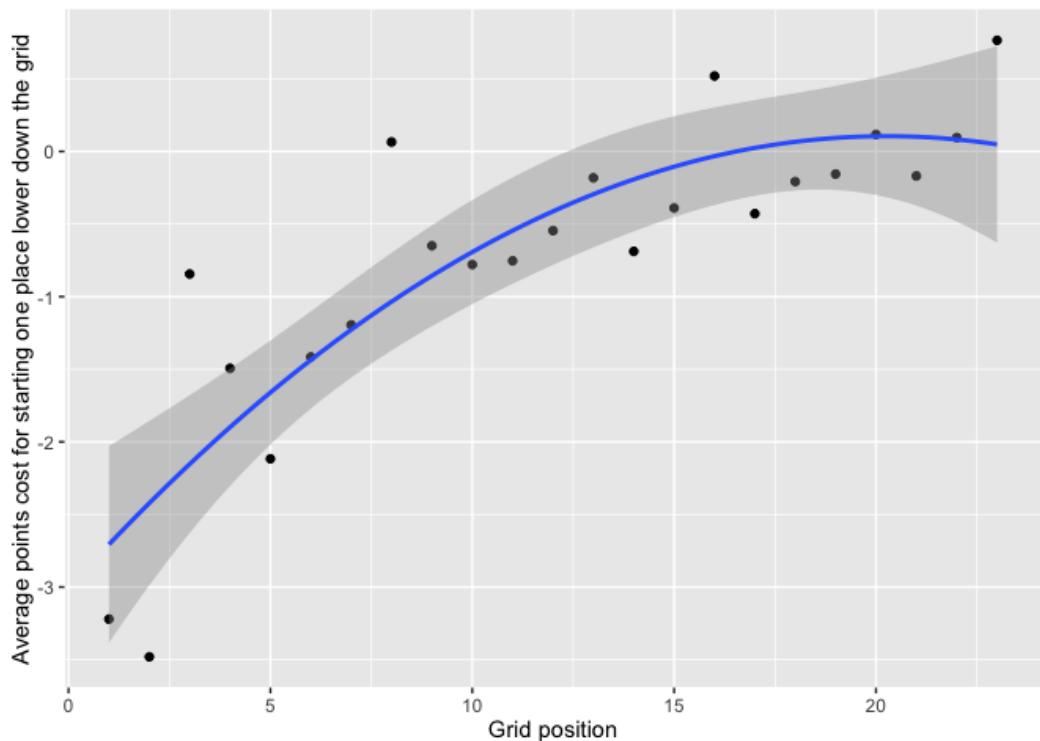
g=ggplot(rrg,aes(x=gr,y=f,group=1))+geom_point()
g=g+stat_smooth(method = "lm", formula = y ~ x+I(x^2))
g + xlab('Grid position') + ylab('Expected final position')
```



Finally, we can generate a chart that shows the average points effect for each grid position of qualifying one place further down the grid.

```
#Find the running difference in the mean number of points from a given grid position
rrpdf=diff(rrp$rn)
dfx=data.frame(i=1:length(rrpdf[-1]),d=rrpdf[-1])

g=ggplot(dfx,aes(x=i,y=d))+geom_point()
g=g+stat_smooth(method = "lm", formula = y ~ x+I(x^2))
g=g + xlab('Grid position')
g + ylab('Average points cost for starting one place lower down the grid')
```



Charting the expected points cost of starting one place lower down the grid for a given grid position

In this chart, we see that starting in 3rd rather than 2nd would cost you more points than if you started in 2nd rather than 1st; similarly, going back from 3rd to 4th costs may not even cost you a point, on average, but starting in 6th rather than 5th is likely to cost you at least 2 points.

Maximising Team Points Hauls

Championship points serve to provide continuity and an element of competition across a season, and are used to maintain audience interest even if the action on track is at times somewhat processional.

If a team's highest classified car finishes in the top 9 positions, the team will maximise its points haul if the other car is placed in the next position. If the highest placed car wins the race, scoring the maximum 25 points (unless it was the last race of the season in 2014, when double points were awarded), the team will obviously maximise its points haul if the other

car is placed second. In fact, this is the ultimate points haul - $25 + 18 = 43$ points for the team from a single race.

Let's start by grabbing some data, joining together two copies of the results table on raceId, driverId and constructorId with distinct driverId. We'll record the results of the higher-placed driver in the team ($r1$) and the lower placed driver ($r2$) in the same team from across several seasons. We'll also grab the total points haul for each team in each race.

```
#Find results for pairs of drivers in the same team and the same race
#At least one driver must score points
#r1 is placed ahead of r2 in terms of points haul
results=dbGetQuery(ergastdb,
  'SELECT year, constructorRef AS team,
    r1.position AS r1pos,
    r2.position AS r2pos,
    r1.points AS r1points,
    r2.points AS r2points,
    (r1.points+r2.points) AS points
  FROM results r1 JOIN results r2 JOIN races r JOIN constructors c
  WHERE r1.raceId=r2.raceId
    AND r1.constructorId=r2.constructorId
    AND c.constructorId=r2.constructorId
    AND r.raceId=r1.raceId
    AND r1.driverId!=r2.driverId
    AND r1.points>r2.points
    AND year >=2010' )
```

We can use a box plot to show the distribution of team points across a series of races where at least one team member was in the points, illustrating the extent to which teams maximised their points haul during each race.

```
points=c(25,18,15,12,10,8,6,4,2,1)

pos.points=data.frame(position=seq(10),
  points=points,
  maximiser.points=c(points[-1],0),
  max pts=points+c(points[-1],0))

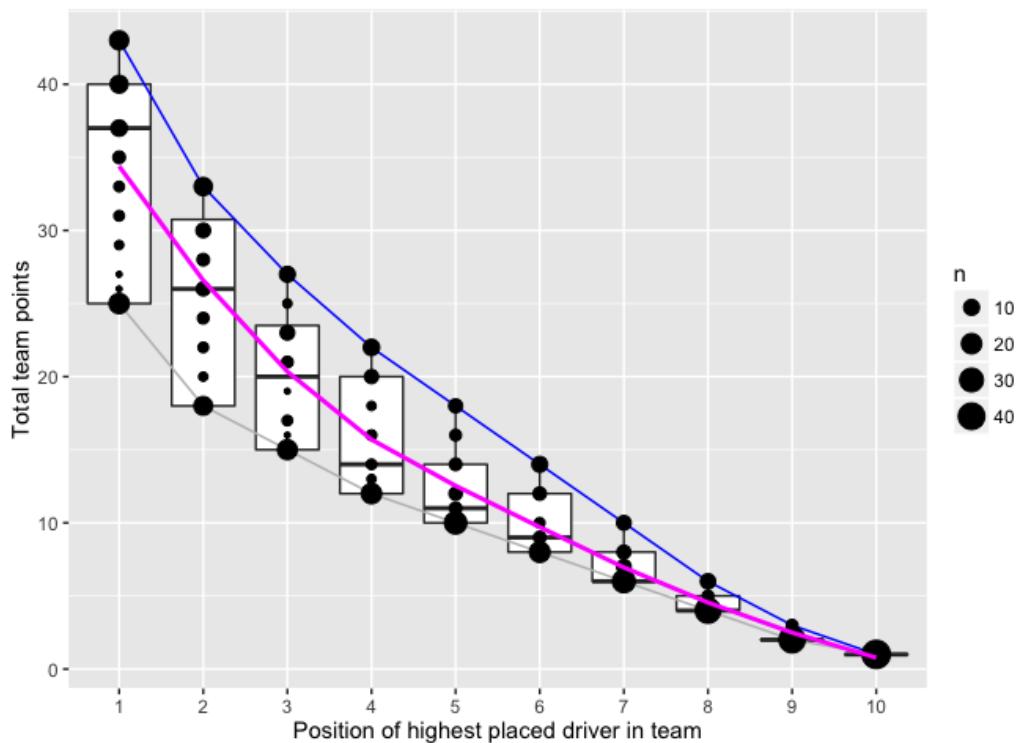
teamPerformance=function(results){
  g= ggplot(results,aes(x=factor(r1pos),y=points))
  g= g+ geom_boxplot(aes(group=r1pos))
```

```

g= g+ geom_line(data=pos.points, aes(x=position, y=max.pts), colour='blue')
g= g+ geom_line(data=pos.points, aes(x=position, y=points), colour='grey')
g=g+stat_sum(aes(size = ..n..))
g= g+ stat_smooth(aes(x=r1pos), colour='magenta', se=FALSE)
g =g+ xlab('Position of highest placed driver in team')
g= g + ylab('Total team points')
g
}

g=teamPerformance(results)
g

```



Box plot showing the distribution of the total team points scored by drivers, per team, in races from the 2010 to 2013 seasons inclusive, against the position of the highest placed driver in team

The blue line shows the maximum points available; the magenta line is a fitted line on the distribution of total points received per ranking position of the highest (or only) placed driver in the team; the grey line shows the minimum points haul (that is, just the number of points

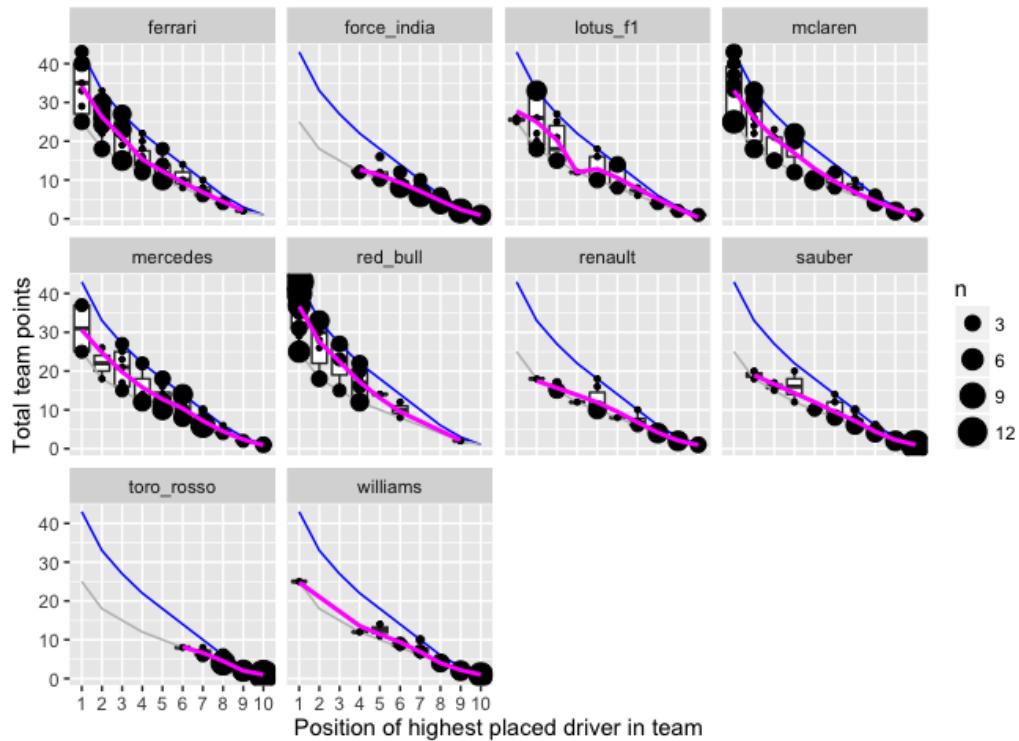
scored by the highest placed driver). The filled circles are proportionally sized according to the number of times a particular team points haul was recorded given the position of the highest ranked team member. In this example, data comes from all teams for the seasons 2010-2013.

If each team maximised its points (given a particular classification for the lead driver) in every race, there would be a single large circle on the uppermost blue line for each finishing position.

In the boxplot itself, the horizontal bar depicts the *median* value of the total points scored by a team for each finishing position of the highest placed driver in the team. The box itself shows the range of the first to the third quartile and the lines extend out to the highest and lowest values that lie within 1.5 times the interquartile range. This all sounds very complicated, but basically it describes the distribution of the points scores and allows outliers to be highlighted specifically. (In this case, there are no outliers.)

A faceted chart allows us to summarise the ability of different teams to maximise their points hauls.

