Wrangling F1 Data With R

A Data Junkie's Guide

Tony Hirst

Wrangling F1 Data With R

A Data Junkie's Guide

Tony Hirst

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

This version was published on 2016-04-02

Tweet This Book!

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

The suggested hashtag for this book is #f1datajunkie.

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

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

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

Acknowledgements

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

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

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

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

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

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

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

Errata

An update of the RSQLite package to version 1.0.0 (25.10/14) 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

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/psychemedia/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 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 under 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 of statistical analysis. R also provides rich support for the production of high quality statistical charts and graphics and is increasingly used in the production of complex data visualisations.

The RStudio IDE is cross-platform application available in a free, open source edition as well as commercially supported versions. Whilst capable of running as a standalone desktop application, RStudio can also run as a server, making the IDE available via a web browser with R code executing on the underlying server. This makes packaging RStudio in a virtual machine, running it as a service, and accessing it through a browser on a host machine, a very tractable affair (for example, RStudio AMI shared by Louis Aslett[1] or Running RStudio via Docker in the Cloud[2]). 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 API[3] 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

[1] http://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 site[4] 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 F1[7] site publish a summary list of race penalties[8] awarded during the course of a race weekend, and then use this information to generate a visualisation of the penalties. Whilst not broken down as *data*, it is possible to make use of the common way in which the penalties are described to parse out certain "data elements" from the penalty descriptions.

[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 repository[9]. The routines in this library can be used to request data from the *ergast API* in JSON form, and then cast it into an R data frame.

Historical data for all *complete* seasons to date is available as a MySQL database export file that is downloadable from the *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 to the MySQL export file to a SQLite database file. This simple database solution allows R to connect to the SQLite database directly. The appendix *Converting the ergast Database to SQLite* describes how to generate a sqlite3 version of the database from the original MySQL data export file. *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/psychemedia/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/psychemedia/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 intelligentF1[10] 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/psychemedia/wranglingf1datawithr/blob/master/src/ergastR-core.R
[10] http://intelligentf1.wordpress.com/

season, fuel saving. The James Allen On F1[11] features a strategy review of each race a day or two after each race.

Applied mathematician Andrew Phillips' F1Metrics blog[12] 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, F1fanatic[13] 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 Chancery[15] 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

[11] jamesallenonf1.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 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 - github[17].

If you would like to buy this book, or make a donation to support its development, please visit Wrangling F1 Data with R - leanpub[18].

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

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

Getting Started with RStudio

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

To get RStudio up and running, you need to:

- **download and install R** from the R Project for Statistical Computing[19]. The download area[20] 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.org[21].

Configuring the server edition of RStudio so that you can run it as a service and access it via a web browser is a rather more involved process and will not be covered in this book. If you would like to run the server, the easiest way is probably to define a virtual machine (for example, as described in Running RStudio via Docker in the Cloud[22]) or make use of an Amazon machine image, such as the RStudio AMI shared by Louis Aslett[23].

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

Getting Started with R

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

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

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

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

```
getwd()
```

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

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

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

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

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

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

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

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

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

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

The data is currently stored in a CSV - comma separated variable/value - file, a simple text based file format for sharing tabular data that you can find here: winners2014start-csv.csv[24]. You can view the raw data file by clicking on the <> icon in the file title bar.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
str(winners)
```

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

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

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

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

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

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

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

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

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

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

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

```
summary(winners)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
str(winners)
```

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

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

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

```
colnames(winners)
```

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

Vectorised Computation

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

Adding New Columns

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

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

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

tmp.df
```

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

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

```
winners2=winners

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

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

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

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

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

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

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

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

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

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

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

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

Filtering data in a dataframe

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

	driverId	race
1	rosberg	Australia
4	hamilton	Malaysia
7	hamilton	Bahrain

Or more cryptically:

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

	driverId	race
1	rosberg	Australia
4	hamilton	Malaysia
7	hamilton	Bahrain

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

Sorting dataframes

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

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

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

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

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

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

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

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

Merging Dataframes

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

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

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

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

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

Reshaping Data

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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


head( wide.df, n=3 )


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

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

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

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


## Using driverId as id variables
```

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

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

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

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

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

Split-Apply-Combine

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

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

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

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

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

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

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

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

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

kable(winners.new)
```

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

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

Summary

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

RStudio is a very powerful tool and can be used to author a wide range of document types that incorporate executable R code and the results of its execution. For example, you can use RStudio to produce slideshows that incorporate R code and the results of running it, or HTML or PDF documents generated from reusable templates. RStudio can also be used to support the development of R packages, which is something I need to learn how to do! To learn more about RStudio, check out the RStudio documentation[25], or one of the increasing number of books on RStudio[26].

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

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

Exercises

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

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

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

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

```
## Warning: package 'knitr' was built under R version 3.1.3
```

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

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

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

There are also several third party applications that have been built on top of the *ergast* data. For further details, see the ergast Application Gallery[27].

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

For convenience, I will tend to refer to *accessing the ergast API* when I mean calling the online API, and *accessing the ergast database* when it comes to querying a local database. However, you should not need to have to install the database for the majority of examples covered in this book - the API will work fine (and is 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!

[27] 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. *For more details, see the appendix.* The library can be found in the file ergastR-core.R[28] 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;

[28] https://gist.github.com/psychemedia/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.

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

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* file[29] 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')
```

```
## Warning: package 'plyr' was built under R version 3.1.3
```

```
#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)
```

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

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

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

#The knitr library contains a handy function, kable(), for generating tabular 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

round	racename	circuitId
6	Monaco Grand Prix	monaco

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

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

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

Having access to laptime data is essential for many race reports. 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/RACE` (API_PATH is set to *http://ergast.com/api/f1/*). For the winner, the construction of the URL thus includes the term **1**.json).We could create a more general function that makes a call for information relating to an arbitrary position, not just first place by parameterising this part of the URL's construction.

That is, we might try something of the form:

```
#Pass in a race position, by default setting it to first place
racePosition = function (year, raceNum, racePos=1) {
  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 (*hamiliton*, *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	constructor	grid	laps	status	fastlapnum	fastlaptime	fastlaprank	code
alonso	14	4	ferrari	4	56	Finished	47	104.165	3	ALO
bottas	77	8	williams	18	56	Finished	31	105.475	9	BOT
button	22	6	mclaren	10	56	Finished	47	106.039	11	BUT

If the columns you want to merge on actually have *different* names, they can 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  strtime rawtime    cuml   diff code
## 1   alonso   1        5 1:56.440 116.440 116.440     NA  ALO
## 2   alonso   2        5 1:49.154 109.154 225.594 -7.286  ALO
## 3   alonso   3        5 1:48.219 108.219 333.813 -0.935  ALO
```

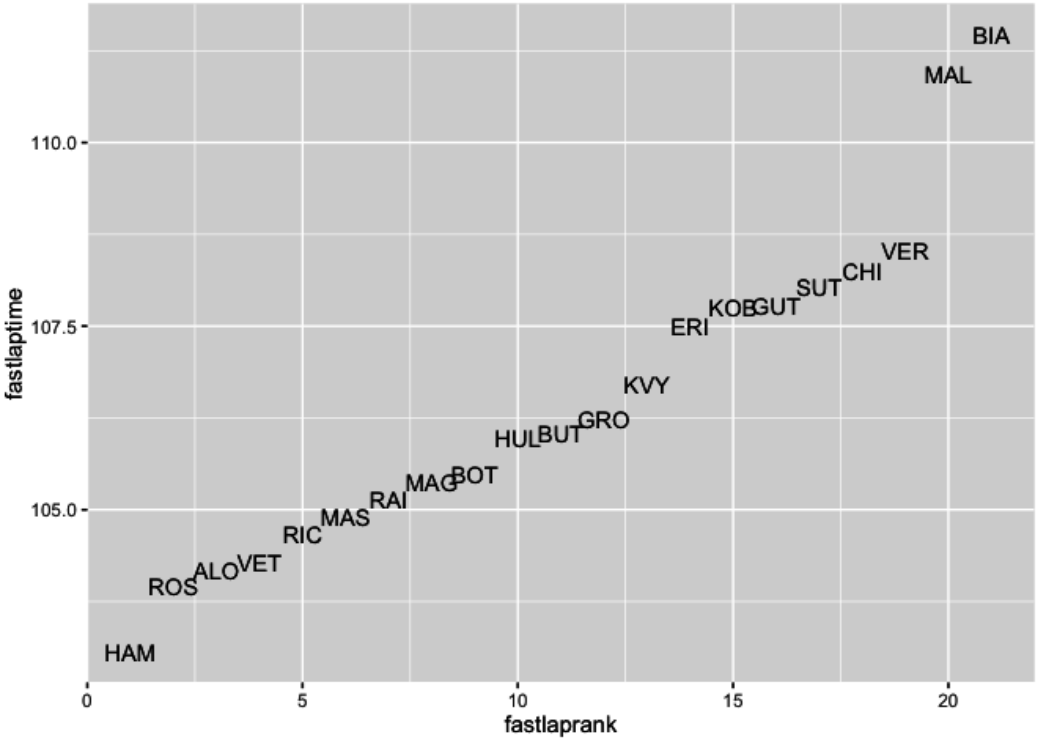
Having the three letter code available in the dataframe directly means we can access it easily when it comes to producing a chart. For example, we might plot the fastest lap time against the fastest lap rank, using the code to identify each point:

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

```
## Warning: package 'ggplot2' was built under R version 3.1.3
```

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

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



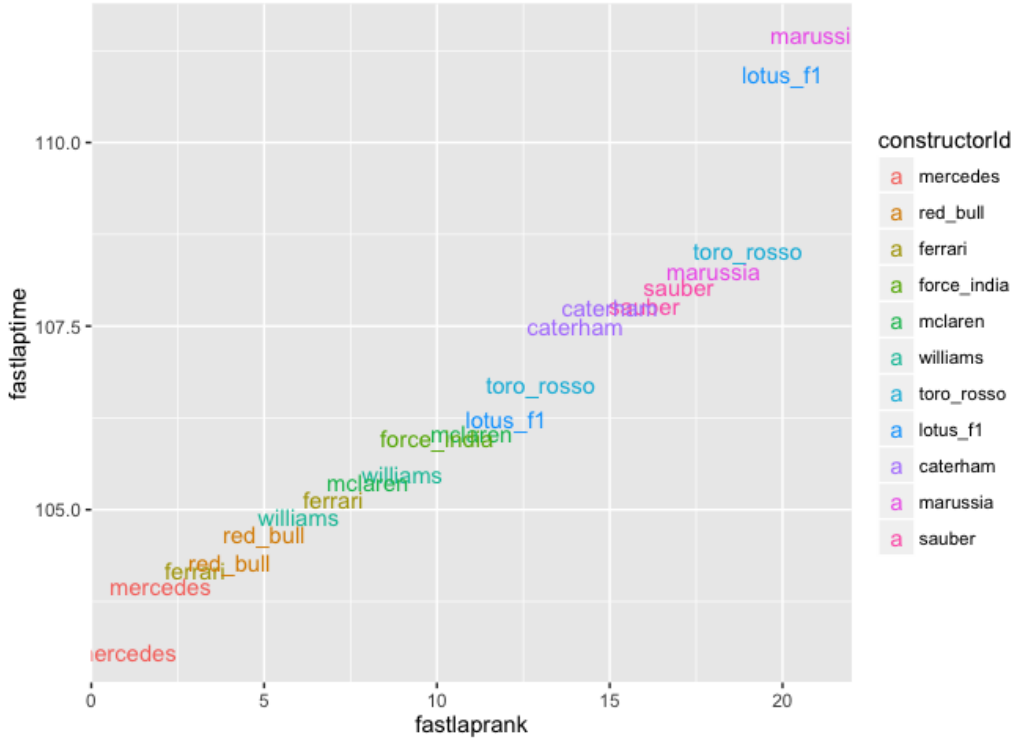
Text plot showing fast lap time versus rank

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

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

Rather than use the driver code for the labels, let's use `constructorId`, further colouring the labels based on the value of the `constructorId`.

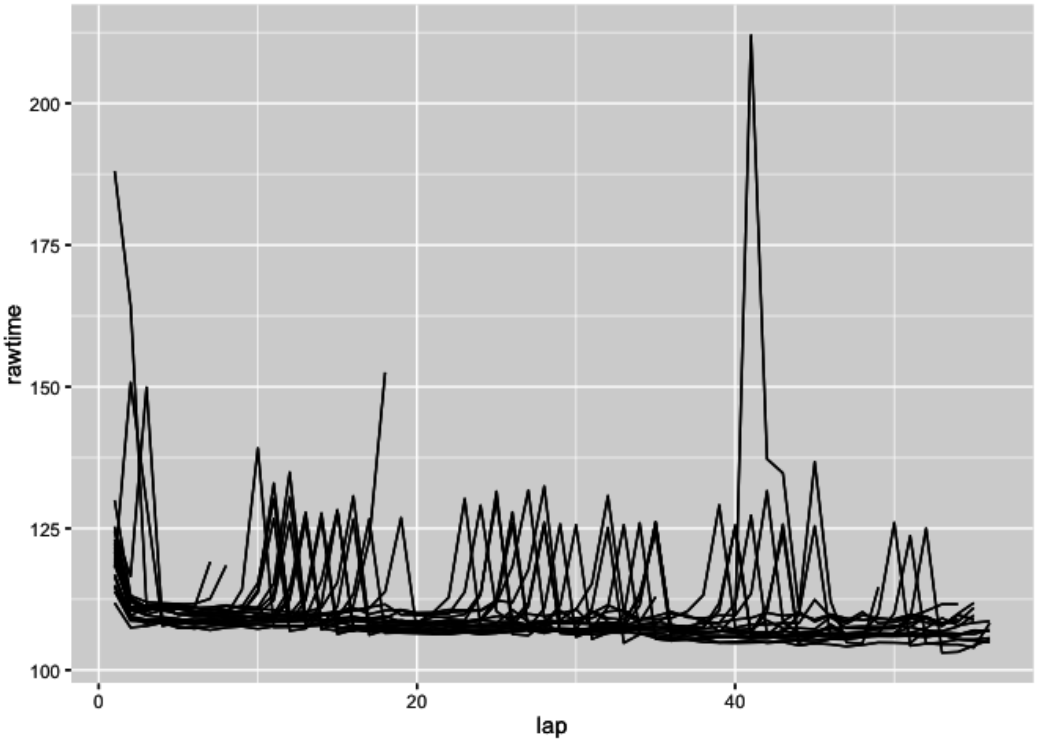
```
g = ggplot(results.df)
g = g + geom_text(aes( x=fastlaprank, y=fastlaptime,
                       label=constructorId, col=constructorId))
g
```



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 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 file[30]. 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 chanpionship; the results for a particular race; the winning driver from a particular race (and by extension of that function, the driver finishing in a any specified position of a given race); the lap times for a particular race; and the careerwise standings for a particular driver in terms of their position at the end of each season in which they competed.

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

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

Exercises

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

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

Practice Session Utilisation

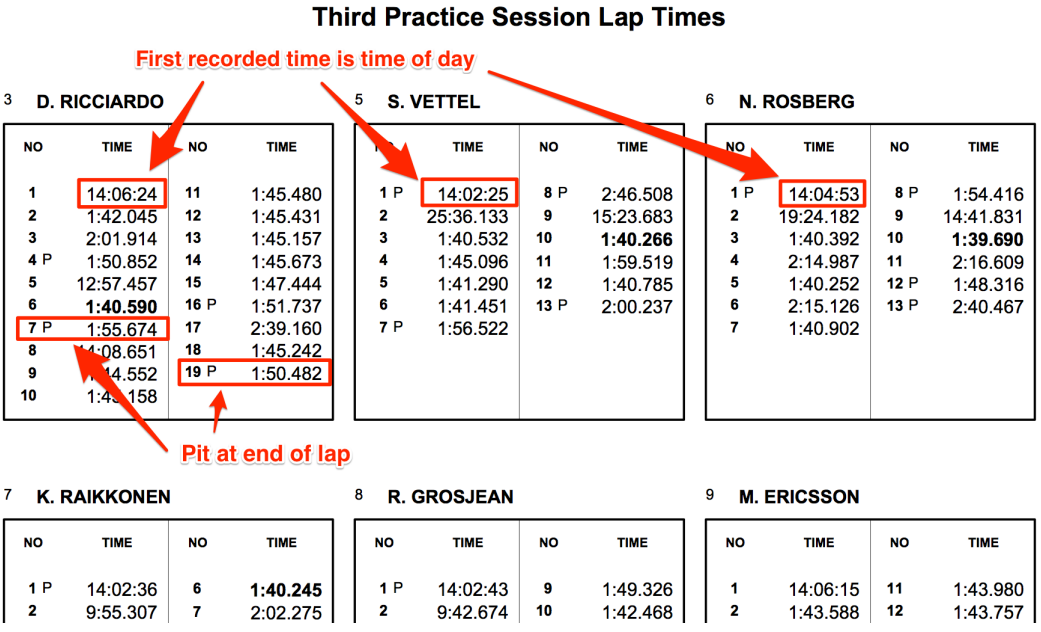
For the teams, the three practice sessions that open up a race weekend provide an opportunity to set the car up for the weekend. For the data junkie, timing sheets published by the FIA provide details of all the laptimes recorded by each driver. Using this data, we can chart the extent to which each 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)*[31] *or Tabula (free/open source desktop tool)*[32].

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

[31] https://pdftables.com/

[32] http://tabula.technology/



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 file[33]). We can then present the data in the following form:

[33] https://gist.githubusercontent.com/psychemedia/11187809/raw/f12015test.csv

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

lapNumber	laptime	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 `laptime` is the laptime 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 laptime (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"= "KVY", "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 laptime recorded so far by a particular driver ("green" times), and the best laptime 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.

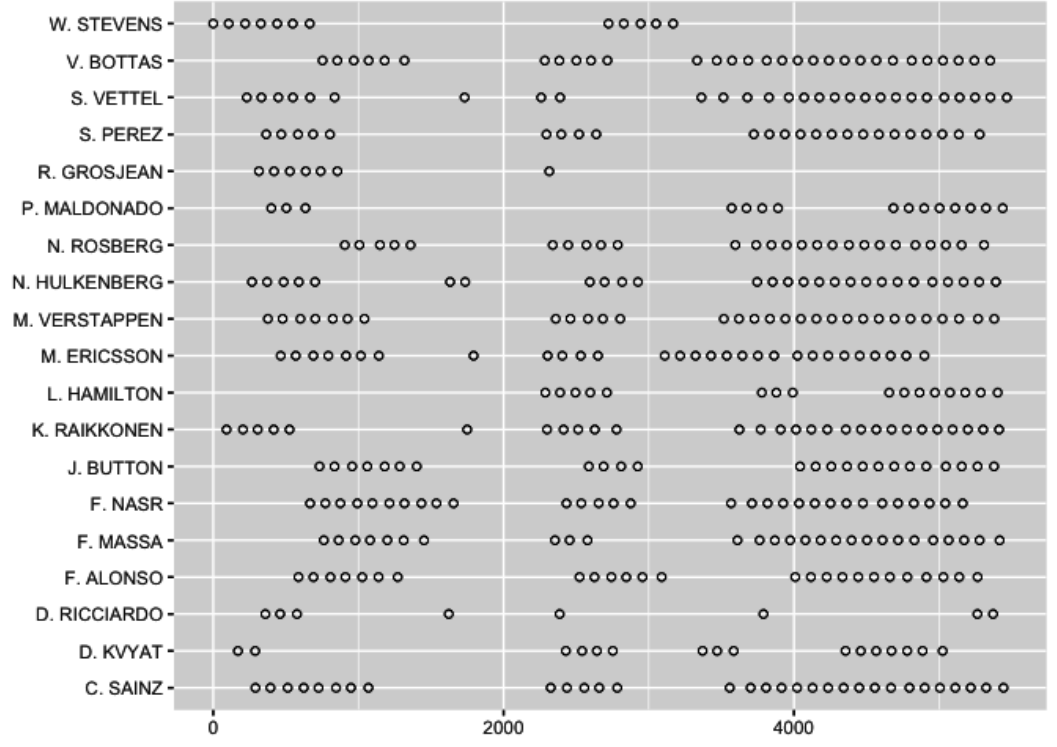
To begin with, we can easily generate an accumulated session time for each driver to record the time into the session they recorded each laptime. This is simply a cumulative sum of laptimes (`stime`) taken over the laptimes, sorted by lap number, recorded by each driver.

```
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
```



Simple session utilisation chart

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.

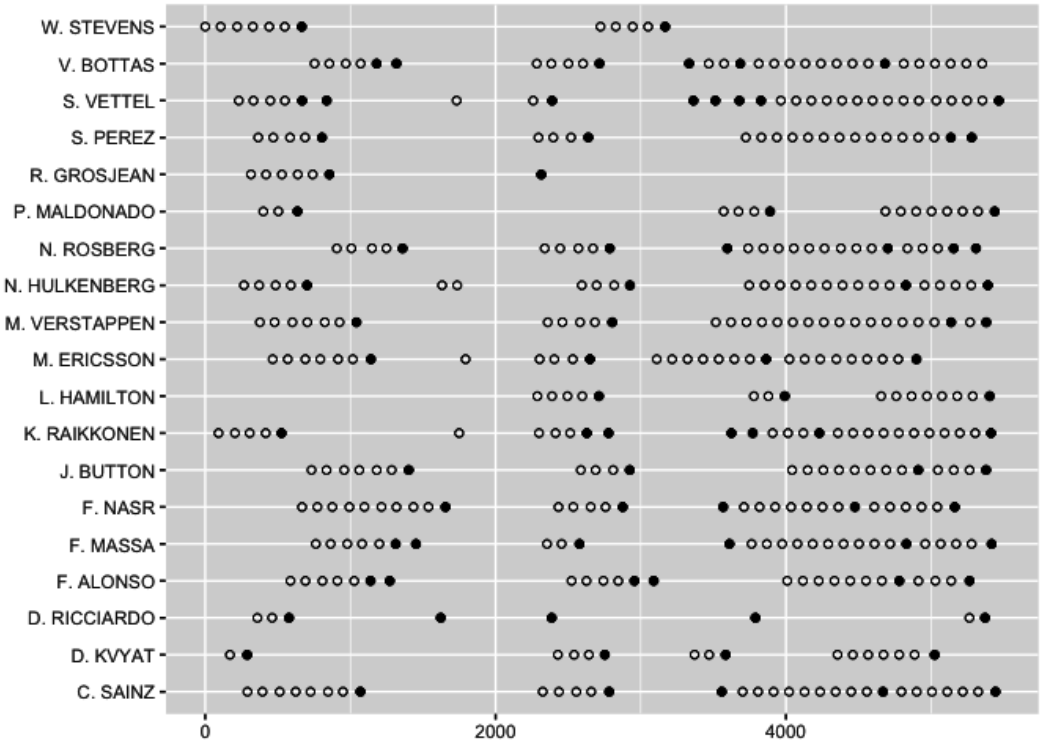
For convenience, convert the `pit` *True/False* flag to a proper R boolean value:

```
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=cuml,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.

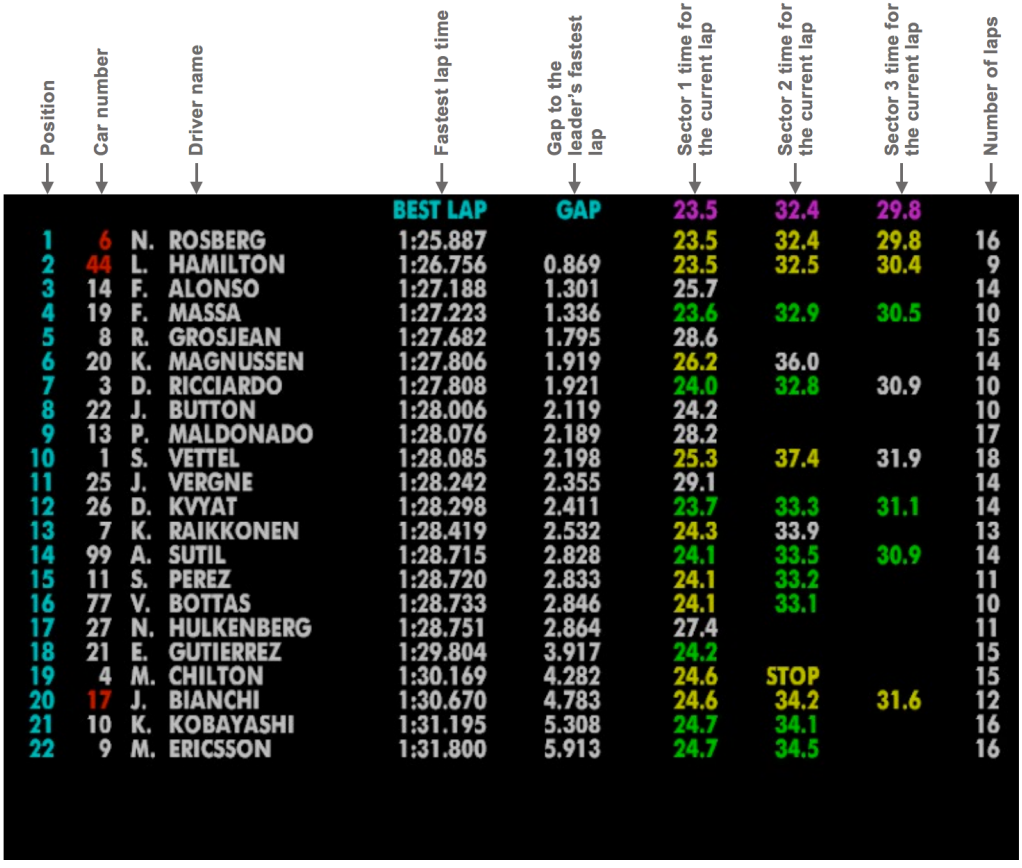
Colour Codes in Motorsport Timing Screens

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



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)])))
```

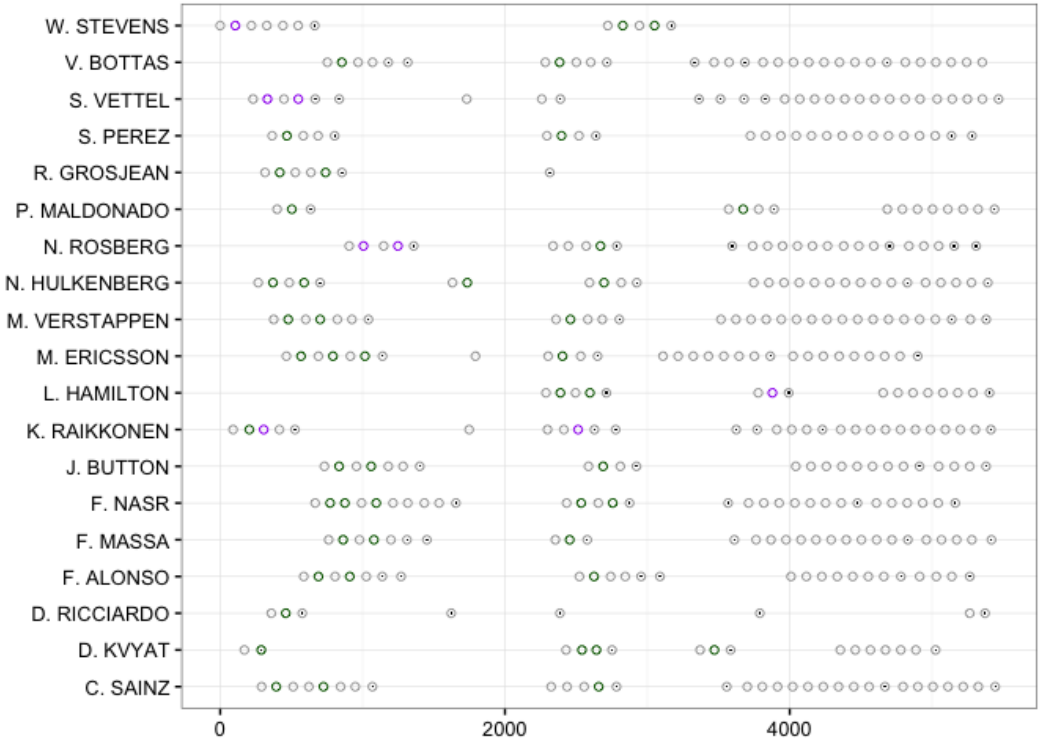
If we now sort by accumulated session time in which the rows are ordered according to the order in which the laps were completed across all drivers, we can identify "purple" times (that is, the best of all the drivers' best laps so far, which is to say, the fastest lap time set so far in the session) 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, lapNumber)
```

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

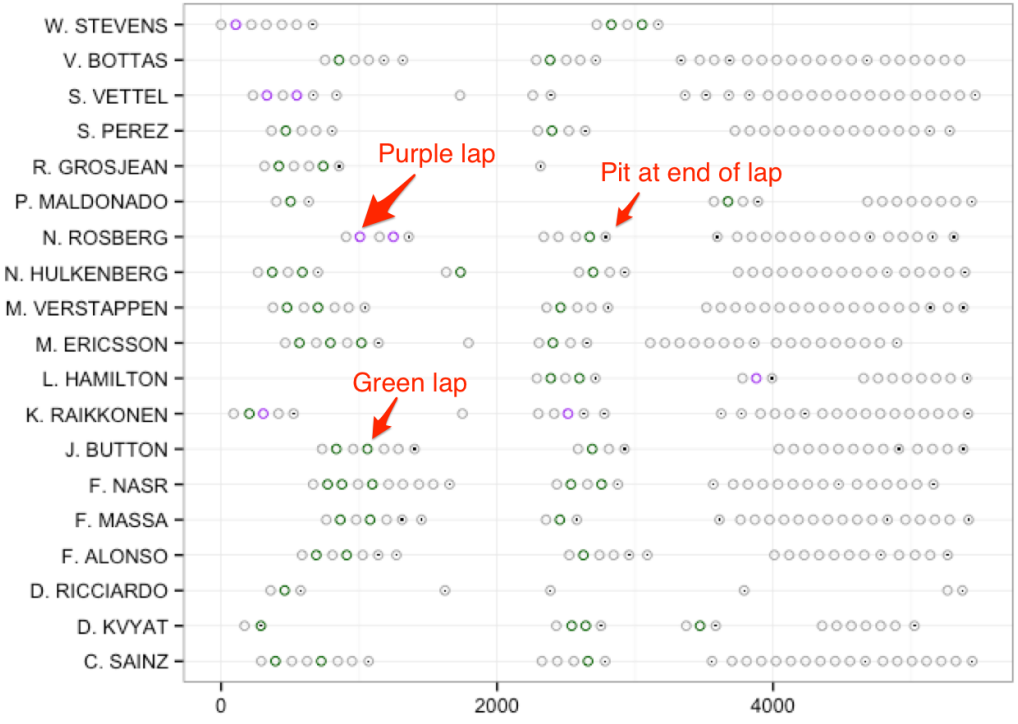
We can now replot the session utilisation chart to show green and purple laptimes, as well as identifying those laps on which the car also came into the pits.

```
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

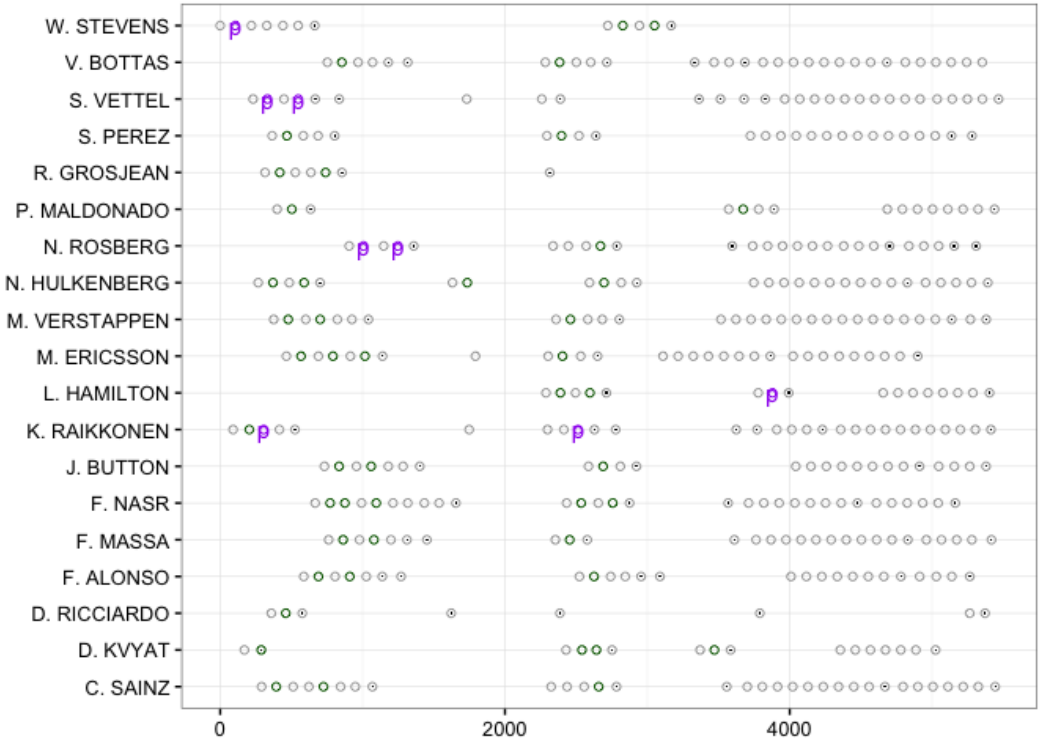
To clarify the chart, three key features of it are the colour coded display of green and purple laps, and identification of the pit stop laps:



Some key features of a session utlisation 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:

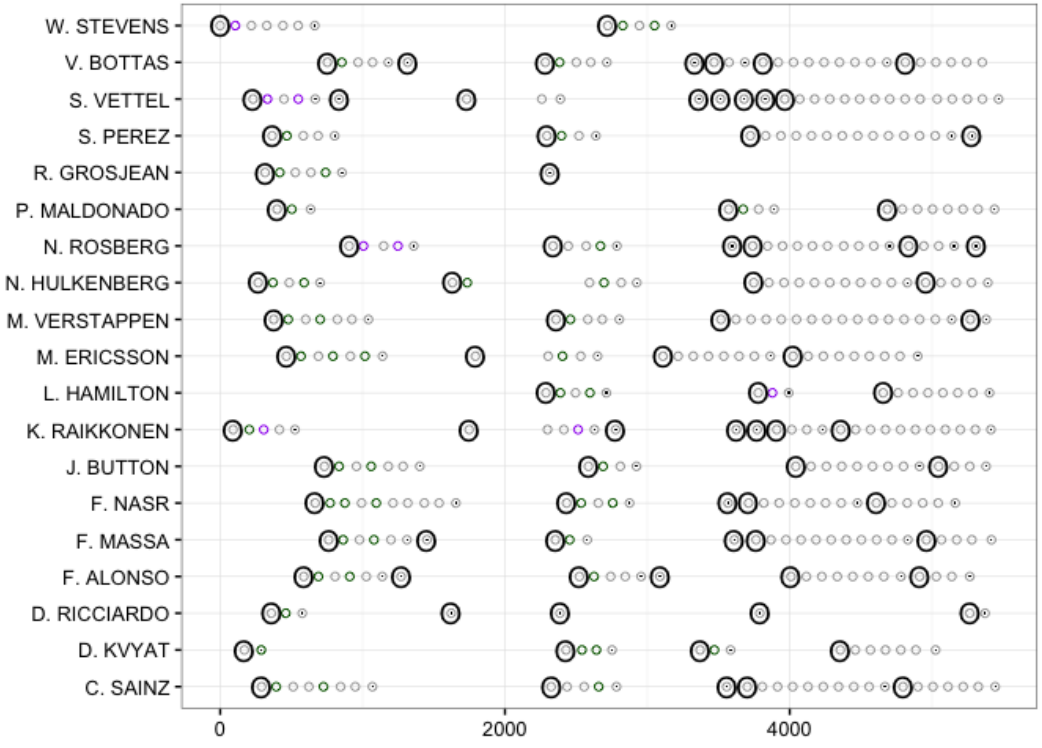
```
ptimes=arrange(ptimes, name, lapNumber)
#Group the rows by driver and stint and number the lap in each stint accordingly
ptimes=ddply(ptimes,
                .(name, stint),
                transform,
                lapInStint=1:length(stint))
```

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='O',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  l
## 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 l
## 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 l
## 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','l')]
    stints=rbind(stints,dft)
}
#Number the stints for each driver
stints=ddply(stints,.(name),transform,stintNumber=1:length(l))
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['nam
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,]['l']))` laps ove\
r `r nrow(stints[stints['name']==name,])` stints, with a longest run of `r max(abs(st\
ints[stints['name']==name,]['l']))` 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.