```
g+facet_wrap(~team)
```

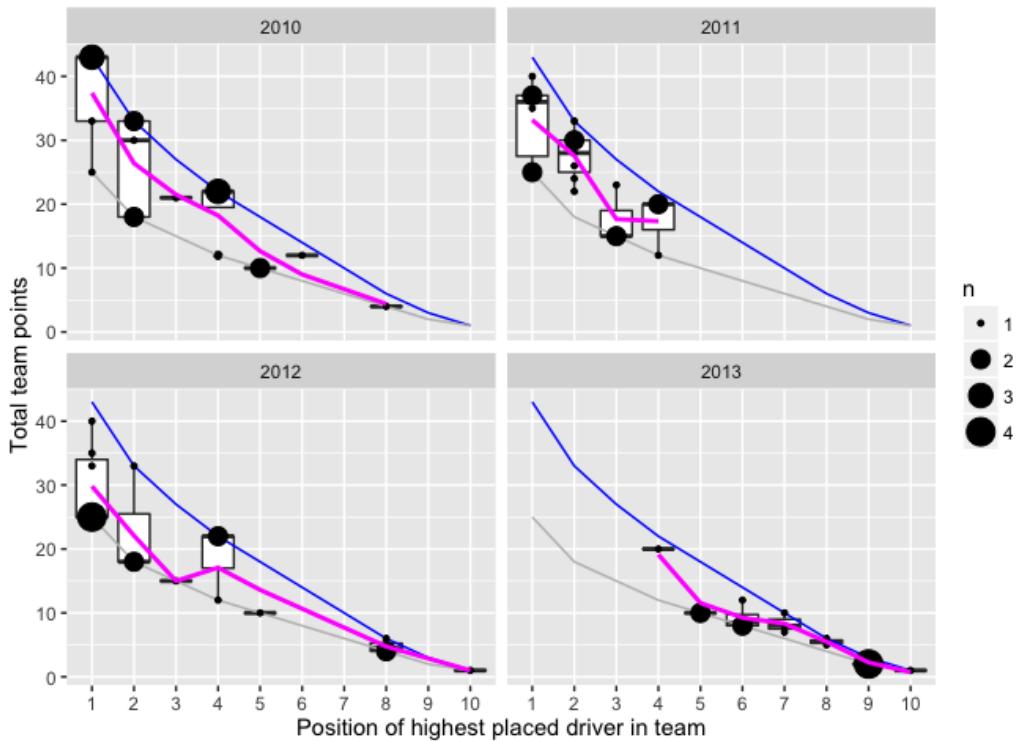


plot of chunk teamPerfBoxPlot_facet

The closeness of the magenta line to the blue line is an indicator of the extent to which the team members were keeping each other honest and the team was maximising its points haul. The closeness of the magenta line to the grey line is an indicator of the extent to which the points haul was solely down to the highest placed driver in each race.

If we limit the data to driver performances from a particular team, we can see how well that team fared over several years.

```
teamPerformance(results[results['team']=='mclaren',])+facet_wrap(~year)
```

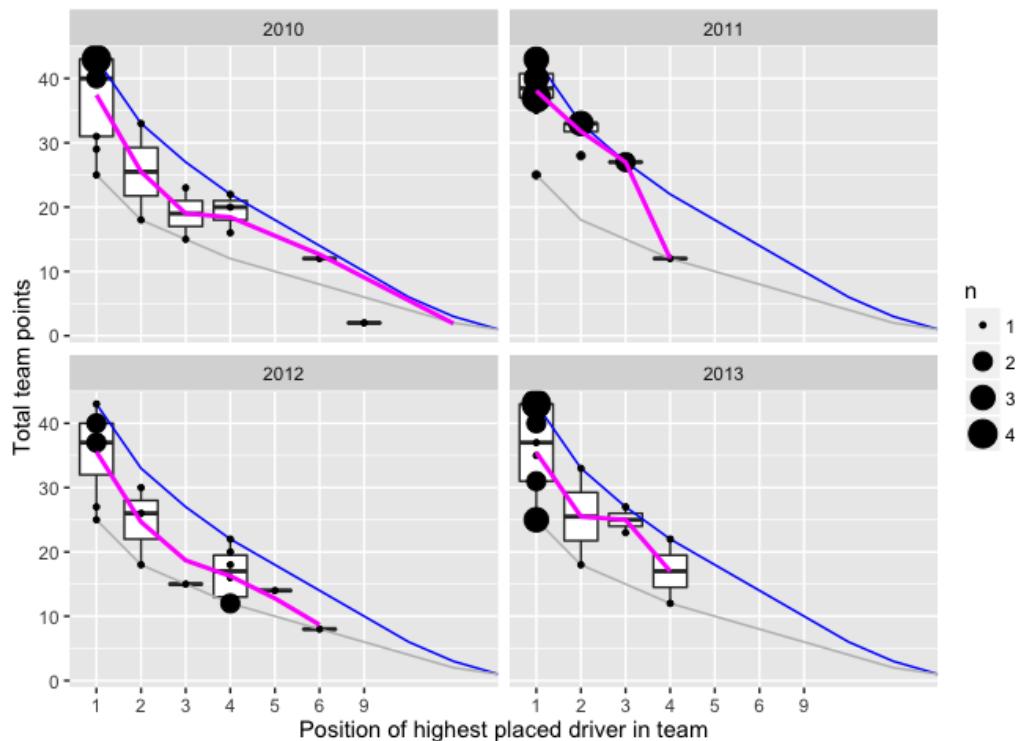


plot of chunk teamPerfBoxPlotMcLaren

This chart shows how, in 2011, McLaren lowest points scoring position was 4th. On the other hand, in 2013, the chart shows how McLaren missed out on the podium for the whole of the season.

But the following chart shows how Red Bull dominated that season even more convincingly:

```
teamPerformance(results[results['team']=='red_bull',]) + facet_wrap(~year)
```



plot of chunk teamPerfBoxPlotRedBull

We can look at summary statistics of the distribution of total points scores more exactly by looking at some key values in a tabular format, in particular the mean and median number of points scored by the teams. The following table summarises results across all teams and all races for the 2010 to 2013 seasons inclusive.

```
teamPerformance.table=function(results){
  teamPerformance =ddply(results, 'r1pos', summarise,
    mean_pts=mean(points),
    med_pts=median(points),
    mean_r2pts=mean(r2points),
    med_r2pts=median(r2points),
    mean_r2pos=mean(r2pos,na.rm=TRUE),
    med_r2pos=median(r2pos,na.rm=TRUE))
  teamPerformance['pts']=points
  teamPerformance['maxPts']=pos.points$max.pts
  teamPerformance['r2maxpts']=c(points[-1],0)
```

```

    teamPerformance
}

```

r1pos	maxPts	mean_pts	med_pts	pts	r2maxpts	med_r2pts	mean_r2pts
1	43	34.75	37	25	18	12	9.75
2	33	25.78	26	18	15	8	7.78
3	27	19.98	20	15	12	5	4.98
4	22	16.12	14	12	10	2	4.12
5	18	12.21	11	10	8	1	2.21
6	14	10.11	9	8	6	1	2.11
7	10	7.08	6	6	4	0	1.08
8	6	4.45	4	4	2	0	0.45
9	3	2.09	2	2	1	0	0.09
10	1	1.00	1	1	0	0	0.00

From this table we can see how far teams in general tend to fall short in terms of their ability to maximise points given the position of their highest placed driver. The discrete nature of the number of points awarded suggests that the median points score may give a more useful indicator of the “central” behavior of the second placed car in the team than the real valued mean points score.

How are we then to read the table? Recall that the median value identifies the number of points for which the team score more points half of the time, and less points the other half of the time. The median points scored column thus suggests that when the lead driver in a team is placed first, their teammate will finish in the top 4, giving a combined team points haul of 37 points.

As well as comparing points, we can also compare the typical positions achieved by the second placed driver in a team. This shows rather more directly how close in support the second driver was to the first placed driver in the team.

r1pos	med_r2pos	mean_r2pos
1	4.0	5.05
2	5.0	5.98
3	6.0	7.73
4	6.0	7.60
5	9.0	9.87
6	10.0	10.50
7	10.0	10.93
8	11.0	11.97
9	13.0	13.06
10	13.5	13.82

Over the course of several seasons, we see that there must have been some rotation of teams on the podium: the median position for teammates of podium finishers was three places behind. Furthermore, if a driver finishes in the top 7, it's as likely as not that his team-mate will also finish in the top 10.

Intra-Team Support

We can get a better idea of how the team mates supported each other by plotting their respective positions on a jittered scatter plot. In the plot on the left, each point relates to a result for a particular team. In the plot on the right, the dots reflect the number of results with that points profile. The horizontal x-axis is the position of the higher classified driver, the vertical y-axis the classification of their lower placed team mate.

```
library(gridExtra)

#ggplot2 linetypes: blank, solid, dashed, dotted, dotdash, longdash, twodash

teampoints.dualchart=function(results){
  g=ggplot(results,aes(x=r1points,y=r2points))
  g=g+stat_smooth(se=FALSE,
                   colour='blue',linetype='dashed')
  g=g+geom_line(data=pos.points,aes(x=points,y=maximiser.points),
                 col='red',linetype='solid')
  g=g+geom_line(data=teamPerformance,aes(x=pts,y=med_r2pts),
                 col='magenta',linetype='dotdash')
  g=g+xlab(NULL)+ylab(NULL)

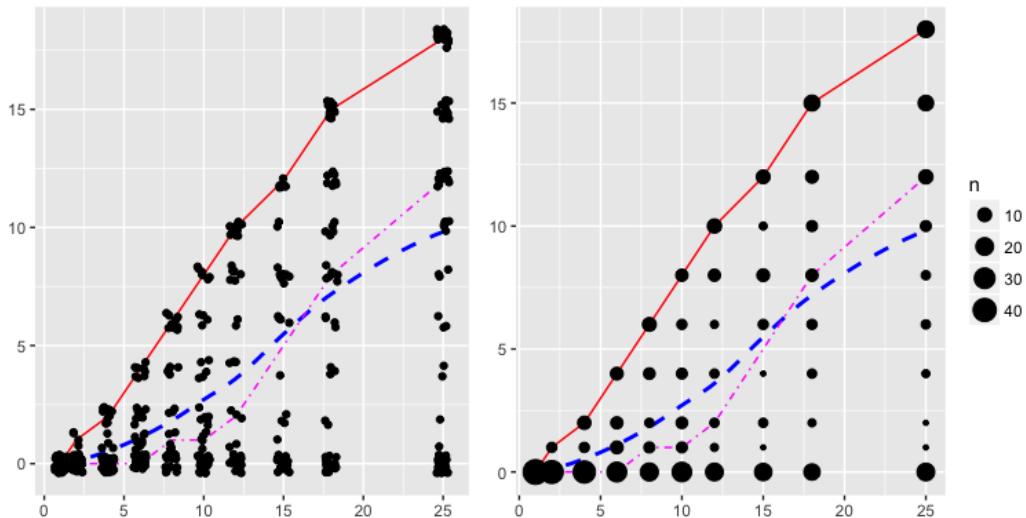
  g1=g+geom_jitter()
```

```

g2=g+stat_sum(aes(size = ..n..))
grid.arrange( g1, g2, ncol=2, widths=c(1.1,1.32))
}

teampoints.dualchart(results)

```



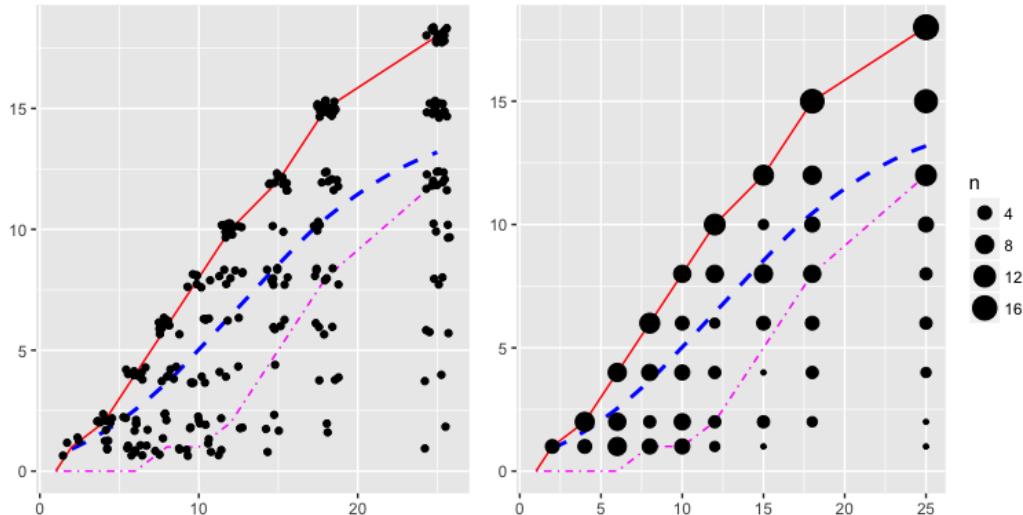
Comparison of team mate classifications where at least one driver is in the points for the 2010 to 2013 seasons. The higher placed driver's points are recorded against the horizontal x-axis, the lower placed driver's against the vertical y-axis.

To improve the clarity of the chart presented in a black and white/photocopied form, we can use linestyle as well as colour to distinguish the separate lines. In the above example, the solid red line is the upper bound on the points the lower classified driver within a team can score. The dashed blue line is a fit that models the typical points contribution made by the second placed driver compared to their higher placed teammate. The dot-dashed magenta line plots the median number of points scored by the lower placed driver.

The above chart shows the behaviour for all teams over the sample period, but it doesn't really allow us to compare anything with anything much, other than perhaps noting a tendency for drivers to well support each other *if* they both finish in the points.

If we limit the sample to just those situations where a team placed *both* its drivers in the points, we can get a clearer view of how many points a team might expect to earn if it places both drivers.