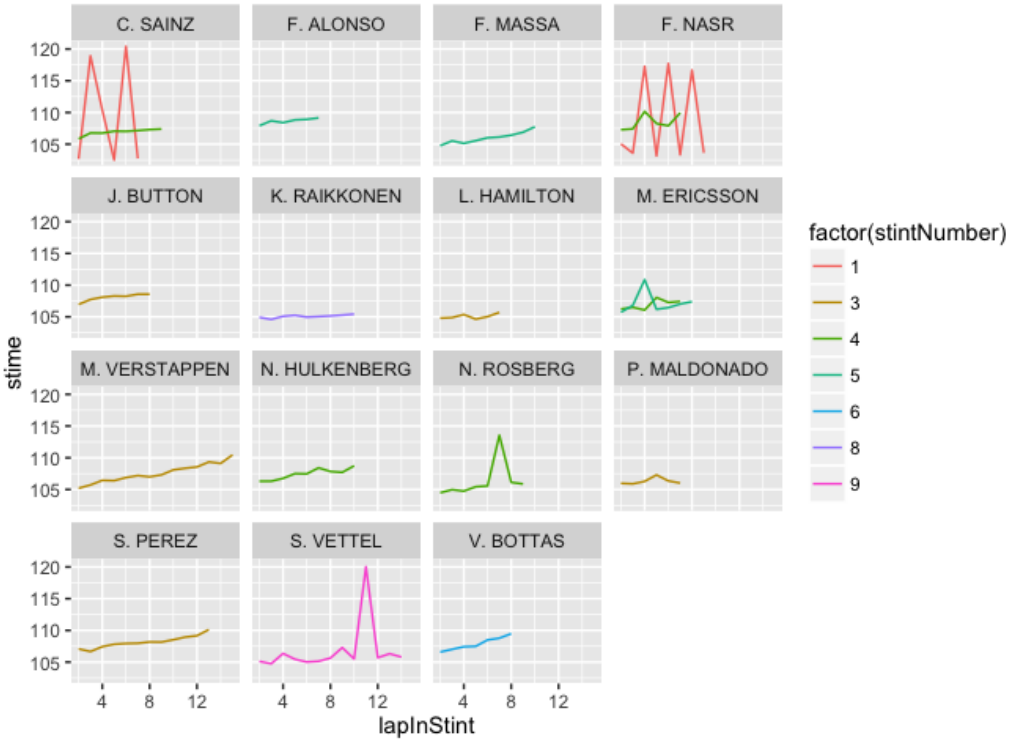
To chart long runs, we need to do a couple of things:

- 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,name,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.

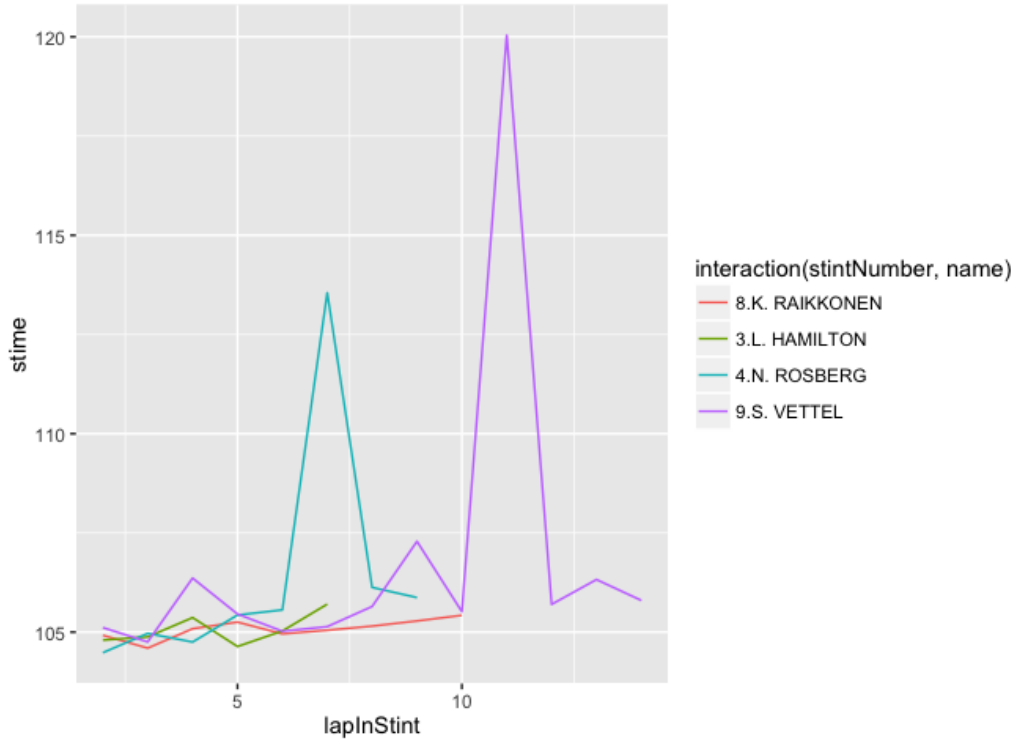
```
#Remove the unrepresentative outlap times
g= ggplot(longruns[!longruns['outlap'] & !longruns['pit'],])
g=g+geom_line(aes(x=lapInStint, y=stime, group=stintNumber,
                  colour=factor(stintNumber)))
g+facet_wrap(~name)
```



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\
s ,])
g+geom_line(aes(x=lapInStint, y=stime, group=interaction(stintNumber,name),
                colour=interaction(stintNumber,name)))
```

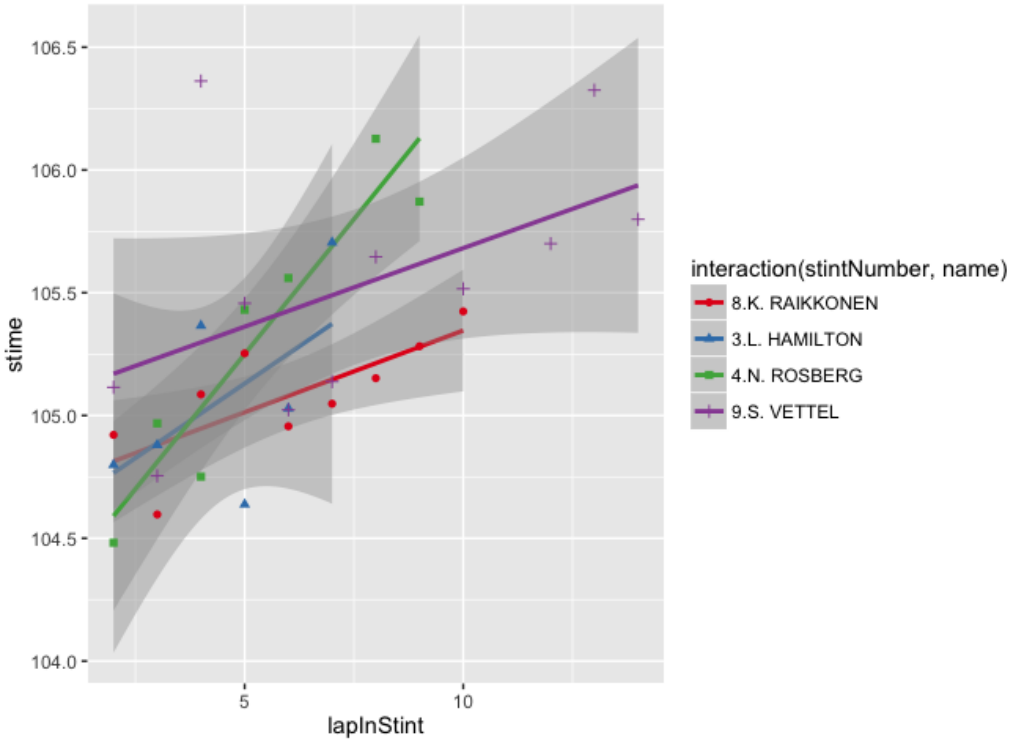


Faceted line charts showing long run laptimes for each driver

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.

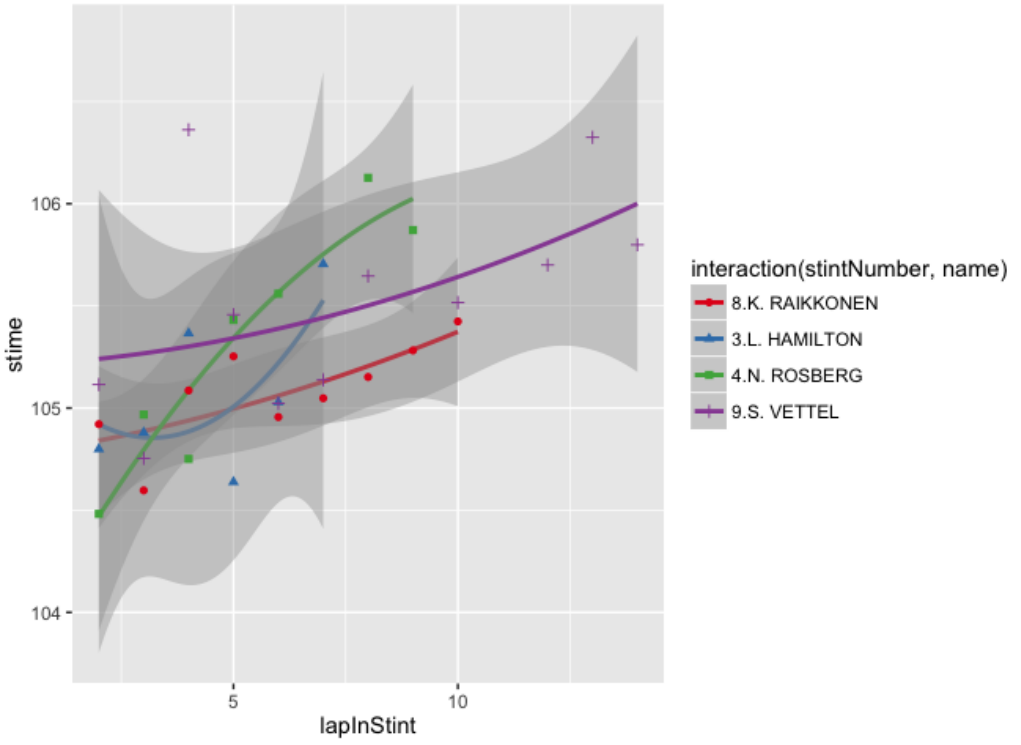
```
drivers=c('L. HAMILTON', 'N. ROSBERG', 'K. RAIKKONEN','S. VETTEL' )
g= ggplot(longruns[!longruns['outlap'] & !longruns['pit'] & longruns$name %in% driver\
s & longruns['stime']<1.07*min(longruns['purple']),],
          aes(x=lapInStint, y=stime,colour=interaction(stintNumber,name)))
g+geom_smooth(method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



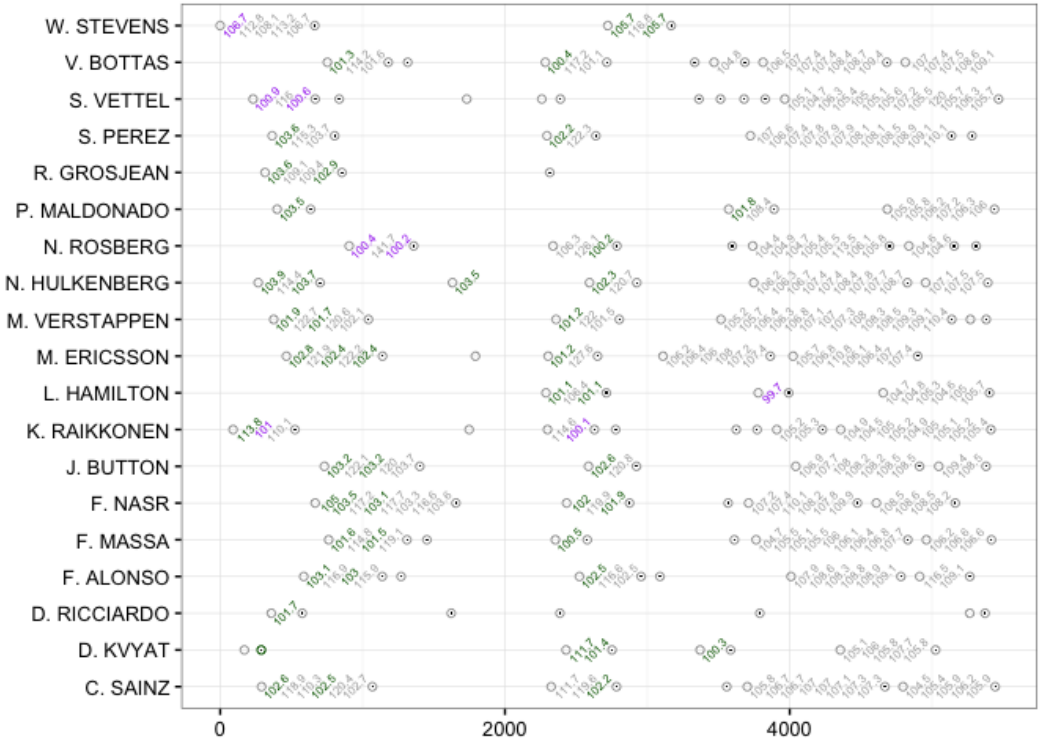
Quadratic model fit for long run laptimes for each driver

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
```



Adding laptimes to the session utilisation chart

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

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.

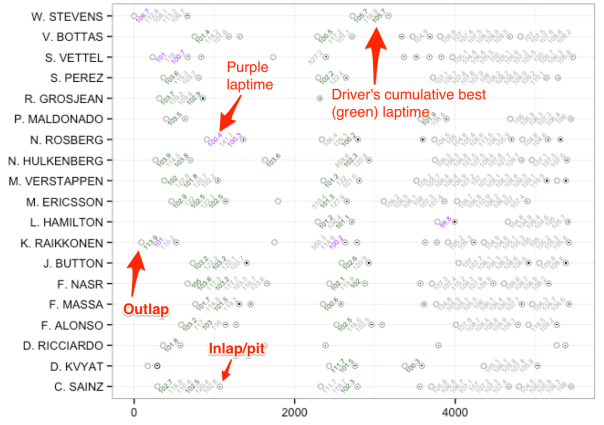
The open circle markers identify outlaps; the circle markers with a spot in the middle represent inlaps, at the end of which the car pitted.

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



Decoding the annotated session utilisation chart

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.

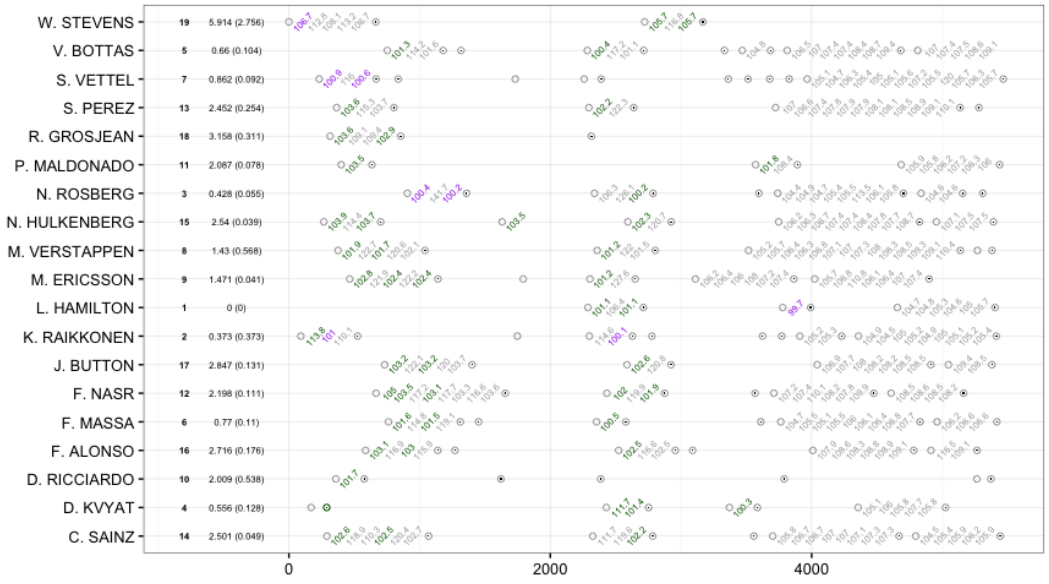
For example, one for of information we can derive is the gap between the best laptime of each driver and the driver with the fastest lap in the session or the gap between consecutively ordered driver best laptimes.

```
#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 driverbest   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
```

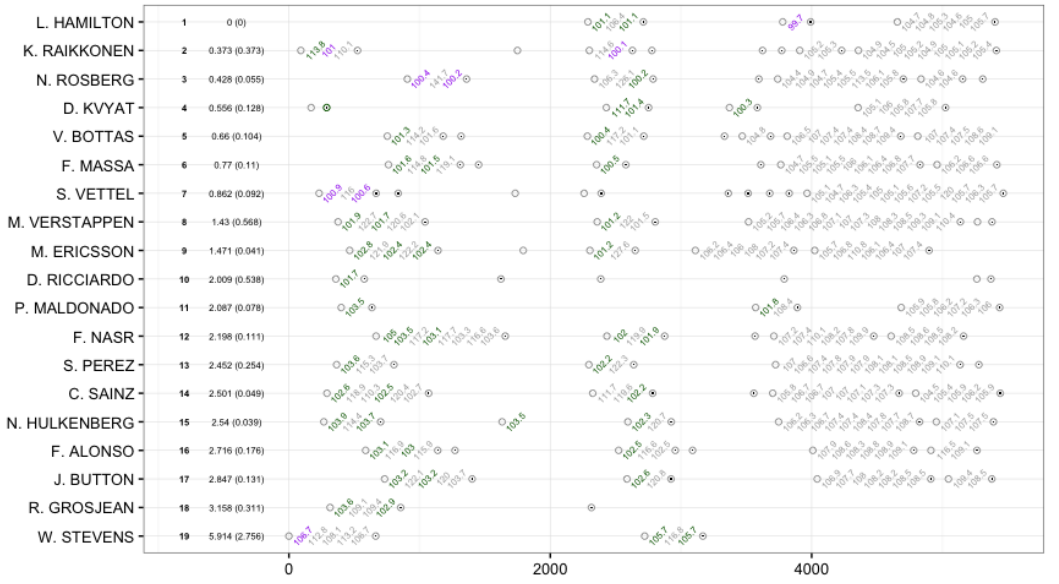


Adding session summary information to the margin of the session utilisation chart

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()
```

Laptime 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 laptime, 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 laptime (i.e. the laptime was *less* than the previous laptime). 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 laptime 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 laptime data 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 laptime 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['colourx']=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['pur\
ple']) & (!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	driverNumber	pos	q2time	q3natTime	q3time	q2natTime	q1laps	team	q1natTime	driverName
115.926	1	3	114.499	1:54.960	114.96	1:54.499	23	Red Bull Racing-Renault	1:55.926	Sebastian Vettel
119.260	10	18	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 laptime
qm=melt(qualiResults,
        id=c('driverNum','driverName'),
        measure=c('q1time','q2time','q3time'),
        variable.name='session',
        value.name='laptime')

#Make sure that the laptime values are treated as numeric quantities
qm$laptime=as.numeric(qm$laptime)
#If a laptime is recorded as 0 seconds, set it to NA
qm[qm == 0] <- NA
#Drop any rows with NA laptime values
qm=qm[with(qm,!is.na(laptime)),]
#This should give us 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	laptime
1	Sebastian Vettel	q1time	115.926
10	Kamui Kobayashi	q1time	119.260
11	Sergio Perez	q1time	118.362

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

```
library(ggplot2)

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

  Text plot showing relative qualifying session times

This chart shows several things 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 faired in their attempt to progress from one qualifying session to the next.

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

  Text plot of qualifying session times with driver connector lines

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

One way we can get round the overlap problem is to introduce an equal amount of separation between the driver names by plotting them against position (that is, rank) values rather than laptime. The upside of 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=laptime))
g = g+ geom_line(aes(group=driverName),colour='darkgrey')

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

  Session times table, with highlights

(Note, if we had included overall position in the *qm* dataframe originally we could have used the *pos* value to highlight the values earlier. Even though positions may change between drivers moving from session to session, the fact that they made it through to a particular session will be given by their overall position in qualifying as a whole.)

The above chart is 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(laptime))`; 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))
```

  Custom line chart showing rank by qualifying session

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.

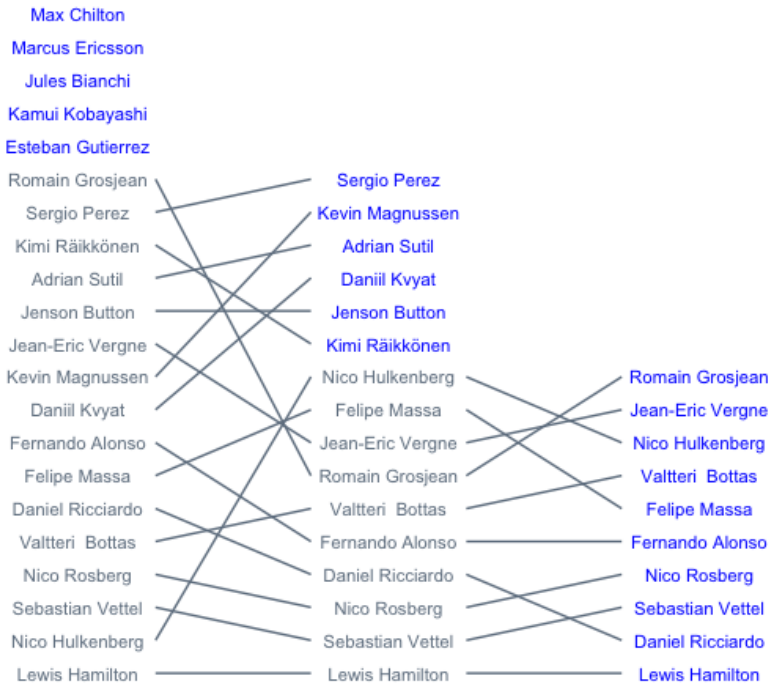
To make the function more general, we can optionally pass in the cut-off limits for each session; in order to access the `cutoff` variable, we need to pass in an appropriate environmental scope to the *ggplot* `aes()` function. Similarly, to account for different label widths, we can pass in the `spacer`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

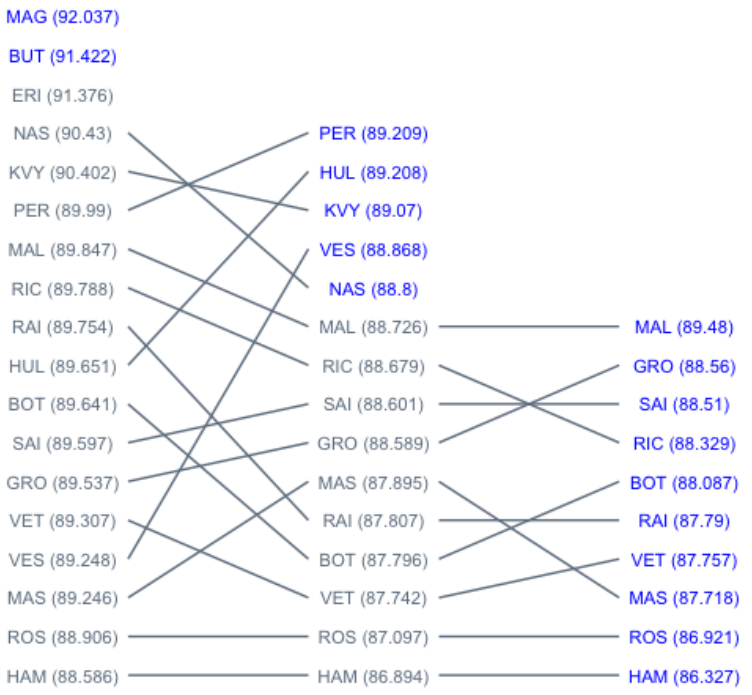
Defining the chart generator as a function makes the production of charts a one-line affair.

```
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='laptime')
qrm$driverName=paste(qrm$driverName," (",qrm$laptime,")",sep='')
qrm=ddply(qrm,'session',mutate,qspos=rank(laptime))



#Make sure that the laptime values are treated as numeric quantities
qrm$laptime=as.numeric(qrm$laptime)
#If a laptime is recorded as 0 seconds, set it to NA
qrm[qrm == 0] <- NA
#Drop any rows with NA laptime values
qrm=qrm[with(qrm,!is.na(laptime)),]



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.

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 colument
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 Kobyashi (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:

$$t_{P,N,GAP} = DIFF_{2,1,N} + D_{3,2,N} + .. + DIFF_{P,P-1,N}$$

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 laptime 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

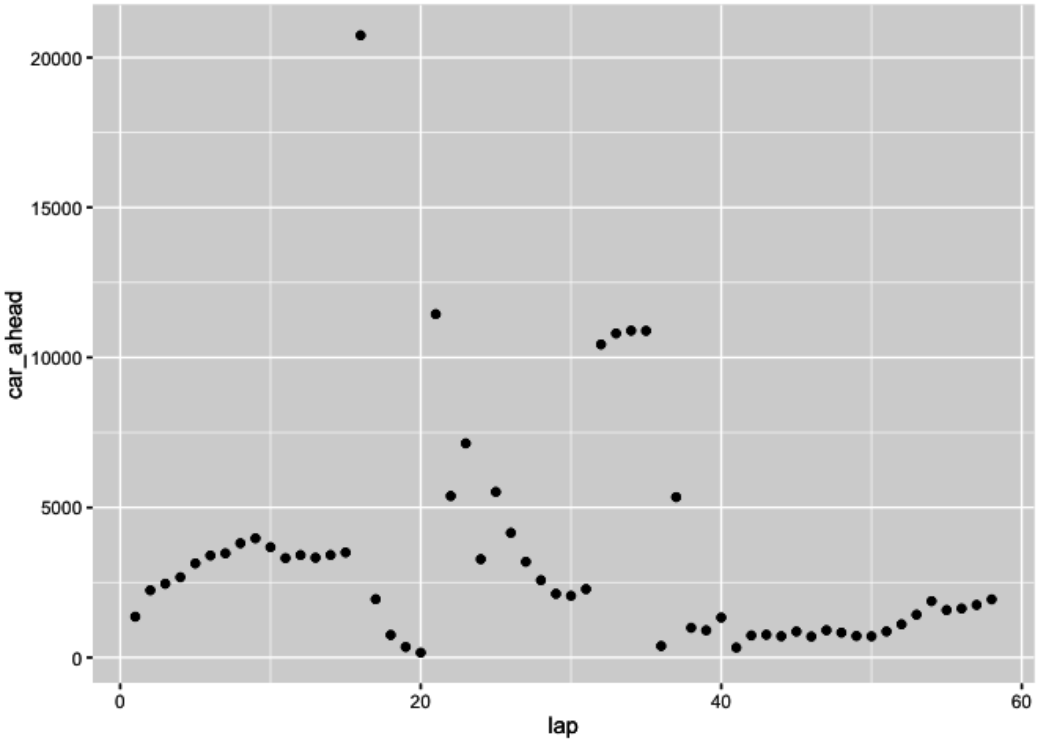
To calculate the time difference to the car ahead on track (`car_ahead`) and the car behind on track (`car_behind`), we can simply calculate the differences between accumulated laptimes for appropriately ordered rows.

```
#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 acculamated 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 .

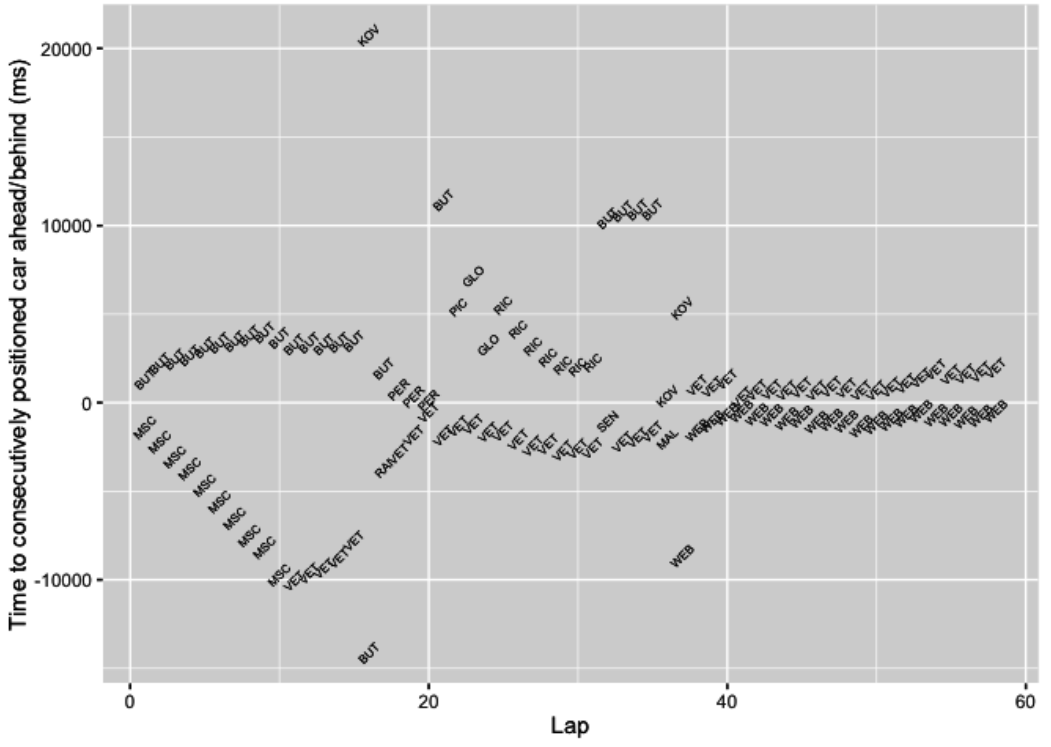
```
library(ggplot2)
#Display the car one position ahead of a named driver
g=ggplot(lapTimes[lapTimes['code']=='HAM',])
g+geom_point(aes(x=lap,y=car_ahead))
```



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.

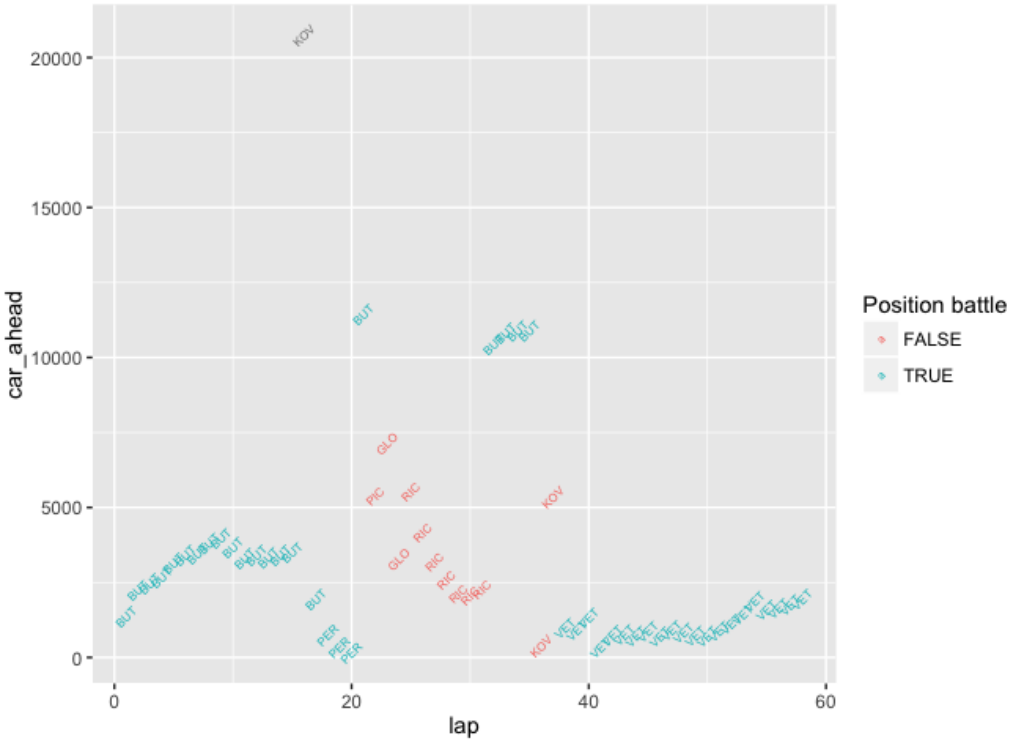
One way to approach this is to generate new columns that identify the driver immediately ahead or behind each car in terms of race position, rather than track position. This new information will allow us to test whether the car ahead or behind on track is in a battle for position with the selected driver.

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 accumaulated 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')
```



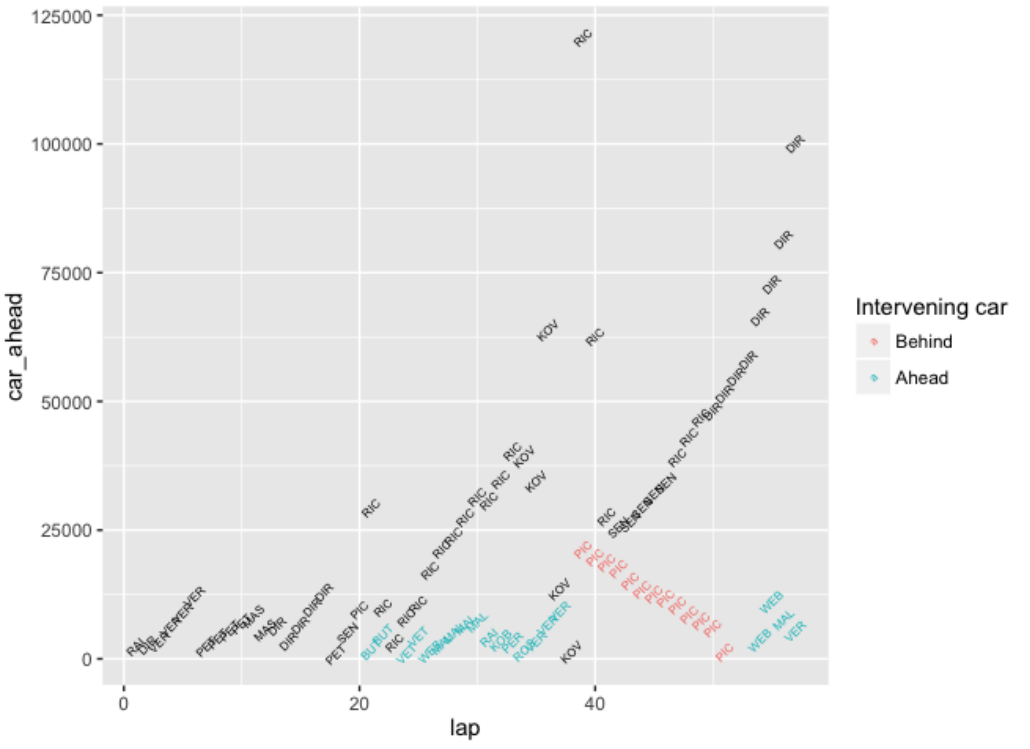
plot of chunk battlesketchHAM

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')
```



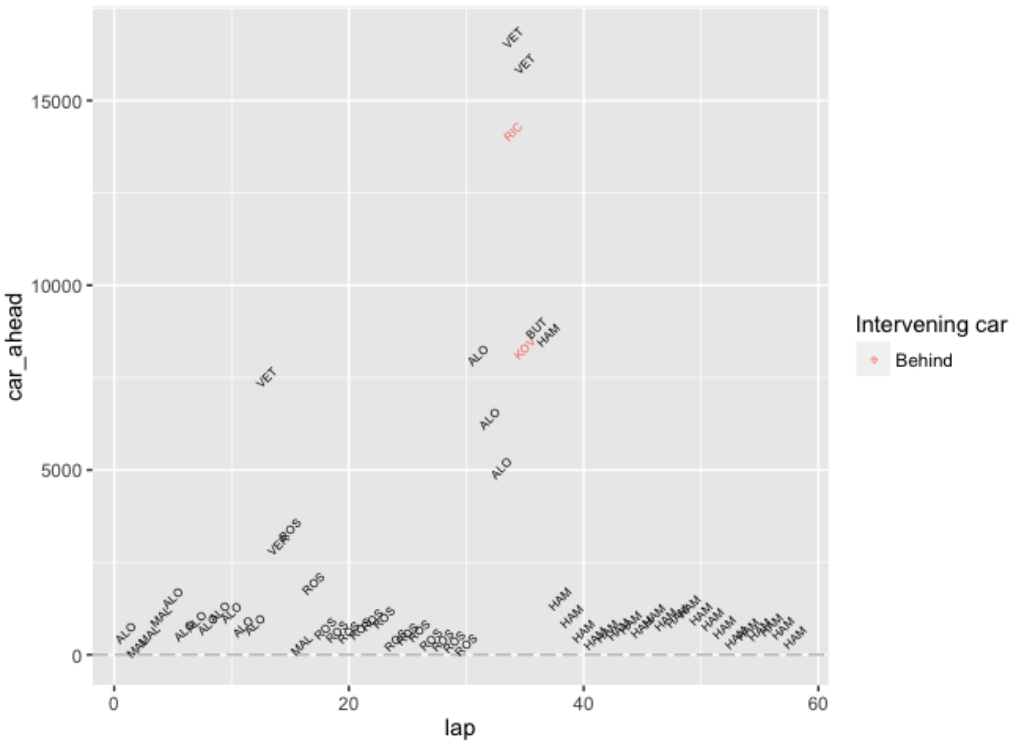
A first attempt at a battle map, showing cars in positions immediately ahead/behind on track, as well as in race position terms

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.