```
teampoints.dualchart(results[results['r2points'] > 0, ])
```



Comparison of team mate classifications where both drivers are in the points for the 2010 to 2013 seasons. The higher placed driver's points are recorded against the horizontal x-axis, the lower placed driver's against the vertical y-axis.

If you win, you expect your team mate to finish in the top 4. If you take third, it looks as if your team mate stands a good chance of being fourth or sixth. Does this tell us anything about race psychology?! The chart also shows how well teams do seem to maximise their points hauls.

Points Performance Charts - One-Way

Trying to get a feel for how teams maximise points in general is one thing, but can we develop this chart style to produce an instrument that gives us a useful indication about how well a particular team is faring in terms of points maximisation?

A *one-way points performance chart* can be used to show how well drivers within a team are supporting each other (or by extension, how closely two drivers from different teams are competing).

The chart will plot counts of the number of races in which each combination of points finishes was attained. In the *one-way* chart, we aren't interested in which driver was higher placed.

The chart will be annotated with a line that shows the maximum possible points haul, give the points take of the highest placed driver.

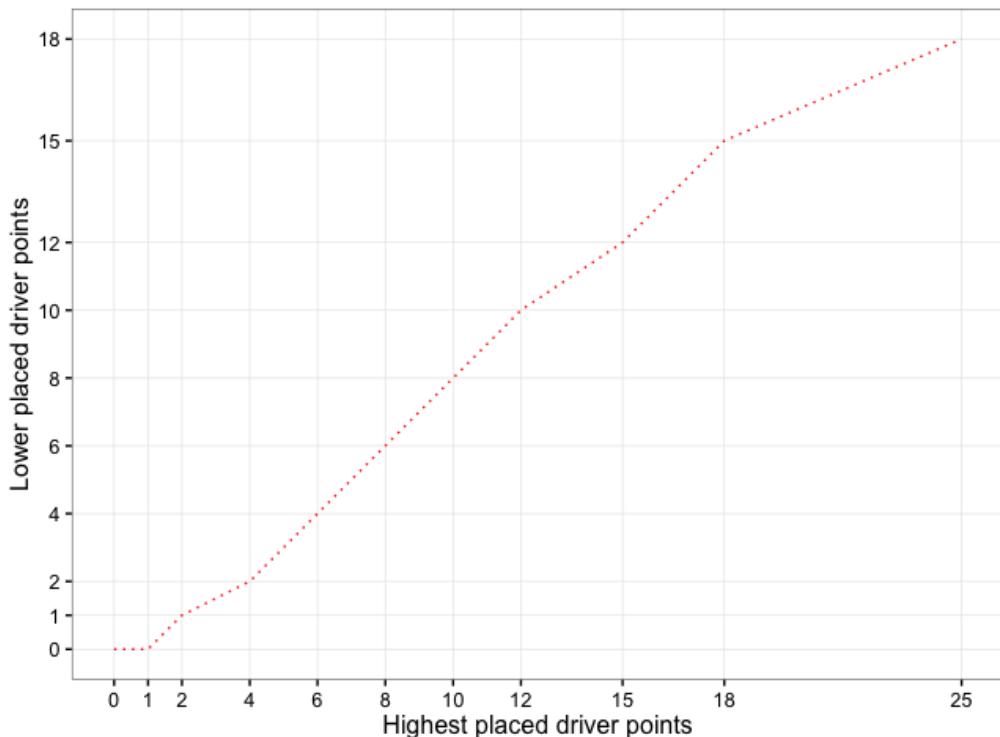
We'll use a clean design for the chart, with squared off axes, and set the grid lines to reflect the actual points scores that are possible.

```
NEWPOINTS =c(25,18,15,12,10,8,6,4,2,1)

#The newpoints dataframe has two columns
#The first column indicates points available, in order
#The second column is the maximum number of points the lower placed driver could score
newpoints=data.frame(x=c(NEWPOINTS,0),y=c(NEWPOINTS[-1],0,0))

baseplot_singleWay=function(g){
  g=g+geom_line(data=newpoints,aes(x=x,y=y),col='red',linetype='dotted')
  g=g+xlab('Highest placed driver points')+ylab('Lower placed driver points')
  g=g+scale_y_continuous(breaks = newpoints$x,minor_breaks=NULL)
  g=g+scale_x_continuous(breaks = newpoints$x,minor_breaks=NULL)
  g=g+coord_fixed()
  g=g+guides(size=FALSE)+theme_bw()
  g
}

baseplot_singleWay(ggplot())
```



The basis of the one-way points performance chart, with maximum points haul line indicated

Line Guides

Line guides can be added to a chart to help split up the chart area and provide a cue that helps us read more sense into the placement of particular marks. I don't know to what extent the Red Arrows pilots use the explosive canopy cords (which various commentators suggest they are) to help them sight their formations, but I think I'd make use of the visual cues they provide...



Red Arrows canopy view

Canopy view from Red Arrows cockpit

To generate our points position chart, we'll get the driver results for the drivers we want to compare and then annotate the chart with the higher and lower of the race positions and points scored by the drivers in each race. A flag is also set to identify whether or not the second listed driver was higher placed than the first.

```

raceoff_d1d2 = function(d1,d2) {
  raceoff=merge(d1[,c('points','position','positionText','round')], 
    d2[,c('points','position','positionText','round')], 
    by='round')
  #Is the second listed driver higher placed than the first?
  raceoff$topd2=(raceoff$position.x>raceoff$position.y)

  #Hack final round points for the 2014 season
  raceoff[raceoff['round']==19,]$points.x=raceoff[raceoff['round']==19,]$points.x/2
  raceoff[raceoff['round']==19,]$points.y=raceoff[raceoff['round']==19,]$points.y/2
  #d1 refers to the higher placed driver
}

```

```
#d2 refers to the lower placed driver
raceoff=ddply(raceoff, .(round),transform,
              d1pos=max(position.x,position.y),
              d2pos=min(position.x,position.y),
              d1points=max(points.x,points.y),
              d2points=min(points.x,points.y))
}
```

We can now generate a “raceoff” datatable from the race results of two drivers. As an example, let’s see how the two McLaren drivers, Jenson Button and Kevin Magnussen, supported each other in the 2014 season.

```
#We're going to use data from the live ergast API for these charts
source('ergastR-core.R')

button=driverResults.df('2014','button')
magnussen=driverResults.df('2014','kevin_magnussen')

#Generate the raceoff dataframe for two drivers
raceoff=raceoff_d1d2(button,magnussen)

#Summarise the counts of higher/lower points allocation combinations
raceoff.team.summary=ddply(raceoff,.(d1points,d2points),summarise,
                           cnt=length(round))
```

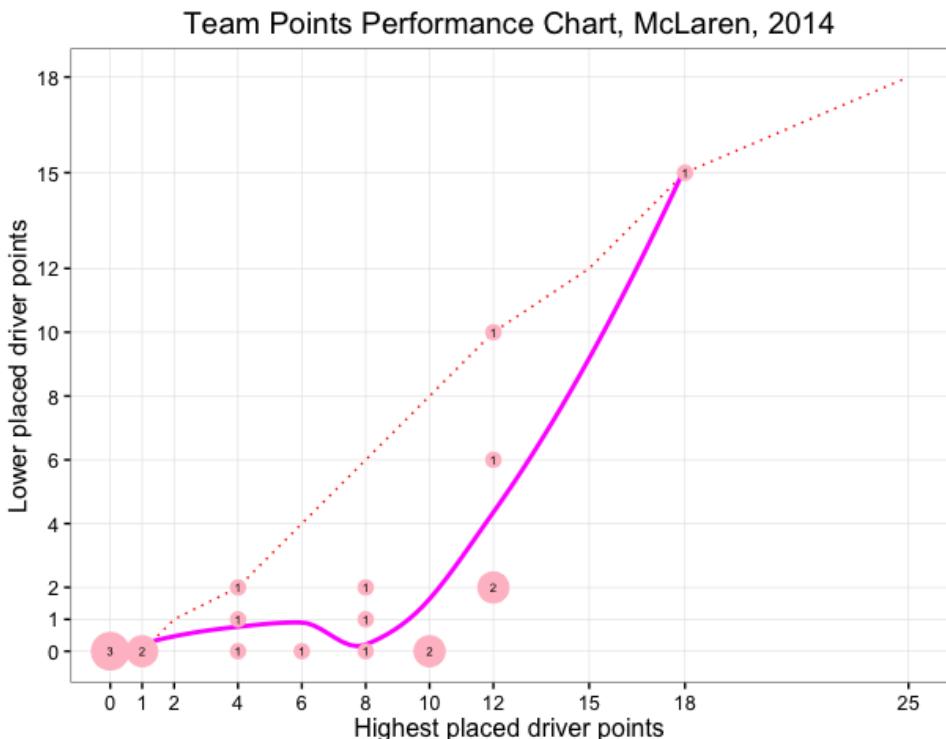
We can then plot this data onto our base chart, using a symbol size proportional to the count of how many races a particular points combination was received in, and further labelled with the same count value.

```
pointsPerformanceChart=function(raceoff,raceoff.team.summary,constructorRef){
  g = ggplot(raceoff.team.summary, aes(x=d1points, y=d2points))
  g = baseplot_singleWay(g)
  g = g + stat_smooth(data=raceoff, aes(x=d1points, y=d2points),
                       se=F,col='magenta')
  g = g + geom_point(aes(size=cnt), col='pink')
  g = g + geom_text(aes(label=cnt), size=2)
  g = g + scale_size_continuous(range=c(3,8))
  g = g + ggtitle(constructorRef)

  g
```

```
}
```

```
pointsPerformanceChart(raceoff, raceoff.team.summary,
    'Team Points Performance Chart, McLaren, 2014')
```



One-way points performance chart showing the extent to which a team did, or did not, maximise its points haul given the classification of its highest placed driver

The closer the magenta best fit line is to the dashed red maximum points haul line, the closer the team members were terms of points finishes within each race. In this example, we see that there were only three races in which both drivers finished in the points in consecutive places and thus maximised the team points haul.

There are two main weaknesses with this chart. Firstly, we don't know whether one driver in particular dominated the other. Secondly, we don't know how the points finishes evolved over the course of the season. We will address the first point in an extension to the idea of the points performance chart below, by considering a *two way* rather than a *one-way* chart. As for tracking the evolution of the points distribution over time, one approach might be to

generate a sequence of charts and animate the construction of the chart across the races in a particular season.

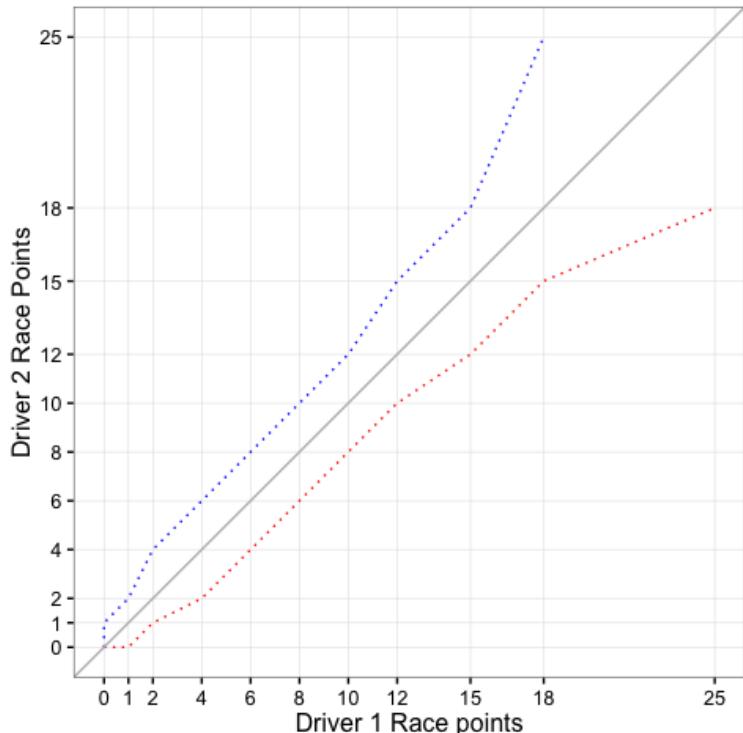
Points Performance Charts - Two-Way

To get round the problem of the one-way points performance chart's inability to distinguish which, if either, of the team's drivers was dominating the points haul, we can generate a two-way chart in which each axis represents the points taken in a particular race by a particular driver.

In this chart, we can plot two guides to indicate the maximum possible points haul within a team depending on which driver within the team is higher placed.