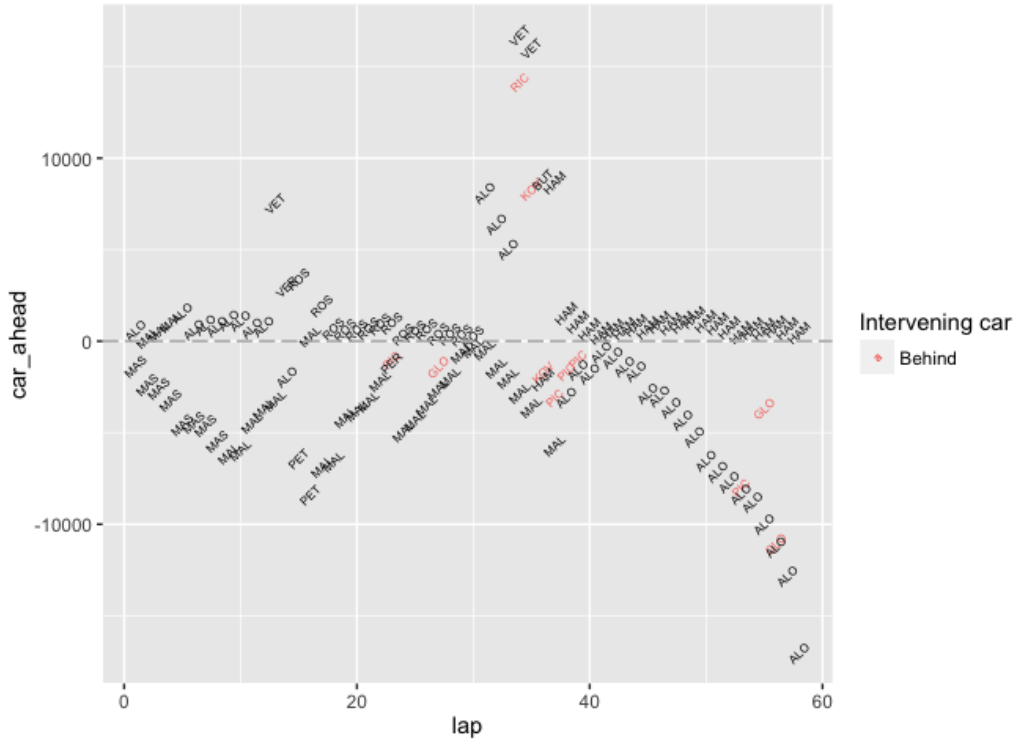
```
dirattr=function(attr,dir='ahead') paste(attr,dir,sep='')

#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='lap',
                  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='lap',
                  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')
```



plot of chunk battleCoreWEB

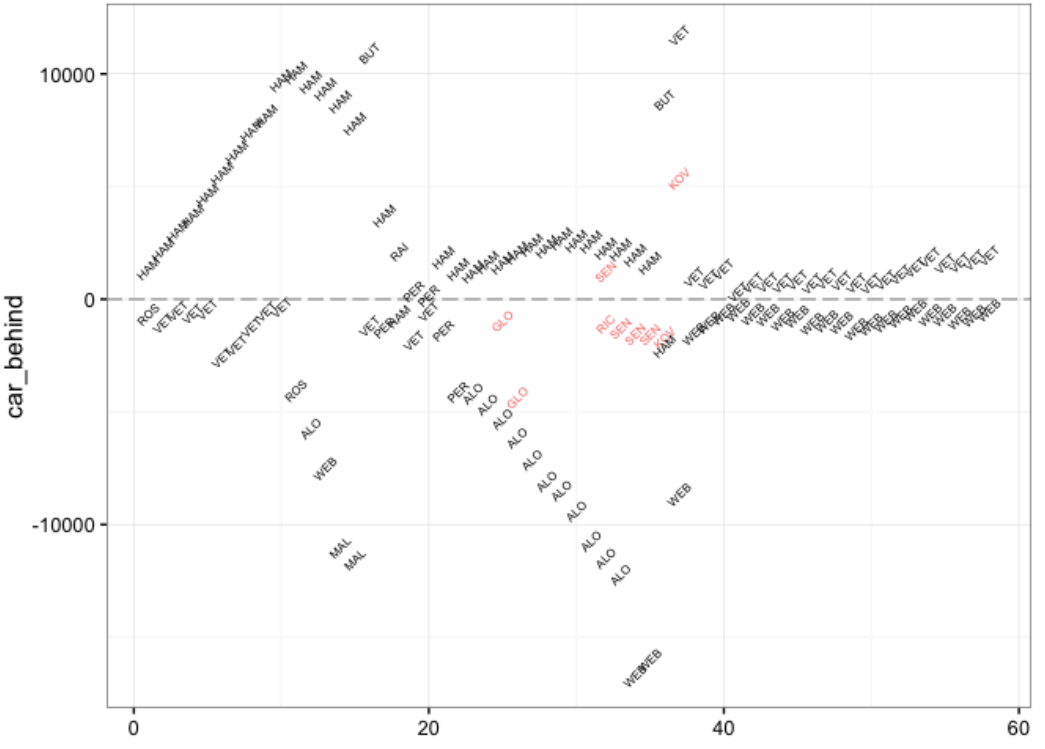
Here we see how at the start of the race Mark Webber kept pace with Alonso, albeit around about a second behind, at the same time as he drew away from Massa. In the last third of the race, he was closely battling with Hamilton whilst drawing away from Alonso.

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.

```
battlemap_df_position=function(position){
  lapTimes[lapTimes['position']==position,]
}

battleForThird=battlemap_df_position(3)

g=battlemap_core_chart(battleForThird,ggplot(),dir='behind')+xlab(NULL)+theme_bw()
g=battlemap_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.

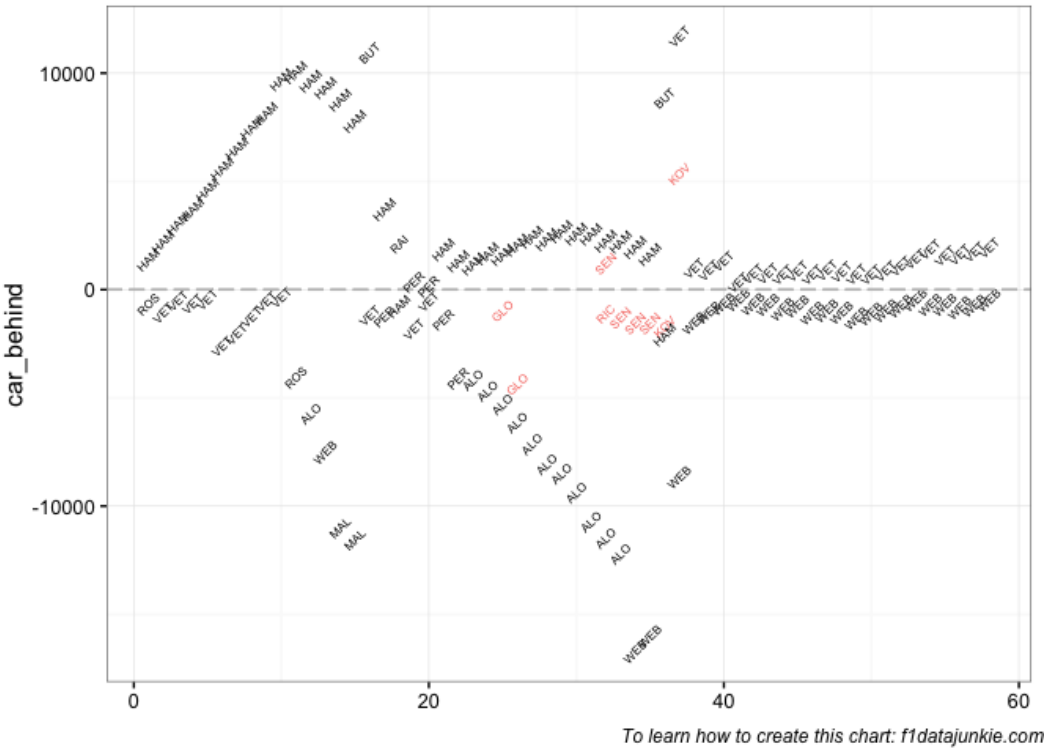
One thing this chart does not show is which driver was in third position on each lap. We might naturally think to add a layer on to the chart that displays the driver in the position we are charting around along the x-axis (that is, at y=0) but this is likely to lead to a very cluttered chart. It may 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 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']
```

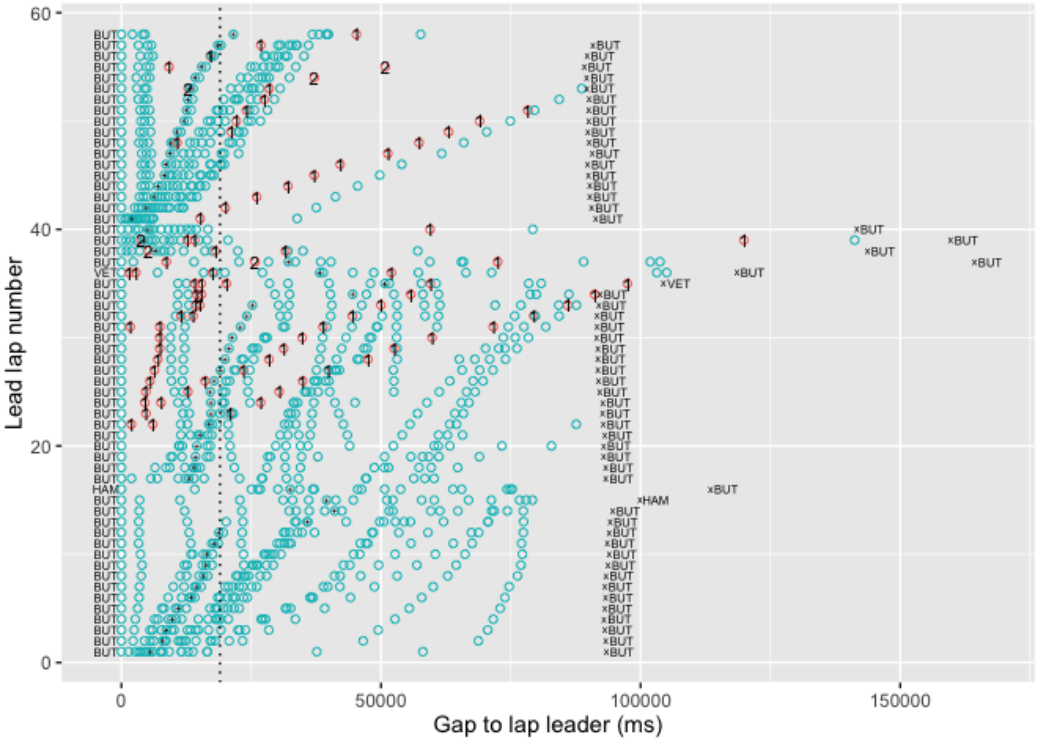
We can also identify the time distance between the leader of the current lap and the leader of the next lap - this gives us an estimate of the "time width" of the lap.

```
#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?
ll=data.frame(t=diff(lapTimes[lapTimes['position']==1,'acctime']))
#Generate a de facto lap count
ll['n']=1:nrow(ll)
#Grab the code of the lap leader on the next lap
ll['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=ll,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 â€˜next-lap-leaderâ€™ 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 processed 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.

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 at 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. 1-7[34]), 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 at 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.

[34] 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))
```

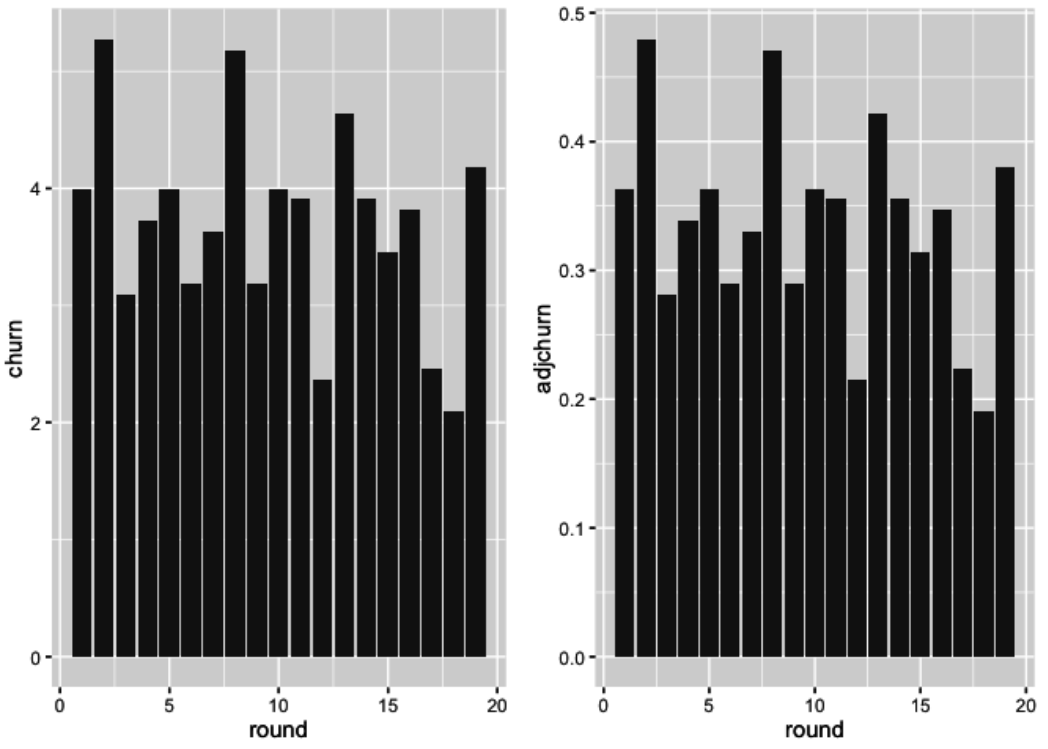
Using these functions, it's straightforward enough to calculate the churn for each race based on the absolute difference (*delta*) between the grid position and final position of each driver.

```
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)
```

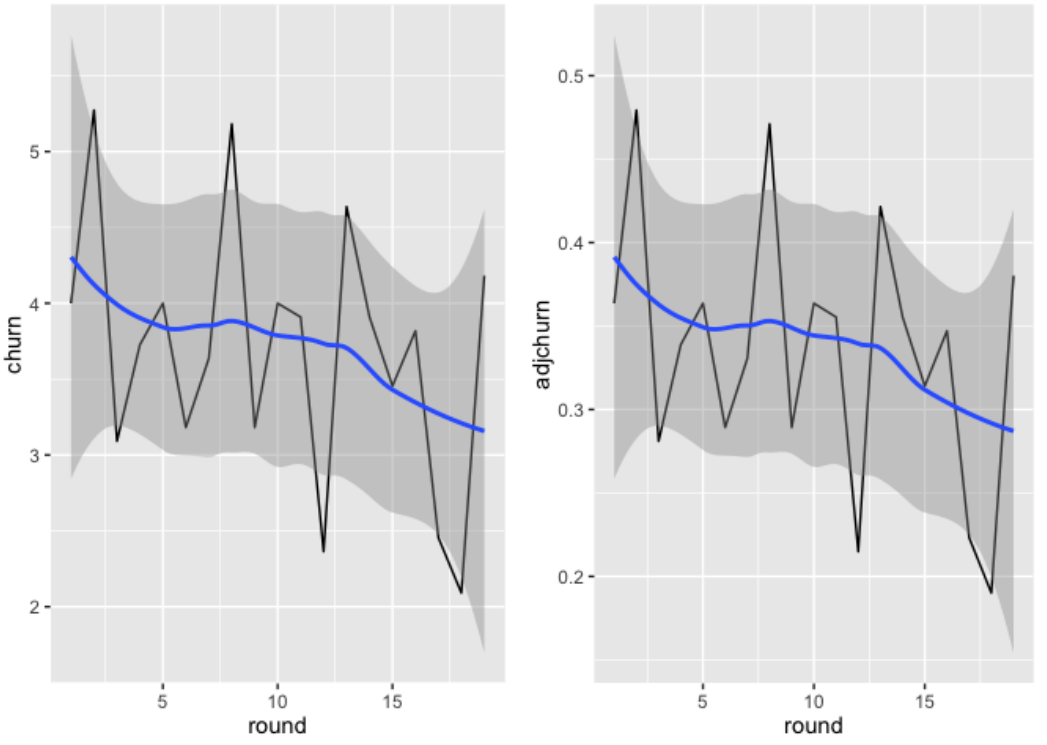


Bar charts showing churn and adjusted churn values for races in the 2013 season

So what sorts of thing 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.

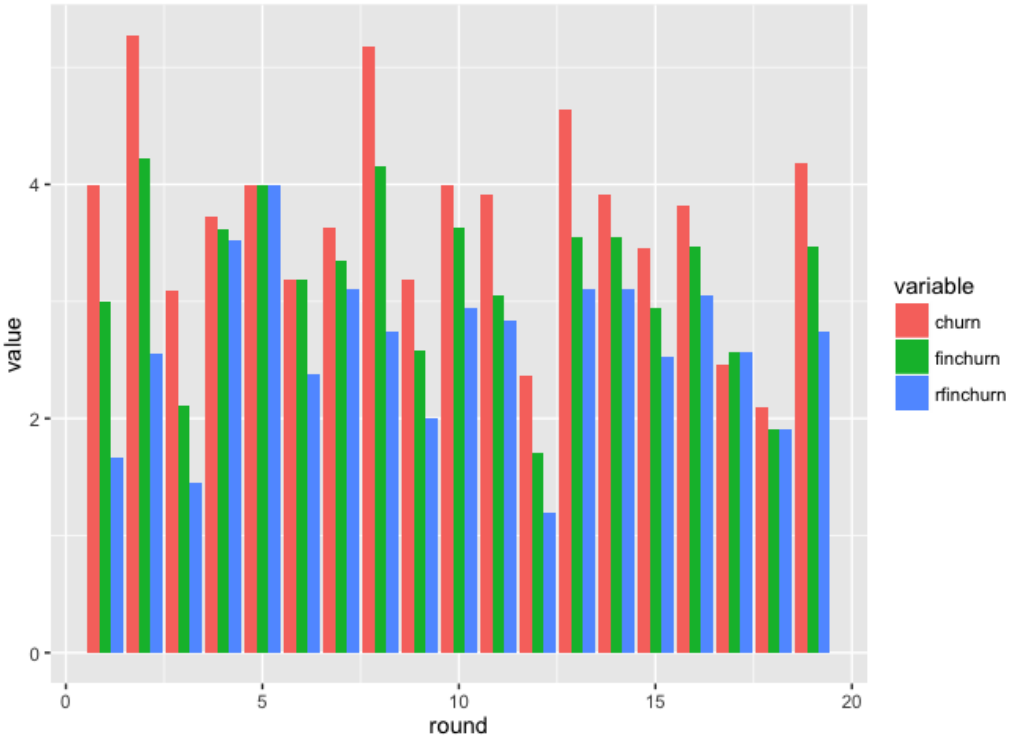
```
#Start by filtering the dataframe to contain only classified drivers
#Then calculate the ranked grid position equivalent for the classified drivers
classresults=ddply(results[!(is.na(results['position'])),],
                   .(round), mutate,
                   delta=abs(grid-positionOrder),
                   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','rfinchu\
rn'))
adjChurnData=melt(churn3.df, id.vars='round', measure.vars=c('adjchurn','finadjchurn'\
,'rfinadjchurn'))
```

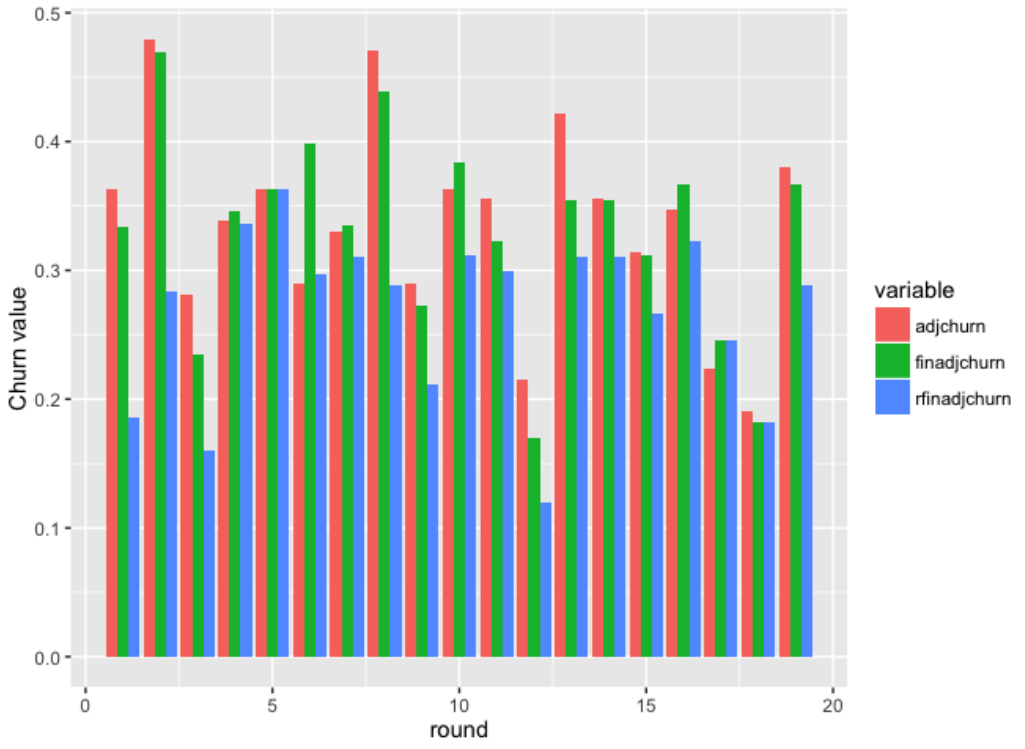
By inspection of the following charts, we note that the rank order of the churn and adjusted churn values for the races differs for the different population and grid/gridrank value combinations.

```
ggplot(churnData) + geom_bar(aes(x=round, y=value, fill=variable),
                             stat='identity', position='dodge')
```



Various churn values calculated for each race of the 2013 season.

```
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* > *finadchurn* > *rfinadchurn*, whereas in rounds 4, 5 and 19 all values are the same or more or less the same, and in rounds 10 and 16 *finadchurn* > *adjchurn* > *rfinadchurn*.

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.

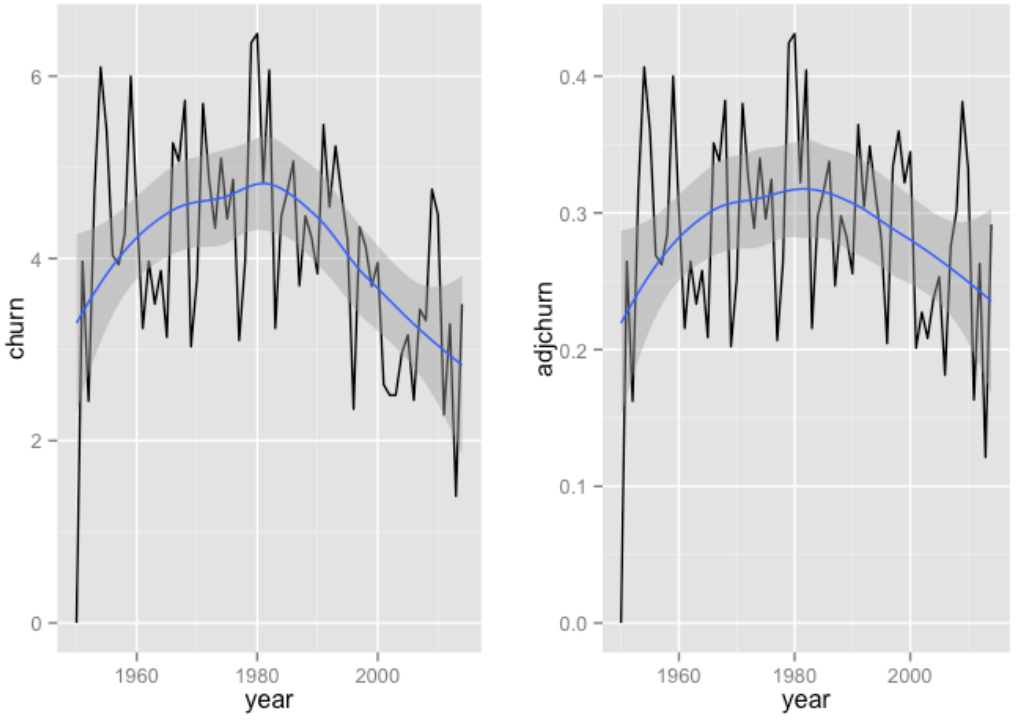
In calculating the churn in standings from one championship year to the next, we need to draw on the difference between a driver's standing in one year and their standing in the previous year.

```
seasons=ddply(seasons,.(driverId),transform,delta=c(0,abs(diff(pos))))
```

We can than apply the churn function to these differences, summarising the data for each year of interest.

```
churner = function(df) ddply(df, .(year), summarise,
                             churn = churn(delta),
                             adjchurn = adjchurn(delta)
                             )

season.churn.df = churner(seasons)
```

```
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)
```

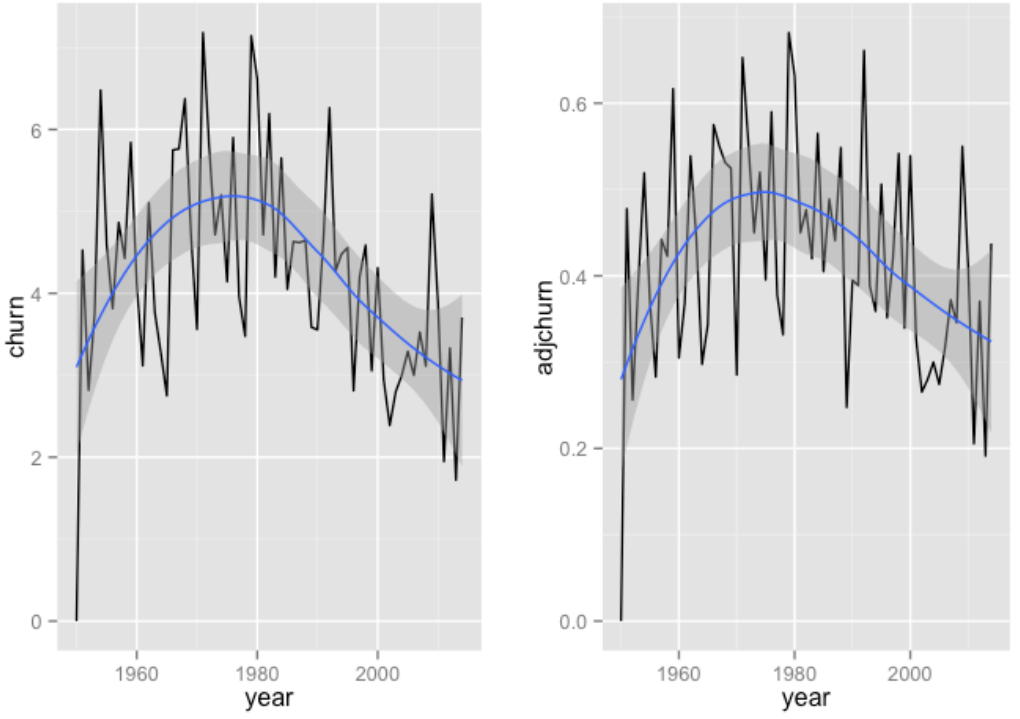


Churn in the Drivers' Championship standings, 1950-2014

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.

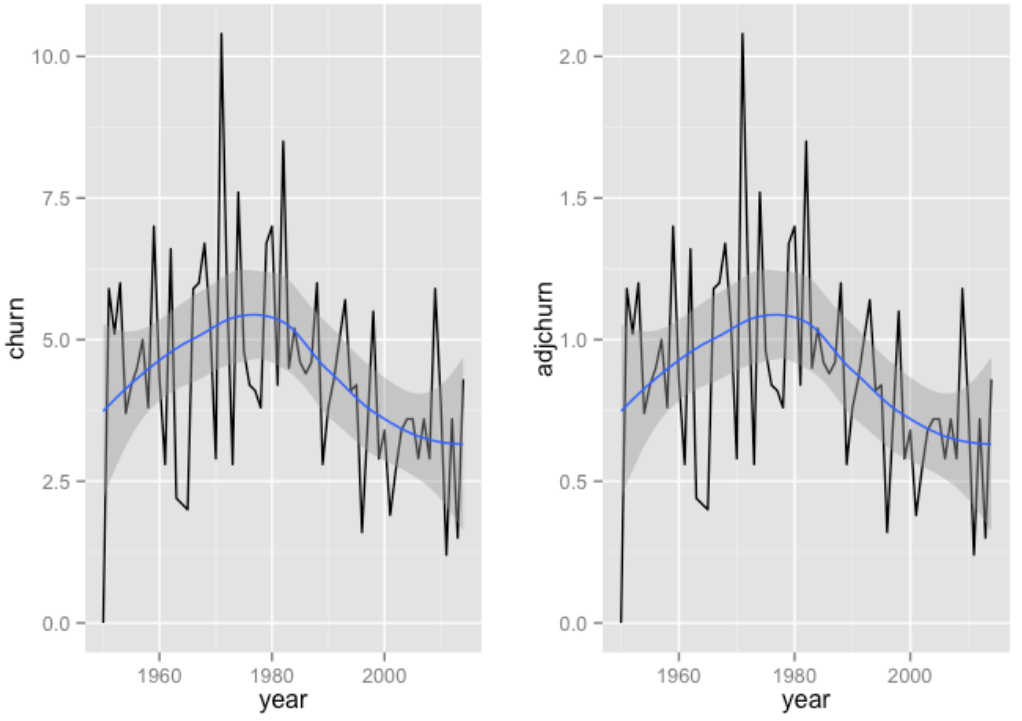
```
season.churn.points.df = churner( seasons[seasons['points']>0,] )

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

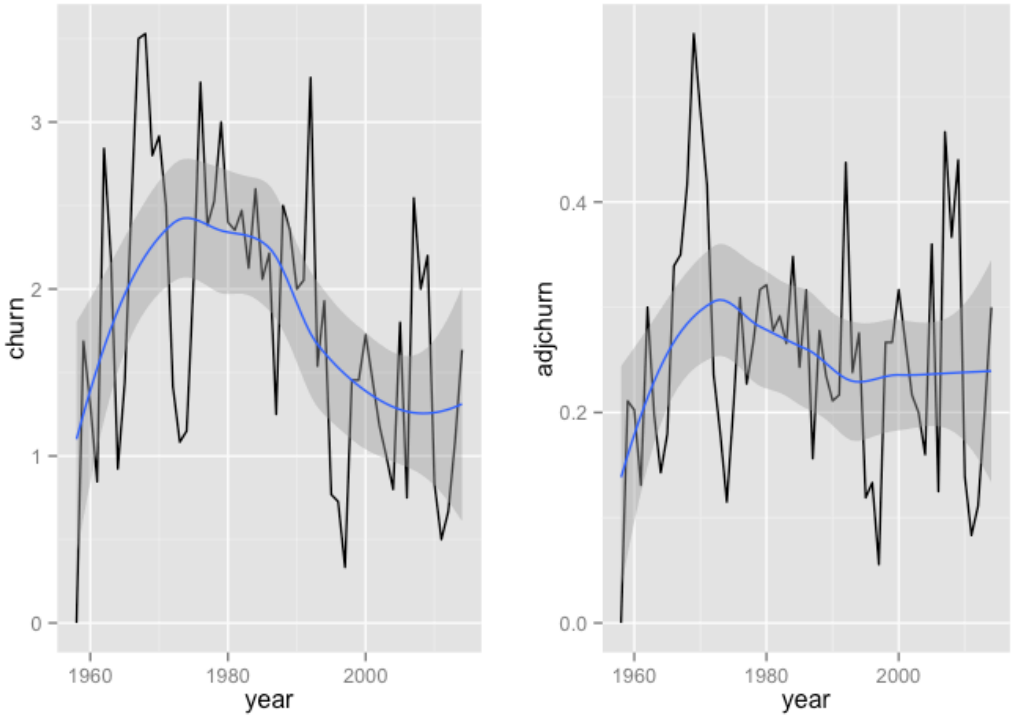
The method for calculating the churn across seasons based on the Constructors' Championship closely follows that used for the Drivers' Championship.

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 the 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 regulations[35]). Do periods of significant change appear to have an influence on competitiveness as measured by churn in championship standings?

[35] http://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 at 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.

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 won for each possible result in the final race, we need to find out the points difference between eacg 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 finishes 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 finishes 2nd and Hamilton finishes 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, Hmailton 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
```

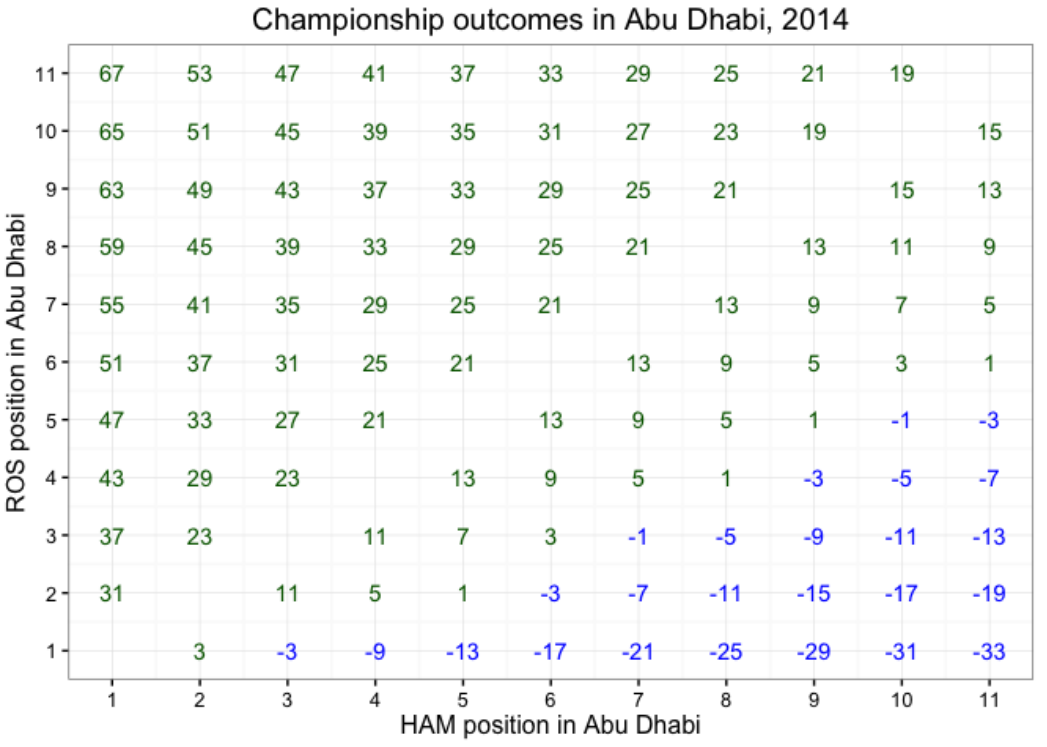
Reading this, we see if Rosberg finishes in first, cells *(N,1)*, if Hamilton finishes second, he is *+3* up on Rosberg after the race and takes the championship, but if he finishes third he'll be three points down (*-3*) and will lose the championship.

If Hamilton finishes seventh and Rosberg finshes third, cell *(7,3)* shows a result of *-1* - Rosberg loses by a single point.

Visualising the Outcome

We can provide a more eyecatching view over the data by colouring each cell according to which driver takes the championship win.