```
baseplot_twoway=function(g,d1="Driver 1 Race points", d2="Driver 2 Race Points"){
  g=g+geom_line(data=newpoints,aes(x=x,y=y),col='red',linetype='dotted')
  g=g+geom_line(data=newpoints,aes(x=rev(y),y=rev(x)),col='blue',linetype='dotted')
  g=g+geom_abline(col='grey')
  g=g+scale_y_continuous(breaks = newpoints$x,minor_breaks=NULL)
  g=g+scale_x_continuous(breaks = newpoints$x,minor_breaks=NULL)
  g=g+xlab(d1)+ylab(d2)
  g=g+coord_fixed()
  g=g+theme_bw()
  g=g+guides(size=FALSE)
  g
}
```

```
baseplot_twoway(ggplot())
```



Line guides on a two-way points performance chart. The upper (blue) dotted line shows the maximum points haul if Driver 1 (x-axis) was higher placed, the lower (red) dotted line the maximum points haul if Driver 2 (y-axis) was higher placed

The red dotted line shows the maximum points haul line if *Driver 1* on the x-axis was higher placed in a particular race; the blue dotted line shows the maximum points haul line if *Driver 2* on the y-axis was higher placed.

A points scoring team maximises its points haul when points combination markers are placed on the dashed lines.

We now need to generate a suitable summary dataset that counts the number of races where each particular combination of points was recorded.

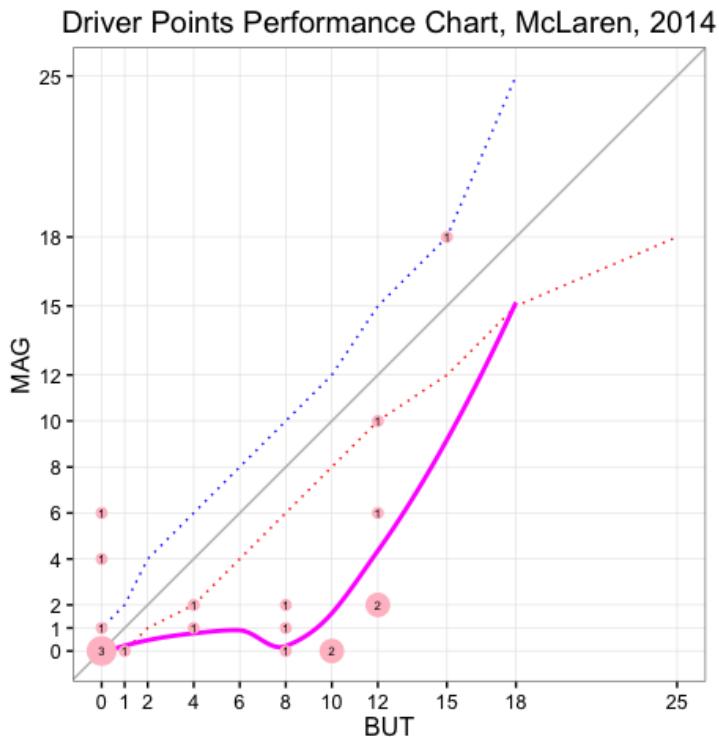
```
#Summary table counting how well each particular driver fares against the other
raceoff.summary=ddply(raceoff, .(points.x,points.y), summarise,
                      cnt=length(round))
```

This data can be overplotted onto the basic two-way points performance chart.

```
pointsDualPerformanceChart=function(raceoff,raceoff.summary,title,d1,d2){  
  g=ggplot(raceoff.summary,aes(x=points.x,y=points.y))  
  g=baseplot_twoway(g,d1,d2)  
  g=g+stat_smooth(data=raceoff,aes(x=d1points,y=d2points),se=F,col='magenta')  
  g=g+geom_point(aes(size=cnt),col='pink')  
  g=g+geom_text(aes(label=cnt),size=2)  
  g=g+scale_size(range=c(2,6),name="Count")  
  g=g+ggtitle(title)  
  
  g  
}
```

Once again, we use a proportionally sized symbol overplotted with a value to denote points combination counts and make use of a best fit or model line to highlight the general performance.

```
pointsDualPerformanceChart(raceoff,raceoff.summary,  
                           "Driver Points Performance Chart, McLaren, 2014",'BUT','MA\  
G')
```



Two Way Points Performance chart showing how well two drivers in the same team supported each other in terms of maximising team points haul

In this case, we see that the results are typically placed *off the maximum points lines*, showing that points take is not being maximised; and the marks are placed in the *lower right quadrant below the red dashed line*, signifying that the driver on the horizontal x-axis (in this case, Jenson Button (BUT)) tended to score more points than Kevin Magnussen (MAG) whose points scores are represented on the vertical y-axis.

If the marks tended to appear in the upper left quadrant above the blue dashed line, it would denote that the driver represented on the y-axis tended to outperform the driver represented on the x-axis.

Note that this chart does not require that drivers are members of the same team, only that they drove in the same races.

Exploring the Behaviour of the Two-Way Points Performance Chart

The *points performance charts* are a novel chart type that use maximum points haul guide lines to help us read sense into the marks placed on them. As a new chart type, it makes sense to use some extreme case dummy data to see what the limiting cases look like.

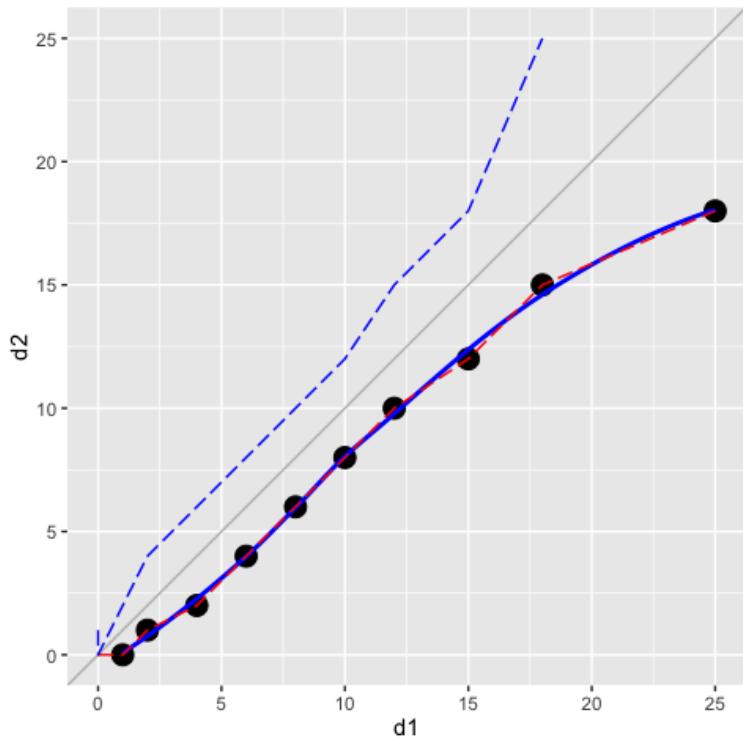
Let's create a simple function to sketch a crude two-way points performance chart.

```
twowayPointsPerfSketch=function(df){
  g=ggplot(df,aes(x=d1,y=d2))+stat_sum(aes(size = ..n..))
  g=g+geom_abline(col='grey')
  g=g+stat_smooth(se=F,colour='blue')
  g=g+geom_line(data=newpoints,aes(x=x,y=y),col='red',linetype='longdash')
  g=g+geom_line(data=newpoints,aes(x=y,y=x),col='blue',linetype='longdash')
  g + coord_fixed() +guides(size=FALSE)
}
```

We'll now consider several extreme cases of points take. First, for *twenty* races, where one driver scores each position in the top 10 twice, beating his team mate, who maximises points below by taking the next position in each race.

```
subpoints=c(NEWPOINTS[-1],0)
df=data.frame(d1=numeric(),d2=numeric())
for (i in seq(length(NEWPOINTS))) {
  df=rbind(df,data.frame(d1=NEWPOINTS[i],d2=subpoints[i]))
}
df=rbind(df,df)

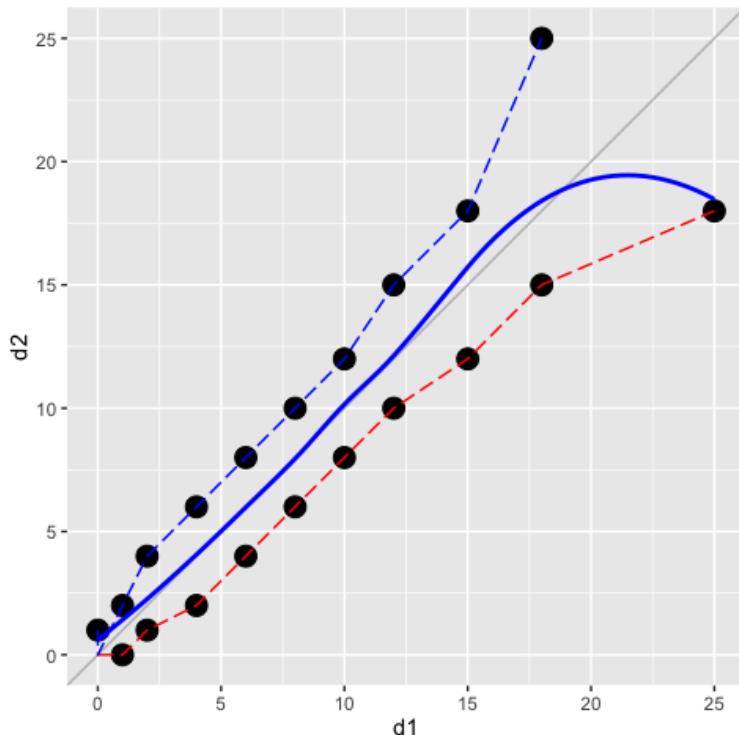
twowayPointsPerfSketch(df)
```



In this case, all the points lie on a single points haul maximisation line. The best fit model line closely follows the points maximisation line.

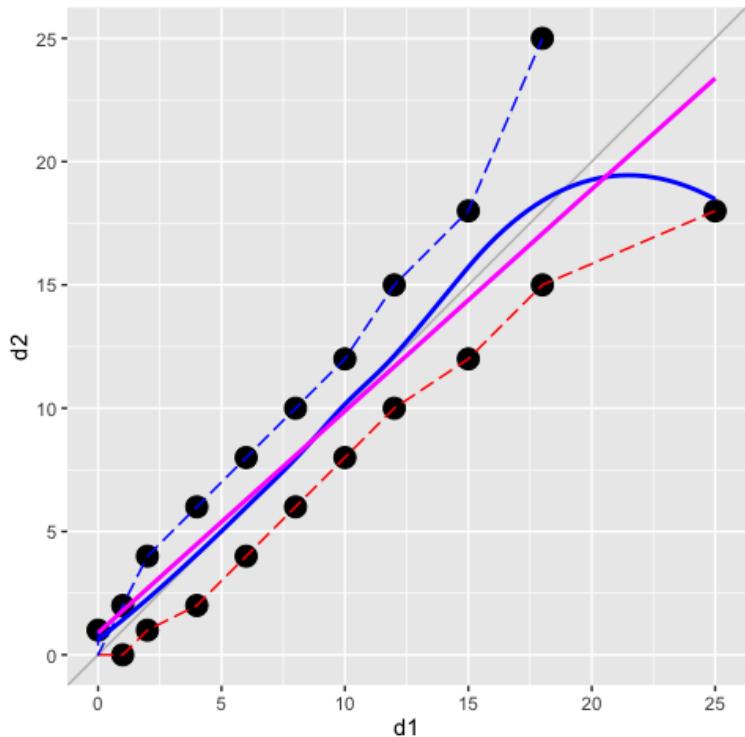
Second, for *twenty* races, where each driver scores each position in the top 10 once, beating his team mate, who once again maximises points below.

```
df=data.frame(d1=numeric(),d2=numeric())
for (i in seq(length(NEWPOINTS))) {
  df=rbind(df,
    data.frame(d1=NEWPOINTS[i],d2=subpoints[i]),
    data.frame(d1=subpoints[i],d2=NEWPOINTS[i]))
}
twowayPointsPerfSketch(df)
```



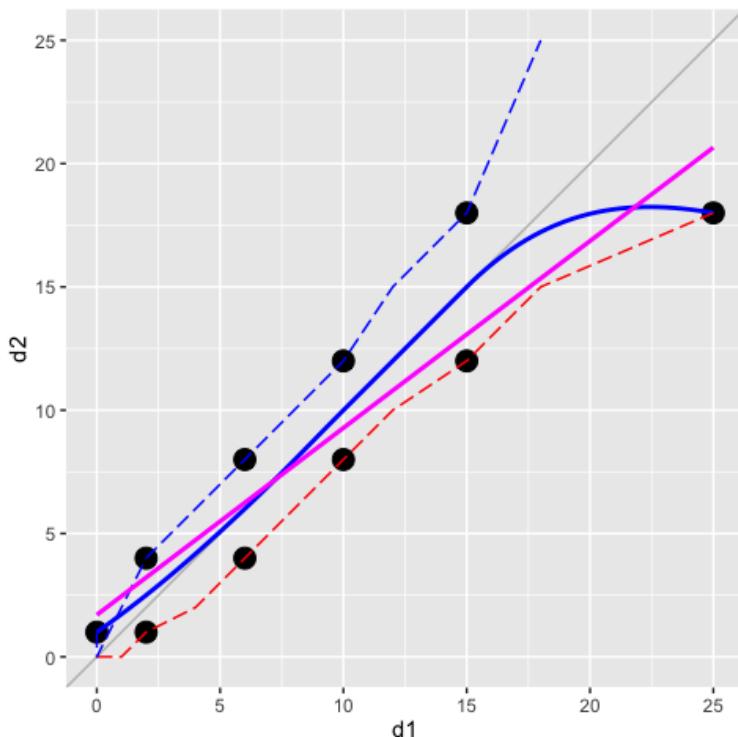
If we add in another best fit line, this time for a simple linear model, we can get a feel for how closely it tends towards the grey equal points line.

```
twoWayPointsPerfSketch(df)+stat_smooth(se=F,method='lm',colour='magenta')
```



Third, this time for just *ten* races, where points are maximised and drivers alternate in terms of who scores the most points.

```
df=data.frame(d1=numeric(),d2=numeric())
for (i in seq(1,length(NEWPOINTS),2)) {
  df=rbind(df,
    data.frame(d1=NEWPOINTS[i],d2=subpoints[i]),
    data.frame(d1=subpoints[i+1],d2=NEWPOINTS[i+1]))
}
twoWayPointsPerfSketch(df)+stat_smooth(se=F,method='lm',colour='magenta')
```



Once again, we see how the best fit lines stay *within* the inner bounds of the points maximisation lines. Generally, if the team is maximising points, the best fit line will be in this inner space in the two-way chart.

The effective misalignment of the axes - consecutively placed drivers score *different* numbers of points - suggests it might be more informative to try to offset or bring the axes into alignment somehow.



Exercise - Rank Performance Charts

The non-linear point distribution (first place taking 25 points, second taking 18 points, and so on) as well as the limited number of points taking positions means that the points performance charts are only informative where the highest placed driver finishes in ninth position or above (so that both drivers have a chance of scoring points.)

To apply this style of chart to the whole field, rather than using points along the axes, we could plot a *rank performance chart* using $(1 + \text{field size} - \text{rank})$ of the drivers. Unclassified drivers should be given a value of 0.

Create a function to plot rank performance charts, or modify the current functions to accept a parameter that allows either a rank performance chart, or a points performance chart to be plotted. Compare data from the same teams using the two approaches. Does the improved linearity of the rank performance chart make it easier to read? Does the improved resolution of the rank based chart - that is, its ability to display information about all classified positions, not just points bearing ones - make the chart more informative?

Summary

In this chapter, we looked at *grid position points productivity* to see how many points a driver might expect to gain from any given starting position on the grid, before exploring the extent to which teams maximise their points hauls, both in general terms by summarising results across all teams and several seasons, and in more detailed way by considering the performance of a single team in a single season.

A *one way points performance chart* can be used to plot a count of team points hauls for a set of races against the points scored by the highest placed driver in the team in each race. A line guide shows the maximum points haul possible given the position of the highest placed driver. The *one way* chart *does not* identify which driver was the higher placed.

A *two way points performance chart* plots a count of points combinations given the (x,y) pairing of points scored by each driver across the specified races. The two way plot *does* identify which driver scored better across the races. In the two way plot, *two* line guides identify the maximum possible points haul in a given race given the number of points scored by the higher placed driver *and* which driver it was.

One problem with both one-way and two-way points performance charts is that they don't reveal which races were responsible for which points hauls, or whether the relative balance of driver superiority evolved over the course of a season.

Another significant issue arises from the eccentric alignment of points across the two axes - it would be cleaner if points maximisation loci fell on a single diagonal, for example.

End of Season Showdown

To keep television audiences interested in F1 throughout a season, the hope is that the Drivers' Championship in particular will go down to the wire, not being decided until the last race of the season. Even without the double points mechanism introduced for the last race of the championship for the 2014 season, Lewis Hamilton's 17 point lead over Nico Rosberg meant that Rosberg could have won the championship with a first or second place finish had Hamilton finished out of the points. With the double points mechanism, however, several other scenarios were possible.

In this chapter, we'll look at a simple model that shows the different championship results for each finishing combination.

Modeling the Points Effects of the Final Championship Race

To start with, we generate a list of the standard points allocations for each finishing position.

```
points=data.frame(pos=1:11, val=c(25,18,15,12,10,8,6,4,2,1,0))
```

To work out which driver would have won for each possible result in the double points scoring final race, we need to find out the points difference between each of the finishing positions. We can represent this information using a square matrix, where the rows represent the finishing position of one of the drivers, and the columns the finishing position of the other driver. The matrix values represent the points difference for the corresponding finishing combination. The diagonal elements, which would represent both cars receiving the same classification, are ignored.

```

pdiff.final=matrix(ncol = nrow(points), nrow = nrow(points))
for (i in 1:nrow(points)){
  for (j in 1:nrow(points))
    if (i==j) pdiff.final[[i,j]]=NA
    else pdiff.final[[i,j]]=2*points[[i,2]]-2*points[[j,2]]
}
pdiff.final

```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]	[,11]
[1,]	NA	14	20	26	30	34	38	42	46	48	50
[2,]	-14	NA	6	12	16	20	24	28	32	34	36
[3,]	-20	-6	NA	6	10	14	18	22	26	28	30
[4,]	-26	-12	-6	NA	4	8	12	16	20	22	24
[5,]	-30	-16	-10	-4	NA	4	8	12	16	18	20
[6,]	-34	-20	-14	-8	-4	NA	4	8	12	14	16
[7,]	-38	-24	-18	-12	-8	-4	NA	4	8	10	12
[8,]	-42	-28	-22	-16	-12	-8	-4	NA	4	6	8
[9,]	-46	-32	-26	-20	-16	-12	-8	-4	NA	2	4
[10,]	-48	-34	-28	-22	-18	-14	-10	-6	-2	NA	2
[11,]	-50	-36	-30	-24	-20	-16	-12	-8	-4	-2	NA

Matrix co-ordinates are described using the convention (*rows, columns*) with the top left element represented as cell (1,1). Reading this matrix, we see that if one driver finished in second place and the other in fourth, the difference in points that they will receive given by locations (2,4) or (4,2) is 6 points.

You should note that there is some asymmetry in the matrix, with the upper right corner displayed as a positive points difference, and the lower left hand corner containing negative points differences. If we associate Rosberg with the columns (that is, the horizontal access) and Hamilton with the rows (that is, the vertical axis) we can see that if Hamilton finished 4th and Rosberg finished 7th (location (4,7)) the points difference is +12, which we read as Hamilton gaining 12 points more than Rosberg.

If Rosberg finished 2nd and Hamilton finished out of the points, modeled as position 11, we read the value at (11,2) as -36, which is to say, Hamilton loses 36 points to Rosberg.

Going in to the final round, Hamilton led Rosberg by 17 points - that is, Hamilton was +17 points up on Rosberg. If we add this amount into each cell, we can see what the points difference would be after each possible combination of race classifications in the final race.

```
pdiff.final+Vectorize(17)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
## [1,]    NA   31   37   43   47   51   55   59   63   65   67
## [2,]     3   NA   23   29   33   37   41   45   49   51   53
## [3,]    -3   11   NA   23   27   31   35   39   43   45   47
## [4,]    -9    5   11   NA   21   25   29   33   37   39   41
## [5,]   -13    1    7   13   NA   21   25   29   33   35   37
## [6,]   -17   -3    3    9   13   NA   21   25   29   31   33
## [7,]   -21   -7   -1    5    9   13   NA   21   25   27   29
## [8,]   -25  -11   -5    1    5    9   13   NA   21   23   25
## [9,]   -29  -15   -9   -3    1    5    9   13   NA   19   21
## [10,]  -31  -17  -11   -5   -1    3    7   11   15   NA   19
## [11,]  -33  -19  -13   -7   -3    1    5    9   13   15   NA
```

Reading this, we see if Rosberg finished in first place, cells $(N,1)$, and if Hamilton finished in second, Hamilton would be +3 up on Rosberg after the race and take the championship; but if Hamilton finished third, he'd be three points down (-3) and would lose the championship.

If Hamilton finished seventh and Rosberg finished third, cell $(7,3)$ shows a result of -1 - Rosberg would have lost by a single point.

Visualising the Outcome

We can provide a more eyecatching view over the data by using colour or additional emphasis to each cell according to which driver would take the championship win.

```
library(reshape)
library(ggplot2)

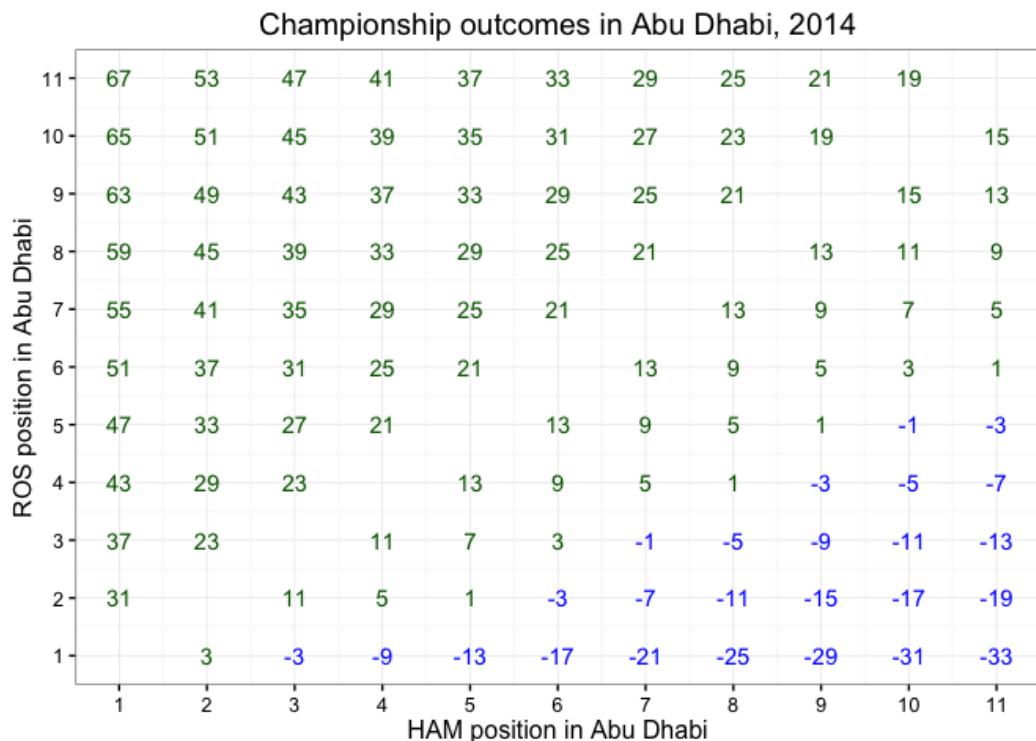
results = melt(pdiff.final+Vectorize(17))

g = ggplot(results)
g = g + geom_text(aes(X1, X2,
                      label=value,
                      col=factor(sign(as.integer(as.character(value))))))
g = g + xlab('HAM position in Abu Dhabi')+ ylab('ROS position in Abu Dhabi')
g = g + labs(title="Championship outcomes in Abu Dhabi, 2014")
g = g + scale_color_manual(values = c('blue','darkgreen'))
```

```

g = g + theme_bw() +theme(legend.position="none")
g = g + scale_x_continuous(breaks=seq(1, 11, 1))
g + scale_y_continuous(breaks=seq(1, 11, 1))

```



Summary

In this chapter, we have seen a simple way of modeling the outcome of a Drivers' Championship race between two drivers with a chance at the championship as they go in to the final race of the season.

An output chart identifies which driver will win or lose given different finishing combinations, as well as the number of points they'll win or lose by.

Charting the Championship Race

Chart types such as lapcharts and race history charts provide a useful, and conventional, way of summarising the evolution of a particular race. But as well as individual races, the seasonal structure of Formula One means we are also offered a championship race.

In this chapter, we'll explore how can we capture elements of the season long race using informative graphical displays. Many of the chart types will have a familiar feel to them, drawing as they do on the charts used to describe individual races. But that is no bad thing - it means we can draw on the lessons we have learned previously when it comes to working out what stories the charts have to tell...

Getting the Championship Data

Perhaps the most obvious way of charting the race to the championship in either the Drivers' or the Constructors' Championship is to plot the total number of points gained to date against each round of the championship for each driver or constructor.

Let's work through an example using the 2014 Championship up to and including round 17. We can pull in the data directly from the online *ergast API*.