```
library(reshape)
library(ggplot2)

results=melt(pdiff.final+Vectorize(17))

g=ggplot(results)+geom_text(aes(X1,X2,label=value,col=factor(sign(as.integer(as.chara\
cter(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")+scale_color_manual(values \
= c('blue','darkgreen'))
g=g+theme_bw() +theme(legend.position="none")
g+scale_x_continuous(breaks=seq(1, 11, 1))+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.

In a forthcoming update to this chapter, we'll see how we can model a situation with two drivers going in to the penultimate race of the season with them both still in contention for the Drivers' Championship.

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 is 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.

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 in to);
- 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 one driver led 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)

faceoff=dcast(qualiResults[,c('code','position','round')],round~code,value.var='posit\
ion')
```

```
##   round HAM ROS
## 1     1   1   3
## 2     2   1   3
## 3     3   2   1
```

We can directly count how many times each driver beats the other based on summing the results of an inequality test applied across the driver columns, 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         12
## 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 is rather laboured, and in part reflects the way the data 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:

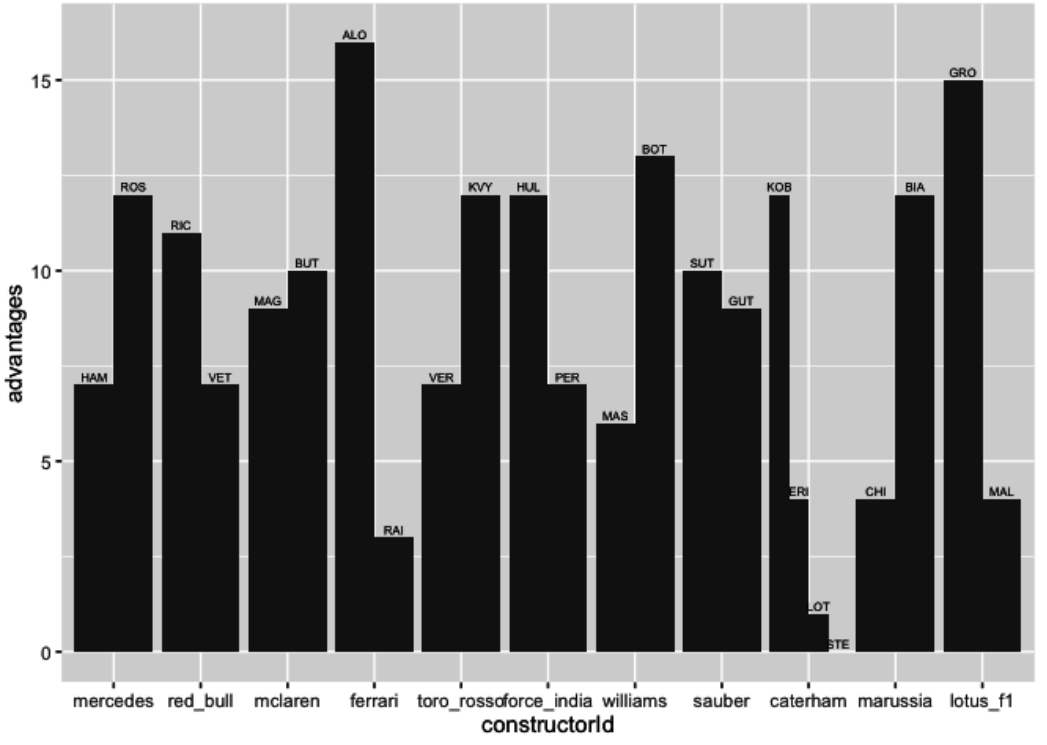
```
library(DBI)
con = dbConnect(RSQLite::SQLite(), ":memory:")
tmp=dbWriteTable(con, "qualiResults2", qualiResults2, row.names = FALSE)

dbGetQuery(con,
           'SELECT q1.constructorId,q1.code,COUNT(*)
            FROM qualiresults2 q1 JOIN qualiresults2 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       12
## 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 - surely there must be a better way of representing this information?

Race Performance

Before asking very particular questions about how drives compared in terms of their race performances during a season, let's have a look at how the drivers compare. 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')
```

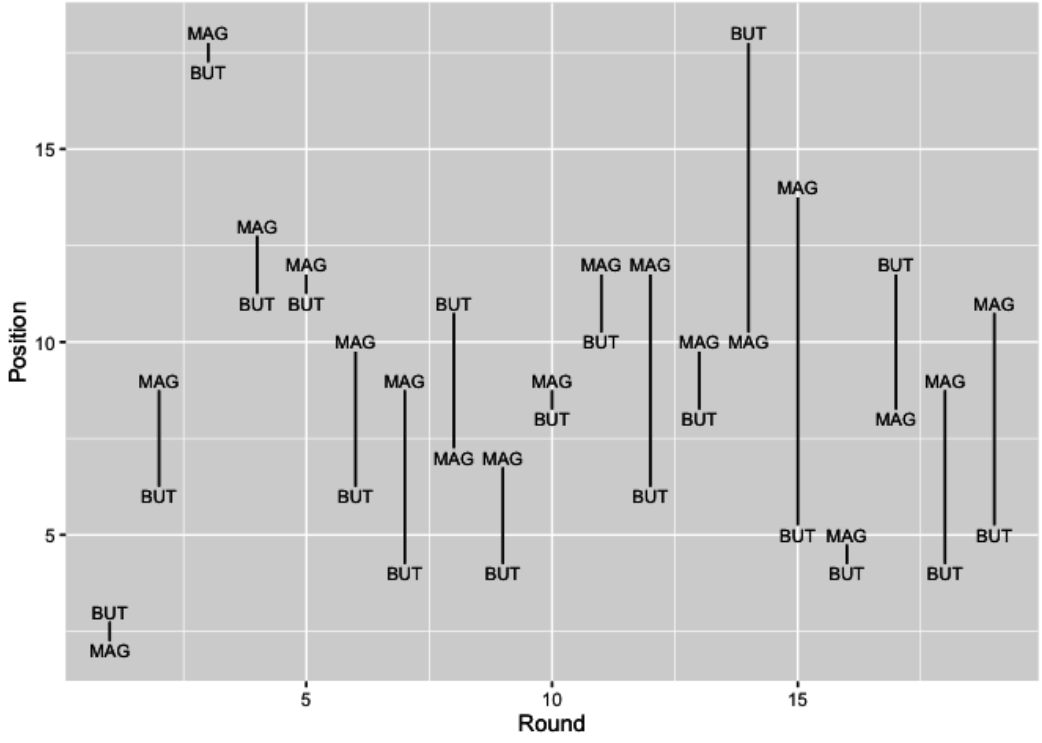
driverId	code	constructorId	grid	laps	position	positionText	points	status	season	round
button	BUT	mclaren	10	57	3	3	15	Finished	2014	1
button	BUT	mclaren	10	56	6	6	8	Finished	2014	2
button	BUT	mclaren	6	55	17	17	0	Clutch	2014	3

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.

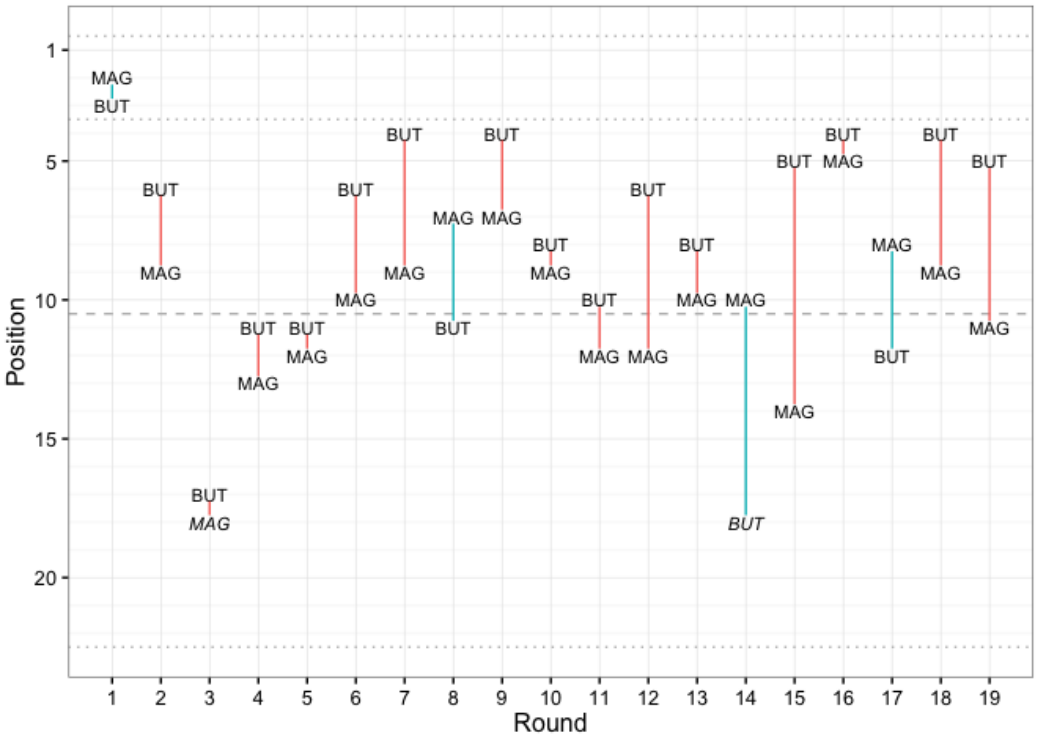
```
#Create the base plot
g=ggplot(raceoff,aes(x=round))
#Add text labels showing the position of each driver in each round
g=g+geom_text(aes(y=position.x,label=code.x),size=3)
g=g+geom_text(aes(y=position.y,label=code.y),size=3)
#Add a line between the two labels for each round to connect them together
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, by round, in the 2014 Drives 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 the position went unclassified. Finally, adding a clear theme and inverting the y-axis scale so the highter placed driver is at the top of the chart produces a chart like the following.

```
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')
  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
  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.

The chart clearly demonstrates how closely 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.

If we 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.

Summary

In this chapter we have started to explore various ways of comparing the performances of drivers within a team. In future updates, we will explore how to use these charts, with asscoiated commentary, as part of a season report.

Points Performance Charts

Knowing the likely payoff for different grid positions based on 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 *points performance charts*, a novel graphical technique for comparing not just driver performances, but also the extent to which a team maximises its points haul when the lead driver attains a particular points scoring position.

This approach is very much a work a 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. So let's start there...

Maximising Team Points Hauls

Since the 2010 season, the top ten classified positions have scored championship points (25 for first, 18 for second, 15 for third, then 12, 10, 8, 6, 4, 2 and finally 1 point for 10th) (FIA F1 Regulations[36]). A tweak to the 2014 regulations makes double points now available in the final race of the season.

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's the last race of the season in 2014 and perhaps onwards), 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` but distinct `driverId` and recording the results of the higher-

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

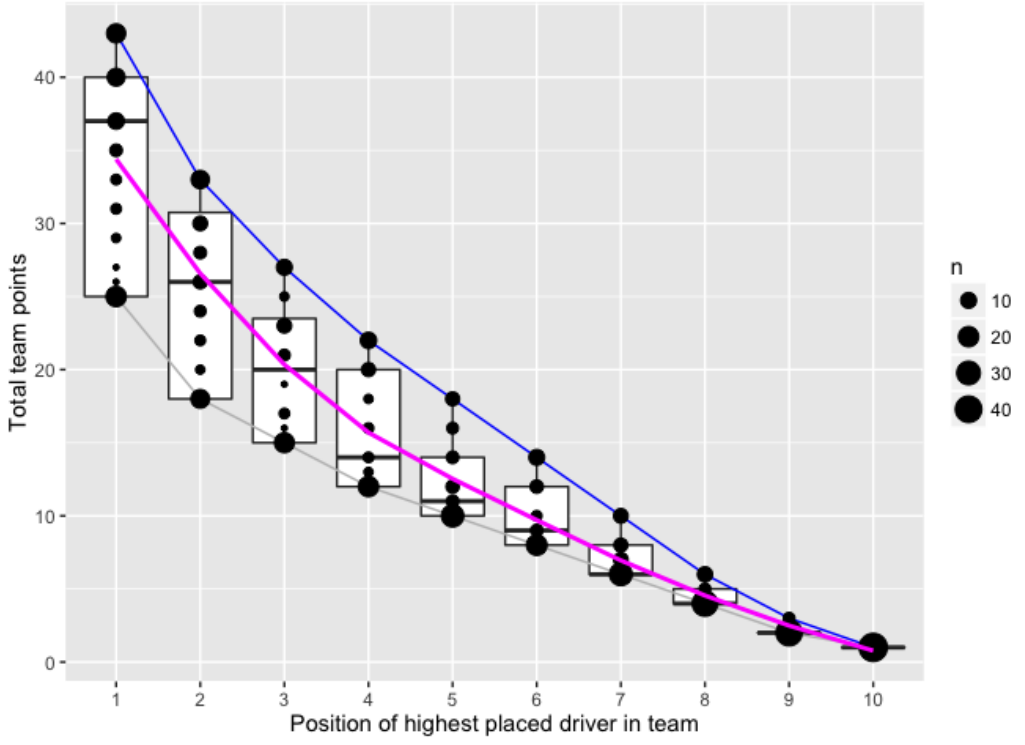
placed driver in the team (*r1*) and the lower placed driver (*r2*). We'll also grab the total points haul for each team in each race.

```
library(DBI)
library(ggplot2)
ergastdb =dbConnect(RSQLite::SQLite(), './ergastdb13.sqlite')

#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

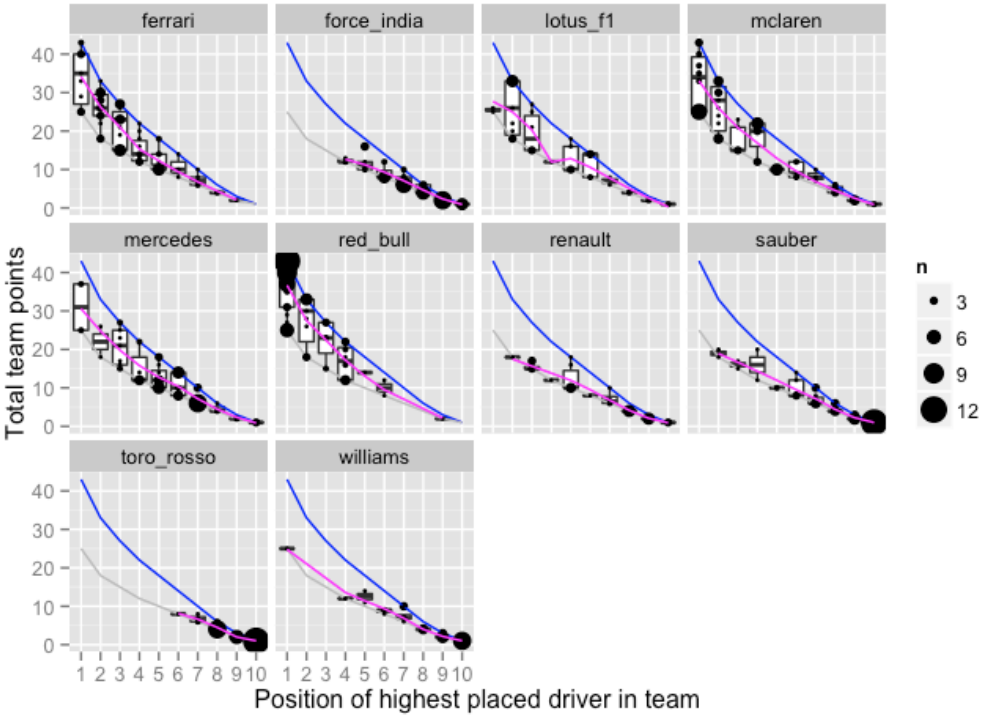
the number of times a particular team points haul was recorded given the position of the highest ranked team member. In this example, data comes from all teams for the seasons 2010-2013.

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 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. Which all sounds very complicated, but basically 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)
```



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)
```

This chart shows how McLaren had a good year in 2011 with its leading car always placed in the top 4 (at least, when the lead car was in a points scoring position; the chart could perhaps be modified to show a count of races where both cars are out of the points with an 11+ position?)

But the following chart shows how Red Bull dominated that season even more convincingly:

```
teamPerformance(results[results['team']=='red_bull',])+facet_wrap(~year)
```

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.

```
library(plyr)

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.

As well as comparing points, we can also compare the typical positions acheived 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.

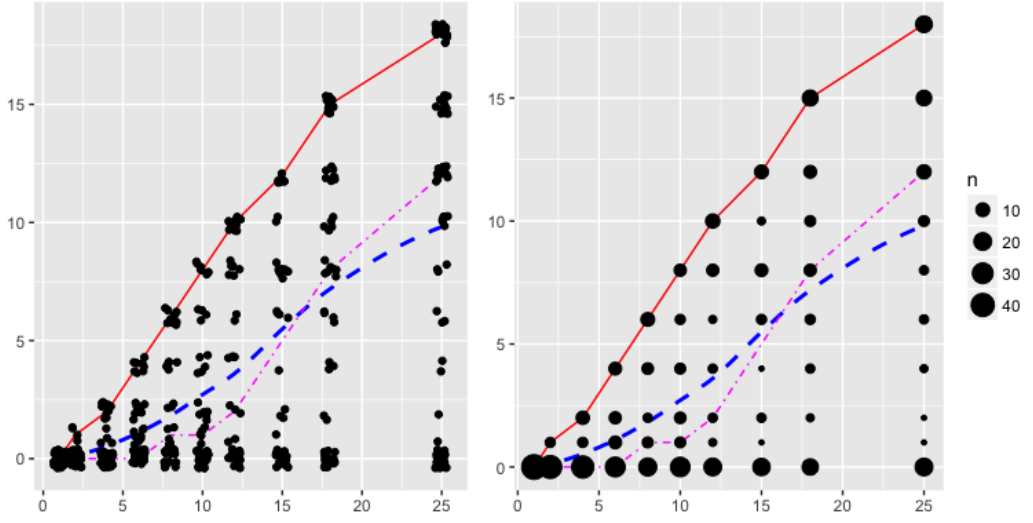
r1pos	med_r2pos	mean_r2pos
1	4.0	5.05
2	5.0	5.98
3	6.0	7.73
4	6.0	7.60
5	9.0	9.87
6	10.0	10.50
7	10.0	10.93
8	11.0	11.97
9	13.0	13.06
10	13.5	13.82

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)

teampoints.dualchart=function(results){
  g=ggplot(results,aes(x=r1points,y=r2points))
  g=g+stat_smooth(se=FALSE,colour='blue')
  g=g+geom_line(data=pos.points,aes(x=points,y=maximiser.points),col='red')
  g=g+geom_line(data=teamPerformance,aes(x=pts,y=med_r2pts),col='magenta')
  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.

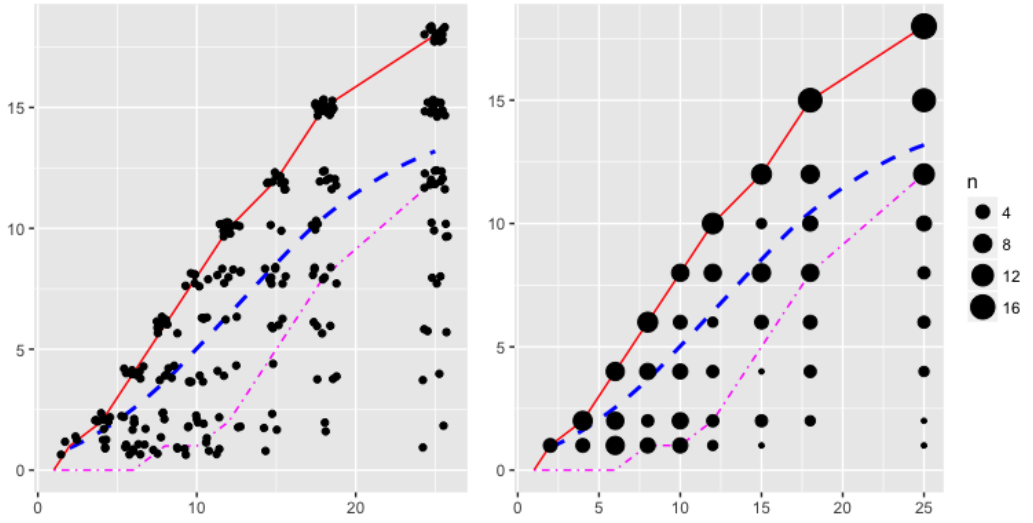
The red line is the upper bound on the points the lower classified driver within a team can score. The blue line is a fit that models the typical points contribution made by the second placed driver compared to their higher placed teammate. The magenta line plots the median number of points scored by 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

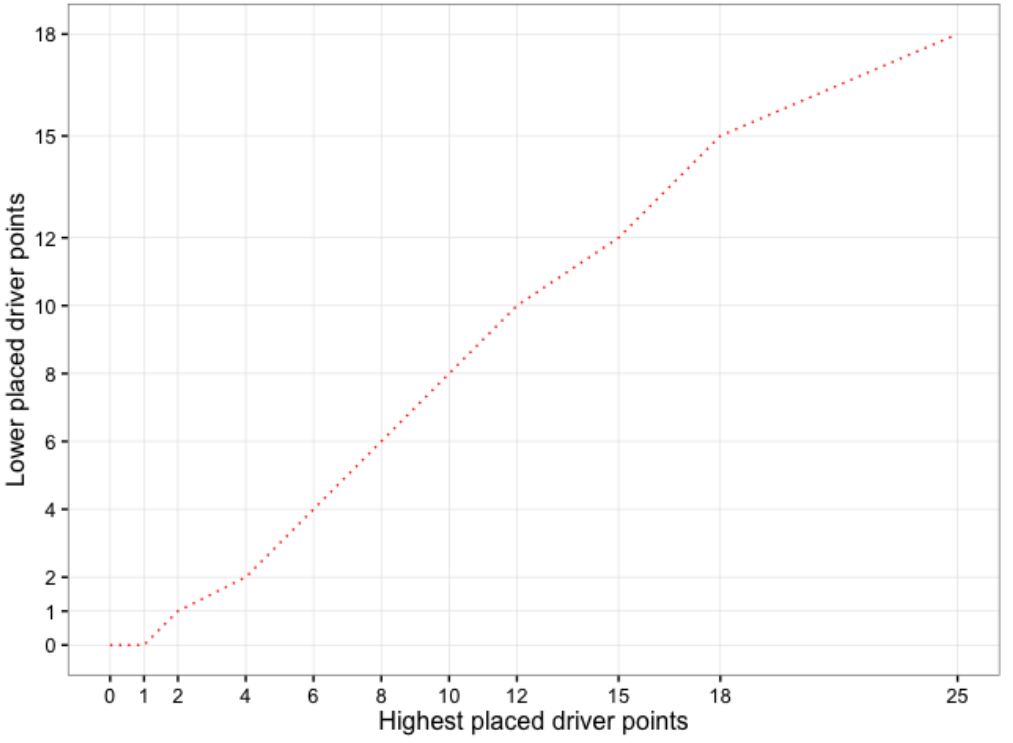
Trying to get a feel for how teams maximise points in general is one thing, but can we develop this chart style to produce an instrument that gives us a useful indication about how well a particular team is faring in terms of points maximisation?

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 how many races 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 it 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))
}
```

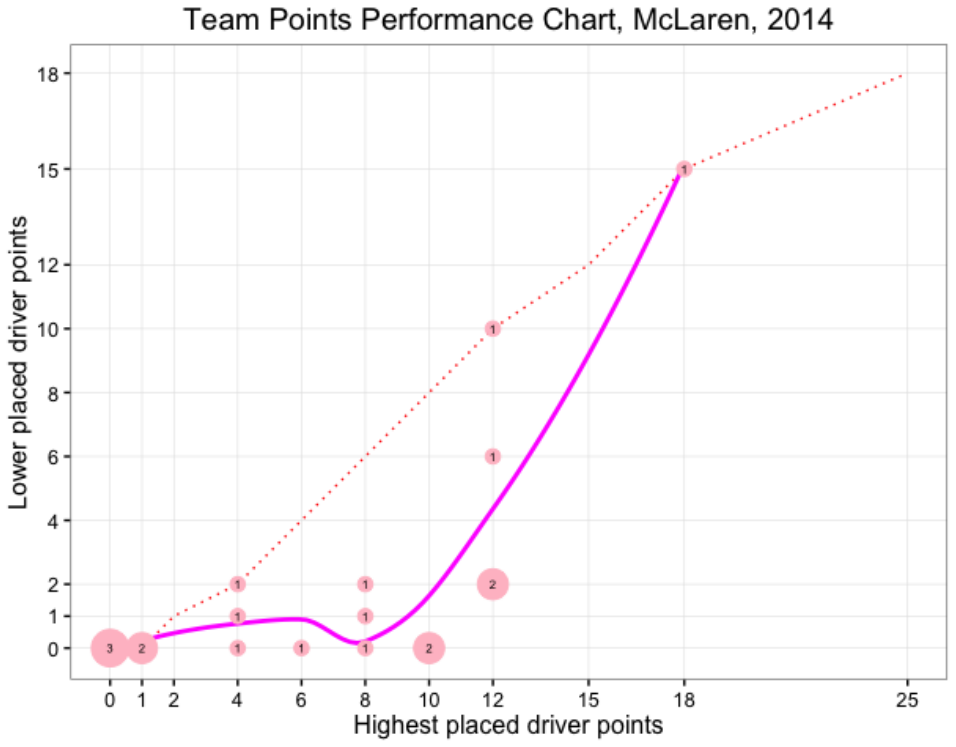
We can now generate a raceoff datatable from the race results of two drivers. As an example, let's see how the two McLaren drivers, Jenson Button and Kevin Magnussen, supported each other in the 2014 season.

```
#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, M\
cLaren, 2014')
```



one-way points performance chart

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 that indicate the maximum points haul depending on which driver 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+stat_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())
```

The red dashed line shows the maximum points haul line if *Driver 1* on the x-axis was higher placed in a particular race; the blue dashed line shows the maximum points haul line if *Driver 2* on the y-axis was higher placed.

We now need to generate a suitable summary dataset that counts the number of races where each particular combination of points was recorded.

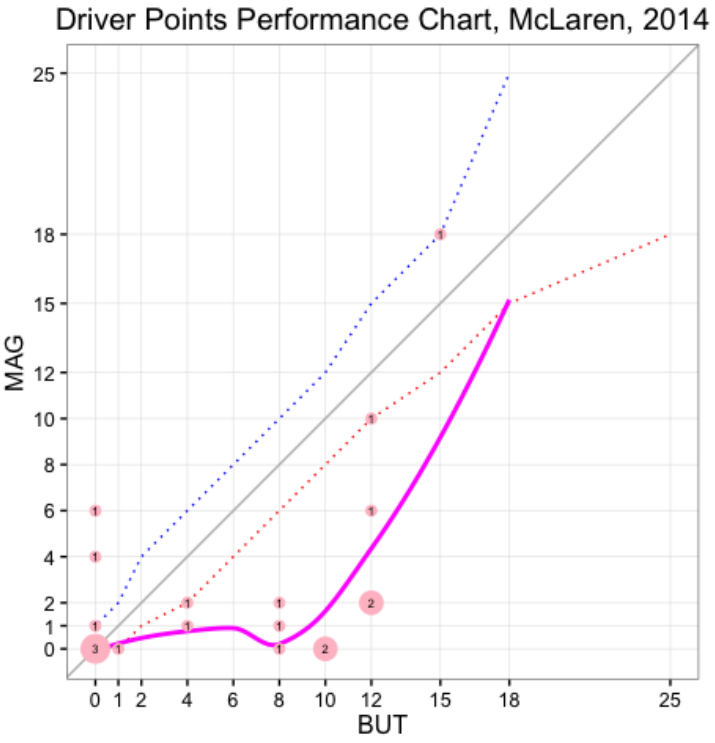
```
#Summary table counting how well each particular driver fares against the other
raceoff.summary=ddply(raceoff, .(points.x,points.y), summarise,
                      cnt=length(round))
```

This data can be overplotted onto the base 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','MAG')
```



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 that Kavin 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

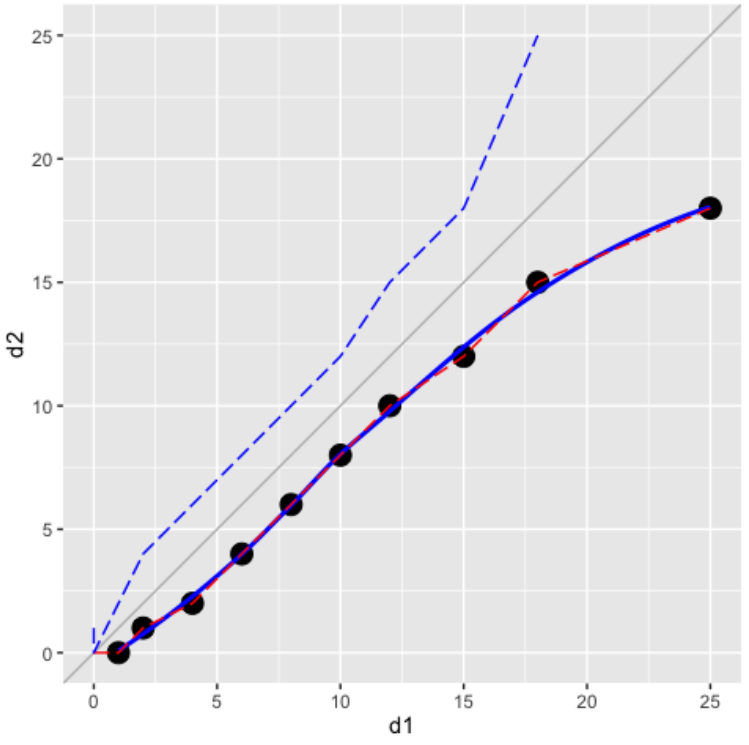
The *points performance charts* are a novel chart type that use maximum points haul guide lines to help us read sense into the marks placed on them. As a new chart type, it makes sense to use some extreme case dummy data to see what limiting cases look like.

Let's create a simple function to sketch a crude two-way points perfromance chart.

```
twowayPointsPerfSketch=function(df){
  g=ggplot(df,aes(x=d1,y=d2))+stat_sum(aes(size = ..n..))
  g=g+stat_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.

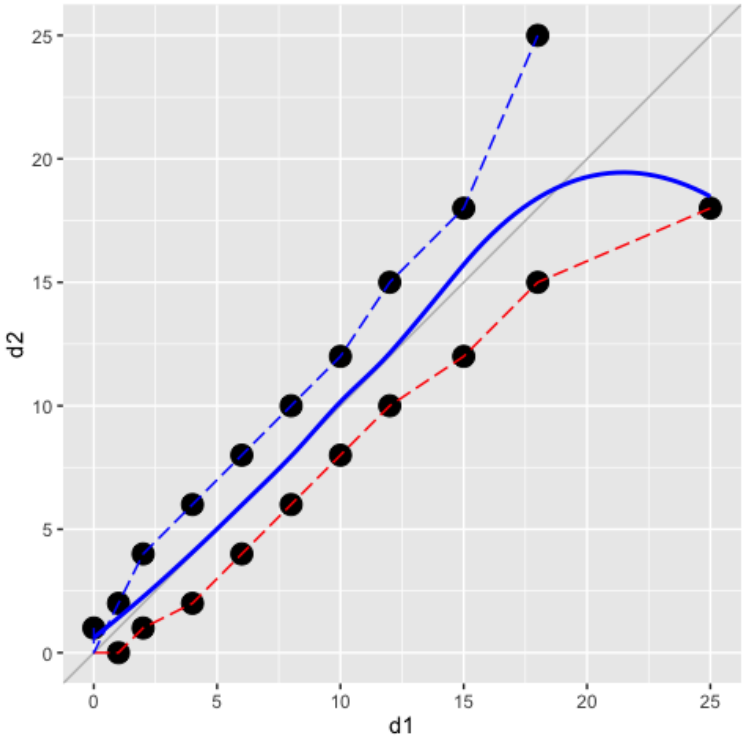
```
subpoints=c(NEWPOINTS[-1],0)
df=data.frame(d1=numeric(),d2=numeric())
for (i in seq(length(NEWPOINTS))) {
  df=rbind(df,data.frame(d1=NEWPOINTS[i],d2=subpoints[i]))
}
df=rbind(df,df)

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.

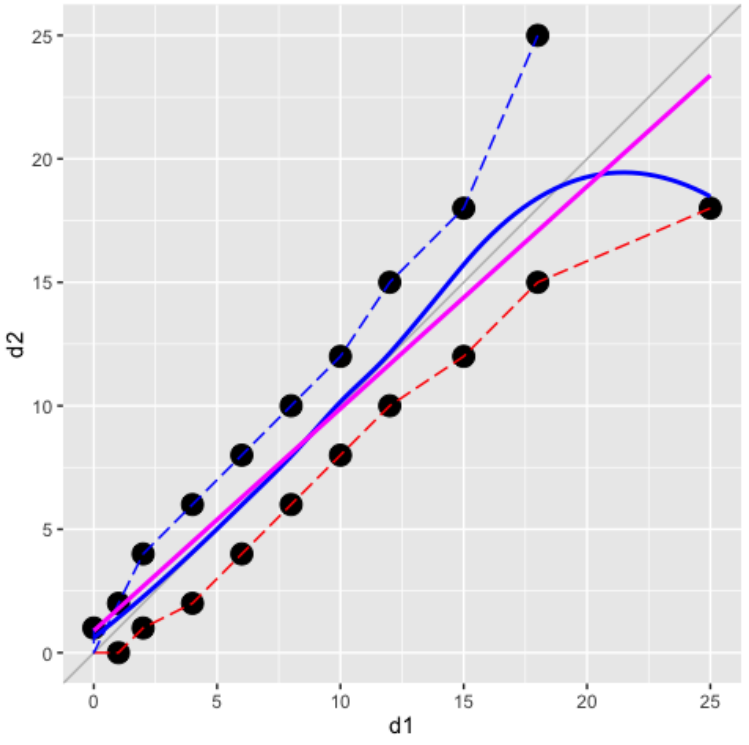
Second, for *twenty* races, where each driver scores each position in the top 10 once, beating his team mate, who once again maximises points below.

```
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)
```



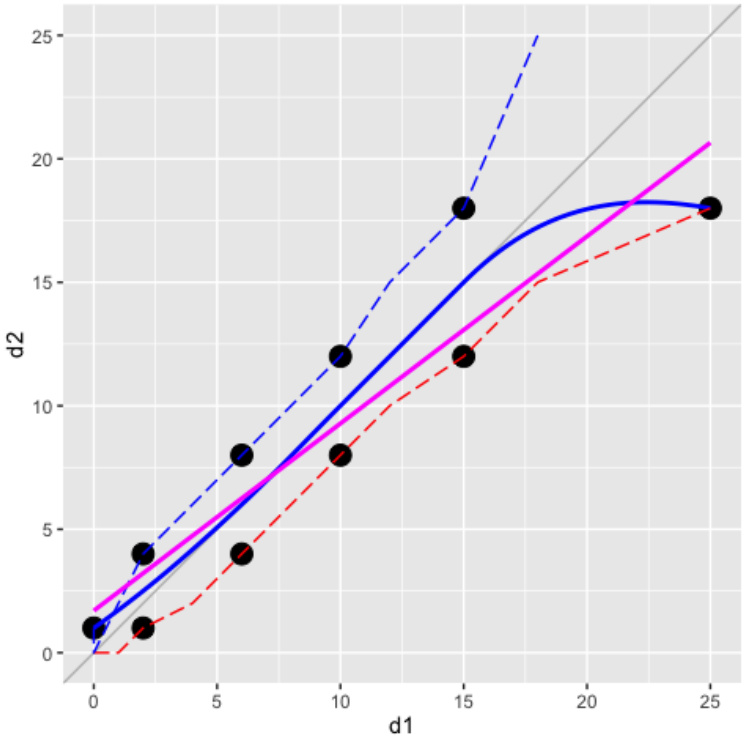
If we add in another best fit line, this time for a simple linear model, we can get a feel for how closely it tend towards the grey equal points line.

```
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.

Summary

In this chapter, we have explored 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.