```
#Load in the core utility functions to access ergast API
source('ergastR-core.R')

#Get the standings after each of the first 17 rounds of the 2014 season
df=data.frame()
for (j in seq(1,17)){
  dft=seasonStandings(2014,j)
  dft$round=j
  df=rbind(df,dft)
}
#Data is now in: df
```

year	driverId	code	pos	points	wins	car	round
2014	rosberg	ROS	1	25	1	mercedes	1
2014	kevin_magnussen	MAG	2	18	0	mclaren	1
2014	button	BUT	3	15	0	mclaren	1

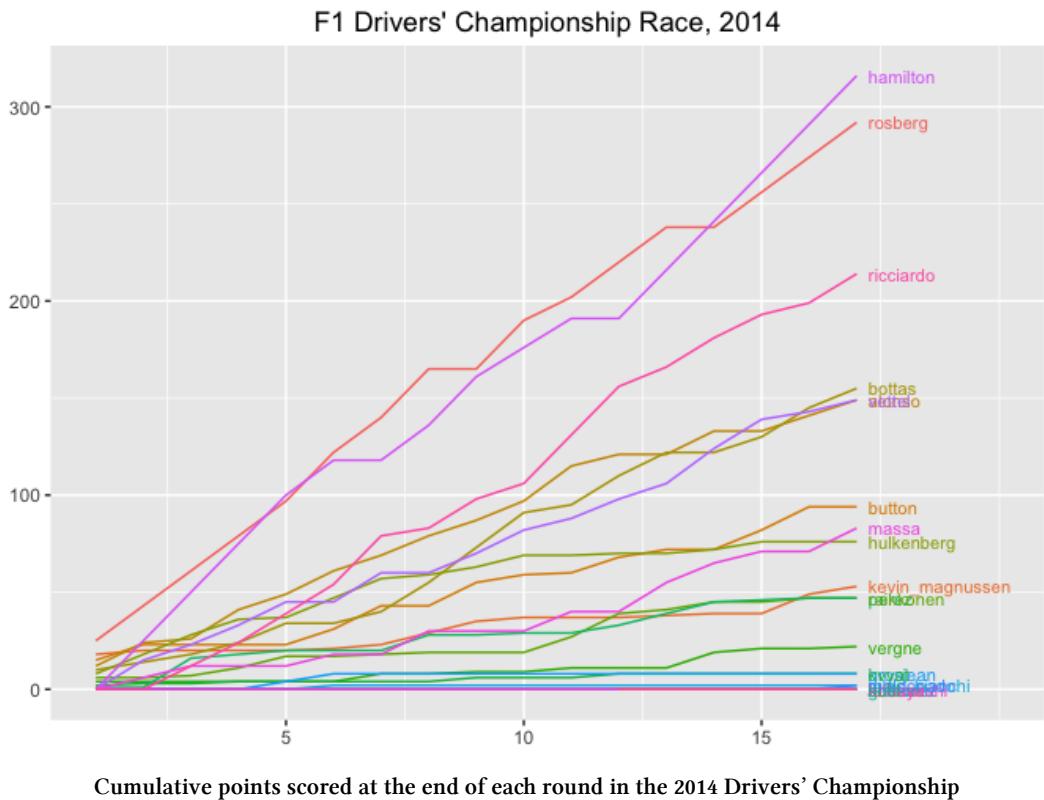
Charting a Championship Points Race

To begin with, I'm going to define some basic chart elements to help with the display of the various charts described in this chapter.

```
library(ggplot2)
library(directlabels)
#The label argument takes one of: code / driverId
#Data for seasons prior to 2005 or so may not include driver codes
championship_race_charter=function(g) {
  #Generate a line chart
  g = g + geom_line()
  #Remove axis labels and colour legend
  g = g + ylab(NULL) + xlab(NULL) + guides(color=FALSE)
  #Add a title
  g = g + ggtitle("F1 Drivers' Championship Race, 2014")
  #Add right hand side padding to the chart so the labels don't overflow
  g + scale_x_continuous(limits=c(1,20), breaks=c(5,10,15))
}
```

Let's now generate our first chart - a plot of points standing of each driver at the end of each round.

```
#All the other elements of the chart definition are the same
g=ggplot(df,aes(x=round,y=points,colour=driverId))
g=championship_race_charter(g)
direct.label(g,list("last.points",cex=0.7,d1.trans(x=x+0.2)))
```



The `directlabels` library handles the placement of the driver code labels at the end of each line, although some of them are overlapping. The line colouring is rather confusing, but used in this case as the basis for grouping by the `direct.label()` function. (Setting the `fill` rather than the `colour` parameter should also satisfy the needs of `direct.label()`.)

What the chart does clearly show however, is how the championship has split into several different battles. At the top, Hamilton and Rosberg, with Ricciardo on his own third. The battle for fourth includes three drivers, although the clashing line colours and overlapping labels makes it hard to read this race clearly. The battle for 7th also includes three cars, but these are more clearly separated. Finally, the battle for 10th also includes three cars, but the points equivalence makes it really hard to see what's going on here. Using different dashed line styles might help to make this clearer, but the the problem of label overlap would still remain.

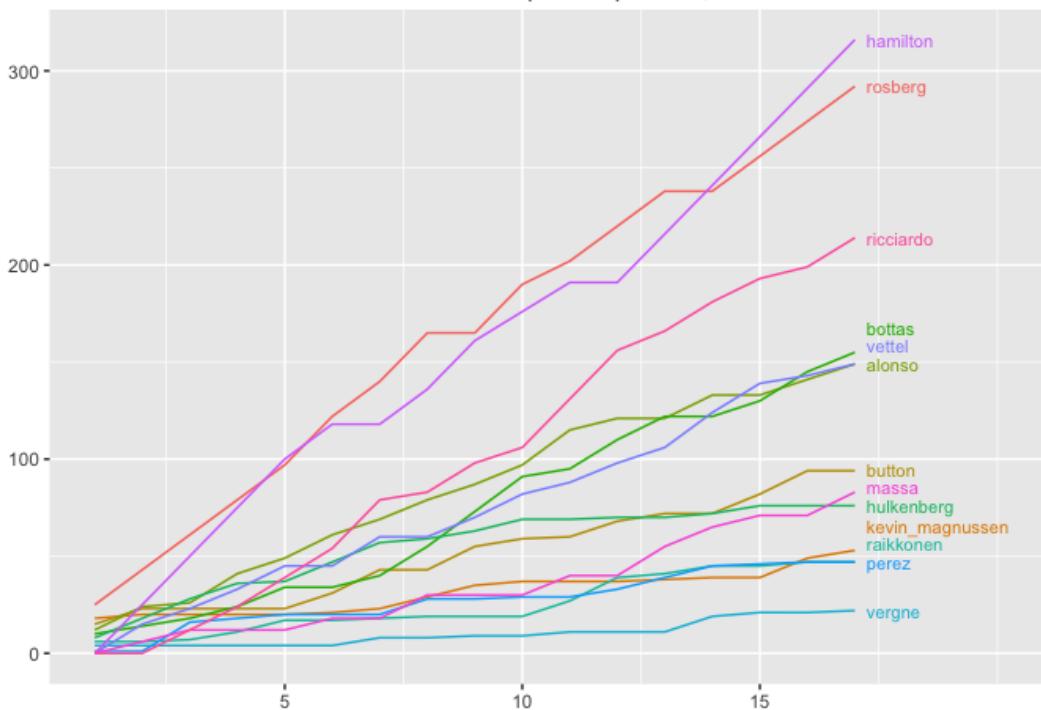
The `directlabels` package can be used to adjust the placement of labels in particular circumstances as the following example shows, where I have limited the display to drivers

scoring at least 10 points. The `bump.up` placement prevents the labels from overlapping.

```
df.tenUp=df[df$round==17 & df$points>=10,]$driverId

#All the other elements of the chart definition are the same
g=ggplot(df[df$driverId %in% df.tenUp,],aes(x=round,y=points,colour=driverId))
g= championship_race_charter(g)
direct.label(g,list('last.bumpup',cex=0.7,d1.trans(x=x+0.2)))
```

F1 Drivers' Championship Race, 2014



Cumulative points scored at the end of each round in the 2014 Drivers' Championship for drivers scoring at least 10 points by round 17

We can now more clearly see the labels, although in situations where points totals in the final round are close there may still to be some ambiguity arising from the placing of the labels.

Probing a Championship Race in More Detail - Adding Guide Lines

As well as charting the whole of a championship race, we can focus in more detail on a particular battle. The following chart shows how we can explore the battle for tenth. In constructing this chart, I have done two things:

- *filter the drivers associated with the battle.* In this example, filter the results to show just the drivers who were contending 10th place at the end of round 10 of the championship;
- *identified the mean number of points calculated over the 10th and 11th positions.* This value then provides a guide line that helps us see how close to the tenth position, in points terms, the driver in 11th position is.

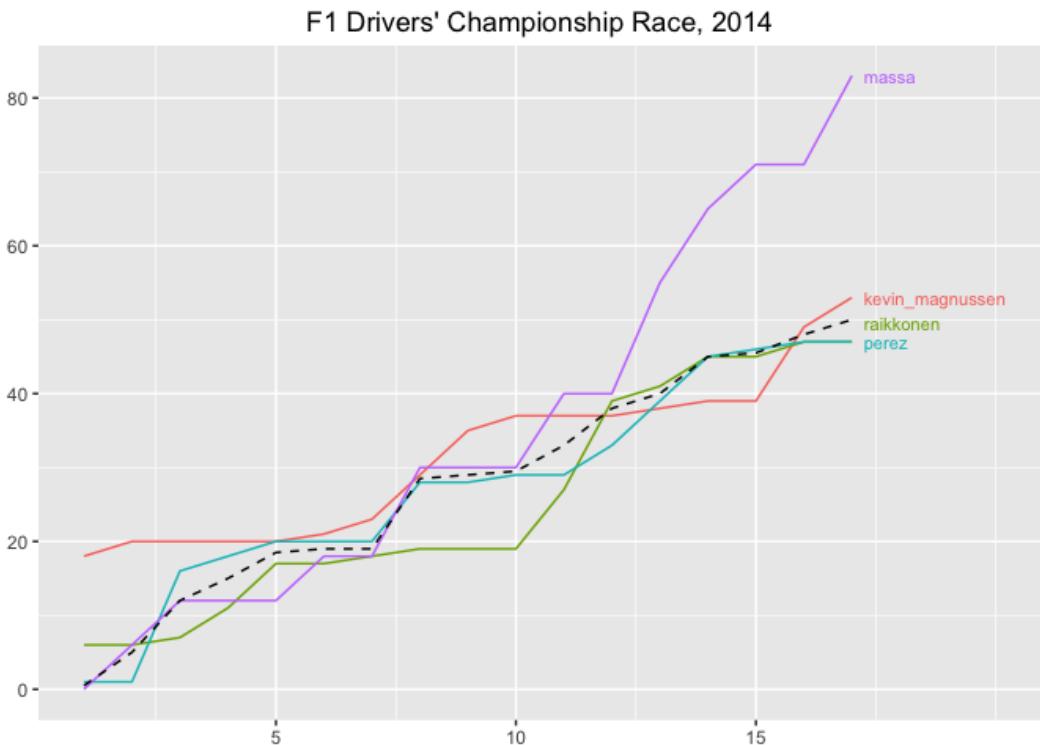
```

library(plyr)
#Generate a guide that is the mean points value of 10th and 11th positions
dfx=ddply(df[df['pos']==10 | df['pos']==11,],
  .(round), summarize, points=mean(points))
#Set the driverId of this line to an empty string
dfx$driverId=''

#Identify the drivers fighting for 10th at the end of round 10
#Include other drivers contending this position earlier in the season
#Ideally, we would automatically identify the positions around this cluster
df.contending = df[df$round==10 & df$pos>=9 & df$pos<=12, 'driverId']
#The following filter identifies drivers fighting for 10th after 17
##df.contending =df[df$round==17 & df$pos>=10 & df$pos<=12, 'driverId']
df.battle=df[df$driverId %in% df.contending,]

#Base chart
g=ggplot(df.battle,aes(x=round,y=points,colour=driverId))
g=championship_race_charter(g)
g=direct.label(g,list('last.bumpup',cex=0.7,d1.trans(x=x+0.2)))
#Add in the guide line
g = g + geom_line(data=dfx, aes(x=round, y=points),
                   col='black', linetype="dashed")
g

```



Once again, the automatic colour selection leaves something to be desired. Using different line styles might also improve the clarity of the charts and make them “photocopy safe”.

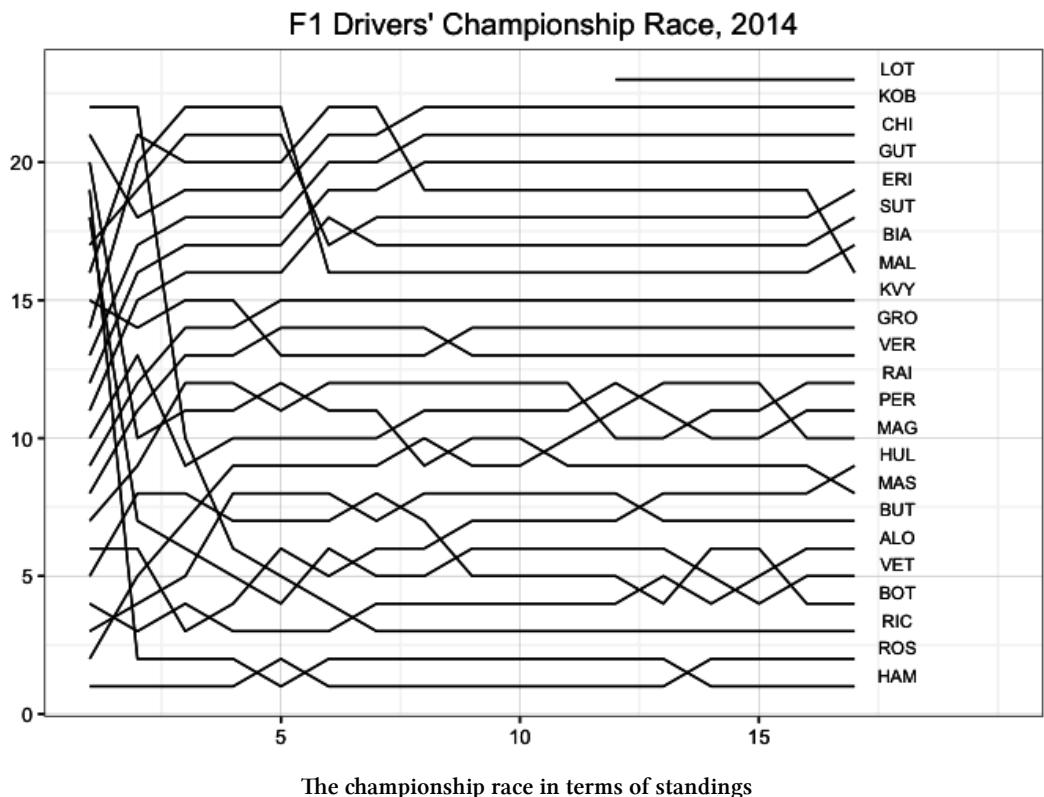
Charting the Championship Race Standings

When displaying the championship race as a points race, it can be quite hard at times to see changes in actual position.

If we view the race as a rank based race, using a chart that in many respects resembles a lap chart as used to describe a single race, we can more clearly see changes in the standings by virtue of lines clearly crossing.

If we used the fixed size three letter driver codes, we can label the lines using a simple `geom_text()` method.

```
#The lap chart style chart - plotting position rather than points
g = ggplot(df, aes(x=round, y=pos, group=driverId))
g = championship_race_charter(g)
#If we know the size of the labels, we can easily fashion our own placement
g = g + geom_text(data=df[df['round']==max(df['round'])],,
                   aes(x=round+0.9,label=code),size=3, vjust=-0.4)
g = g+theme_bw()
g
```



Although this chart clearly shows how *positions* change over the course of the season, it does not really give us any information about what happened in *points* terms during each round. We can address this by annotating the chart so that it communicates the following information:

- the number of points earned by each driver in a particular round;

- the points standing of each driver at the end of each round.

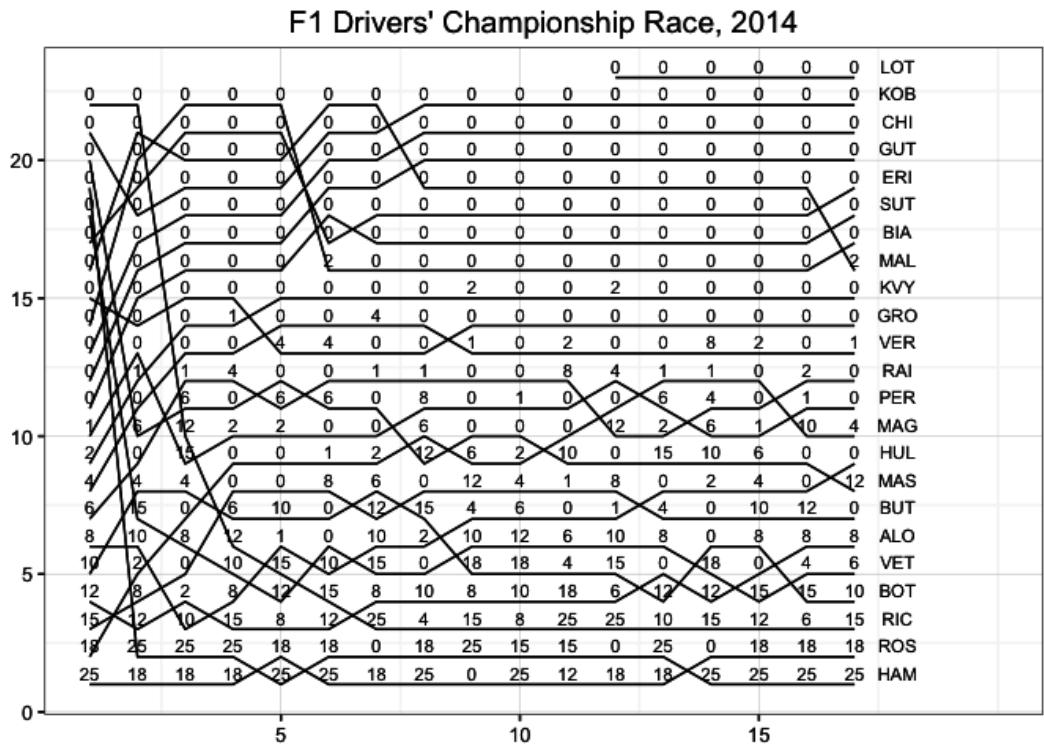
Let's start by considering the number of points won by each driver at each round. The call we made to the *ergast* API doesn't provide this information directly, but we can easily derive it: sort the data into increasing round then calculate the difference between consecutive points totals.

```
#Sort by ascending round
df = arrange(df, round)
#Derive how many points each driver scored in each race
df = ddply(df, .(driverId), transform, racepoints=diff(c(0,points)))
```

round	points	racepoints
1	25	25
2	43	18
3	61	18

We can now use the *points scored per race* information to annotate the chart:

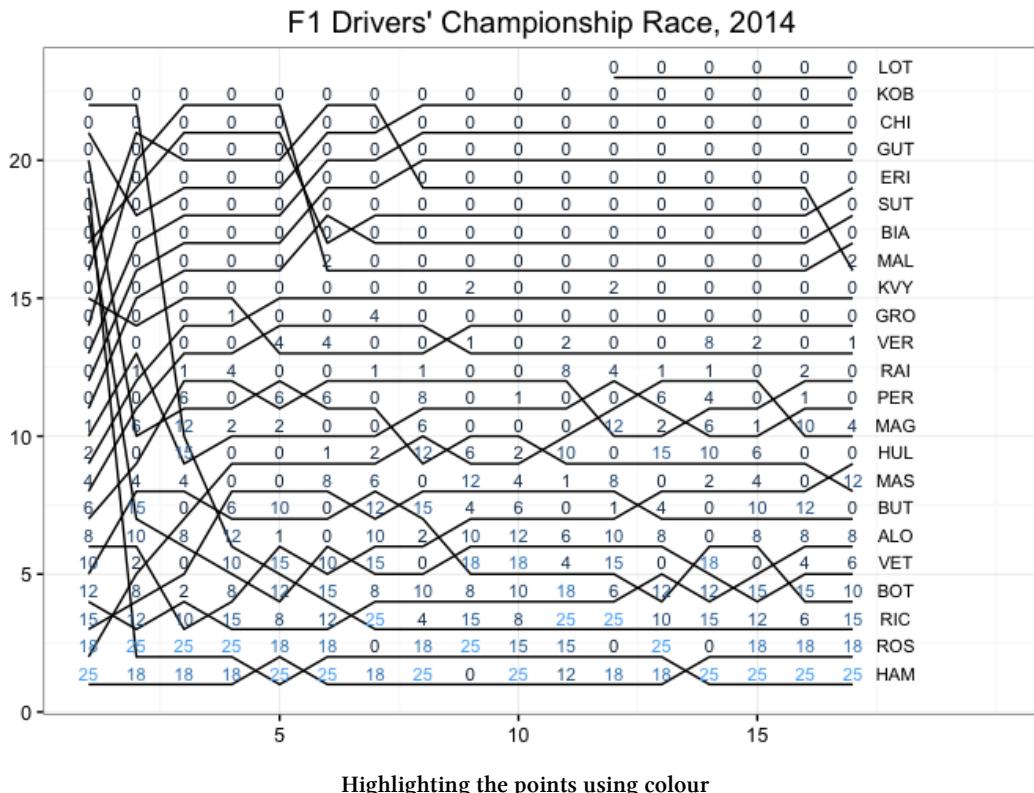
```
g+geom_text(data=df,aes(label=racepoints),vjust=-0.4,size=3)
```



Annotating the championship standings chart with points won per race

That's okay, insofar as it goes, but we could perhaps add in colour relative to the number of points scored in each race to highlight the higher values a little more clearly.

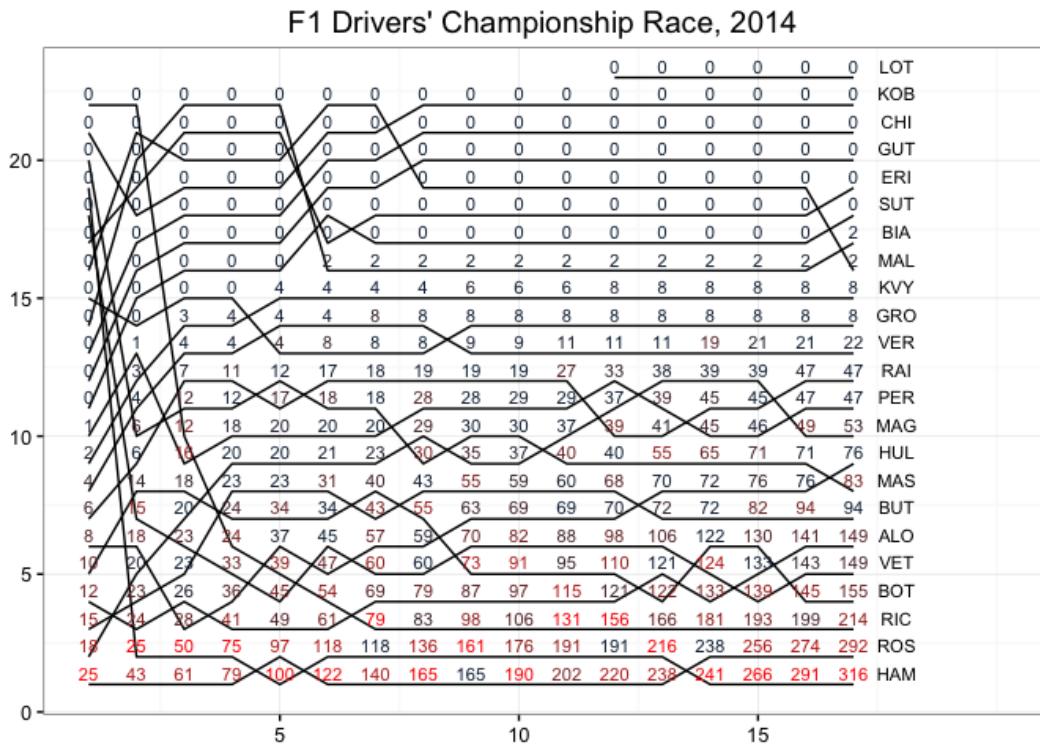
```
g + geom_text(data=df, aes(label=racepoints, col=racepoints), vjust=-0.4, size=3)
```



The default colour scheme scales from black to light blue. The higher values look a little washed out and the graphic doesn't work particularly well when printed in black and white, so it might be worth exploring other colour mappings to highlight the higher values more clearly.

Annotating the chart with points scored per race helps us see how well each driver fared in a particular race, but the chart does not give us a sense of how many points separate drivers in the championship standings at the end of each round. We can address this by using the total number of championship points scored to date as the text label, preserving an indication of the number of points awarded for each race by using the colour dimension.

```
g = g + geom_text(data=df, aes(label=points, col=racepoints), vjust=-0.4, size=3)
#Colour scheme for the points scored in each race
g + scale_color_continuous(high='red')
```



Annotating the championship standings chart with championship points at each round, coloured by points earned per race

This chart now has the advantage of communicating standings over the course of the season (via the lines), standing changes (the crossed lines), the total number of championship points at the end of each round (from the numerical label), and a comparative indication of the points earned during each round (from the numerical label colour).

Looking down a column, we can compare the number of points separating drivers in the drivers championship at the end of each round. From the colour field we can see how drivers placed next to each other compared in terms of points awarded in each round. Looking along a line, we can (if necessary) calculate the number of points obtained in a particular round as a simple subtraction going from one round to the next.

Summary

In this chapter, we have started to explore some of the ways in which we can graphically review a season in terms of the race for a Drivers' Championship. The combination of lapchart

style graphics and text labels that record the accumulated championship points for each driver at the end of each round shows how close the championship was in terms of points scored across the season. The use of colour further helps to guide the eye towards points scoring drivers within each race.

Conclusion

Data is everywhere. *Looking* at data, whether through simple summary statistics, data visualisations, or via complex predictive models, provides us with opportunities for telling stories about the world that might otherwise be missed.

In this book we have looked at a wide range of techniques for wrangling, and visualising, F1 related datasets.

I am well aware that this book may well fall short in terms of the actual stories it tells about particular races, championships or career histories, or how to read stories, in detail, from some of the charts shown. Many of the sections would benefit from detailed examples of how to actually read the charts. However, I have tried to structure the data wrangling elements in terms of how to structure data and charts in order to answer particular questions or tease out particular insights. The stories themselves come from learning how to read the colours, shapes and positioning of the marks on the page, and starting to recognise particular patterns within them that reveal particular sorts or story or story feature.

Now that the code is in place, this project has an opportunity to move on, deploying the tools and techniques developed in these pages in the analysis of F1 races and championships on a race by race basis. The f1datajunkie⁷⁰ blog will be the home to my own explorations in this area.

There are three levels of wider success, or *impact*, I think, for the various techniques and strategies described in this book as they particularly relate to Formula One and motorsport in particular, and other sports in general:

1. They get adopted by teams and drivers for tracking and helping to improve performance, which I fear is unlikely - I'm sure the teams have many more tried and tested tools and techniques at their disposal.
2. They are used by the media and PR teams for reporting and explaining motorsport news and help contribute to the discovery and telling of motorsport stories.
3. They provide grist to the mill for F1 stats fans looking for things to do with the data.

⁷⁰<http://www.f1datajunkie.com/>

If nothing else, I hope that this book has inspired *you* to start having conversations with the data, conversations that will reveal compelling stories about this fast paced technological sport, that inspires many of us with its short lead times and constant innovation. Whether by learning to read the charts developed in these pages, or using the techniques or ideas contained herein as a springboard to creating your own analytical tools, I hope you have fun with the data!

This book started off it's published life as a public work in progress on Leanpub.com⁷¹, with many chapters containing elements that were originally presented in note or draft form. As such, the structure and organisation of the book took on a life of its own and may yet continue to evolve.

⁷¹<http://leanpub.com>

Bibliography

- Ganesh, T. V. (2016) Cricket analytics with cricket⁷², Leanpub, 2016.
- Albert, J. (2008) “Streaky hitting in baseball.” Journal of Quantitative Analysis in Sports 4(1).
- Allen, James (2014) The secret of undercut and offset⁷³, UBS Formula1 microsite, May 2, 2014.
- Berkowitz, J. P., Depken, C. A., & Wilson, D. P. (2011). When going in circles is going backward: Outcome uncertainty in NASCAR. *Journal of Sports Economics*, 12(3): 253-283.
- Burn-Murdoch, J. (2015) Who are the best ever Formula One drivers?⁷⁴ FT Interactive Graphics Site, 29/11/2015
- Castellucci, F., Pica, G. & Padula, M. (2011) *The age-productivity gradient: evidence from a sample of F1 drivers* by, Labour Economics 18.4: 464-473 (also available as Ca' Foscari University of Venice, Department of Economics, Working Paper No. 16/WP/2009⁷⁵).
- Marchi, M. & Albert, J. (2013) **Analyzing Baseball Data with R**⁷⁶.
- Mizak, D., Neral, J., & Stair, A. (2007). The adjusted churn: an index of competitive balance for sports leagues based on changes in team standings over time. *Economics Bulletin*, Vol. 26(3) pp. 1-7⁷⁷.
- Phillips, A. (2014) Building a Race Simulator⁷⁸, F1Metrics blog, 3/10/2015.
- Phillips, A. (2015) A Reconstructed History of Formula 1⁷⁹, F1Metrics blog, 21/8/2015.
- Saward, Joe (2011). *Lap Charts*⁸⁰, Joe Blogs F1, 14/4/2011
- Tufte, Edward R. (2006) “Beautiful evidence.” Graphics Press, New York.
- Wilkinson, L. (2006) *The grammar of graphics*, Springer Science & Business Media.

⁷²<https://leanpub.com/cricketr>

⁷³<http://www.ubs.com/microsites/formula1/en/james-allen/the-secret-of-undercut-and-offset.html>

⁷⁴<https://ig.ft.com/sites/the-best-f1-drivers-ever/>

⁷⁵http://www1.unive.it/media/allegato/DIP/Economia/Working_papers/Working_papers_2009/WP_DSE_castellucci_pica_padula_16_09.pdf

⁷⁶<http://baseballwithr.wordpress.com/about/>

⁷⁷<http://core.kmi.open.ac.uk/download/pdf/6420748.pdf>

⁷⁸<https://f1metrics.wordpress.com/2014/10/03/building-a-race-simulator/>

⁷⁹<https://f1metrics.wordpress.com/2015/08/21/a-reconstructed-history-of-formula-1/>

⁸⁰<http://joesaward.wordpress.com/2011/04/14/lap-charts/>

Wickham, H. (2010) *A layered grammar of graphics*](<http://vita.had.co.nz/papers/layered-grammar.html>), Journal of Computational and Graphical Statistics 19(1) pp.3-28.

Wickham, H. (2011). The Split-Apply-Combine Strategy for Data Analysis *Journal of Statistical Software*, April 2011, Volume 40(1)⁸¹

Wickham, H. (2014). Tidy Data. *Journal of Statistical Software*, August 2014, Volume 59(10)⁸²

The Numbers Game by Chris Anderson and David Sally

⁸¹<http://www.jstatsoft.org/article/view/v040i01>

⁸²<http://www.jstatsoft.org/article/view/v059i10>

Appendix - Converting the ergast Database to SQLite

This recipe describes how to convert the original MySQL export file of the ergast database to a SQLite3 format.

From the ergast website download area⁸³, download the ANSI version of the database. This has all the required options for the conversion script published at <https://gist.github.com/esperlu/943776>⁸⁴ (archive copy: <https://gist.github.com/psychemedia/8519869>) which reprinted below.

Although we will be using sqlite to query the ergast data, you will need to use MySQL to run the conversion script. You will also need to ensure you have sqlite3 installed.

- Download the ANSI version of the ergast database export from <http://ergast.com/mrd/db>⁸⁵
- Create an empty database in MySQL, for example: *ergast*
- Import the downloaded ergast database export file into the *ergast* database
- Download the conversion script as *mysql2sqlite.sh*
- In the terminal, *cd* into the folder containing *mysql2sqlite.sh*
- Run the conversion script using a shell (command line) command of the form:

```
./mysql2sqlite -u root ergast | sqlite3 ergastdb.sqlite
```

You should now have a copy of the ergast database available as the sqlite database *ergastdb.sqlite*.

⁸³<http://ergast.com/mrd/db>

⁸⁴<https://gist.github.com/esperlu/943776>

⁸⁵<http://ergast.com/mrd/db>

```

#!/bin/sh

##Originally from: https://gist.github.com/esperlu/943776
#Author: Jean-Luc Lacroix [Github user: esperlu]

##Archived at: https://gist.github.com/psychemedia/8519869

#DATA FILE: from http://ergast.com/mrd/db download the ANSI database
##Import into MySQL eg as database ergast
##In the terminal cd into the folder containing this file then enter eg:
## ./mysql2sqlite -u root ergast | sqlite3 ergastdb.sqlite

# Converts a mysqldump file into a Sqlite 3 compatible file. It also extracts the MyS\QL ^KEY xxxxx^ from the
# CREATE block and create them in separate commands _after_ all the INSERTs.

# Awk is chosen because it's fast and portable. You can use gawk, original awk or even the lightning fast mawk.
# The mysqldump file is traversed only once.

# Usage: $ ./mysql2sqlite mysqldump-opts db-name | sqlite3 database.sqlite
# Example: $ ./mysql2sqlite --no-data -u root -pMySecretPassword myDbase | sqlite3 database.sqlite

# Thanks to and @artemyk and @gkuenning for their nice tweaks.

mysqldump --compatible=ansi --skip-extended-insert --compact "$@" | \
awk '
BEGIN {
    FS=",$"
    print "PRAGMA synchronous = OFF;"
    print "PRAGMA journal_mode = MEMORY;"
    print "BEGIN TRANSACTION;"
}

# CREATE TRIGGER statements have funny commenting. Remember we are in trigger.
/^\\/*.*CREATE.*TRIGGER/ {
    gsub( /*.*TRIGGER/, "CREATE TRIGGER" )
    print
}

```

```

    inTrigger = 1
    next
}

# The end of CREATE TRIGGER has a stray comment terminator
/END \*/;; { gsub( /\*\//, "" ); print; inTrigger = 0; next }

# The rest of triggers just get passed through
inTrigger != 0 { print; next }

# Skip other comments
/^\/\*/ { next }

# Print all `INSERT` lines. The single quotes are protected by another single quote.
/INSERT/ {
    gsub( /\\\047/, "\047\047" )
    gsub( /\n/, "\n")
    gsub( /\r/, "\r")
    gsub( /\", \")
    gsub( /\\\\", \")
    gsub( /\\\032/, "\032")
    print
    next
}

# Print the `CREATE` line as is and capture the table name.
/^CREATE/ {
    print
    if ( match( $0, /"[^"]+/" ) ) tableName = substr( $0, RSTART+1, RLENGTH-1 )
}

# Replace `FULLTEXT KEY` or any other `XXXXX KEY` except PRIMARY by `KEY`
/^ [^"]+KEY/ && !/^ PRIMARY KEY/ { gsub( .+KEY/, " KEY" ) }

# Get rid of field lengths in KEY lines
/ KEY/ { gsub( /\([0-9]+\)/, "" ) }

# Print all fields definition lines except the `KEY` lines.
/^ / && !/^(`KEY|`)/ {
    gsub( /AUTO_INCREMENT|auto_increment/, "" )
    gsub( /(CHARACTER SET|character set) [^ ]+/, "" )
    gsub( /DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP|default current_timestamp|mp on update current_timestamp/, "" )
}

```

```
gsub( /(COLLATE|collate) [^ ]+ /, "") )
gsub(/(ENUM|enum)[^)]+\)/, "text ")
gsub(/(SET|set)\([^\)]+\)/, "text ")
gsub(/UNSIGNED|unsigned/, "")
if (prev) print prev ","
prev = $1
}

# `KEY` lines are extracted from the `CREATE` block and stored in array for later pri\
nt
# in a separate `CREATE KEY` command. The index name is prefixed by the table name to
# avoid a sqlite error for duplicate index name.
/^(` KEY|`);)/ {
    if (prev) print prev
    prev=""
    if ($0 == ";"){
        print
    } else {
        if ( match( $0, /\\"[^"]+/" ) ) indexName = substr( $0, RSTART+1, RLENGTH-1 )
        if ( match( $0, /\\"[^()]+/" ) ) indexKey = substr( $0, RSTART+1, RLENGTH-1 )
        key[tableName]=key[tableName] "CREATE INDEX \"\" tableName \"_ indexName \"\" ON \"\" \
tableName \"\" (" indexKey ");\n"
    }
}

# Print all `KEY` creation lines.
END {
    for (table in key) printf key[table]
    print "END TRANSACTION;"
}
'

exit 0
```