Data Science for Software Engineering

An Introduction to Data Science for Software Engineers

Yim Register

Contents

Chapter 1

Introduction

The software engineering world is full of claims about best practices, languages, packages, styles, and workflows, but most software engineering students are never taught how to find, read, and interpret actual evidence on those topics. Is agile development really the secret to success? Do some languages actually cause more defects than others?

We have created a series of meaningful lessons that explore research in software engineering for the beginner R programmer. These lessons teach students to interpret and replicate research findings while learning meaningful results for their field in addition to common statistical methods. The lessons serve as a primer for software engineers to participate in a data-driven society, from advertising and business to combating misinformation and helping user experience.

Each lesson uses the R programming language, but does not assume any prior experience. The lessons cover introduction to R syntax, data cleaning and wrangling, data visualization, statistical tests, some predictive modeling, experimental methods, how to read academic research, and key findings from software engineering research. These lessons are intentionally written colloquially: far too many resources are written without remembering to help the learner laugh and find joy in their work. While you may not find all of the jokes funny, keep in mind that the author sure tried.

1.1 Who are these lessons for?

FIXME: learner personas

1.2 How do I get started?

FIXME: setup instructions

1.3 See Also

- Our license allows you to use this material however you want, provided you cite us as the original source.
- Contributions are very welcome, from questions and corrections to new material. All contributors are required to abide by our code of conduct.
- Please see our glossary for terms used in these lessons.

You may also enjoy:

- *Empirical Software Engineering Using R*
- *The Art and Science of Analyzing Software Data*
- *Cooperative Software Development*
- *Sharing Data and Models in Software Engineering*
- *Perspectives on Data Science for Software Engineering*
- *Making Software*

Chapter 2

Curiosity and Play

Here is a blog post that I wrote about this lesson and about playing with data and how to infuse joy into learning statistics.

2.1 Curiosity killed the cat, but satisfaction brought it back

Sometimes, some of the most intricate problems arise from simple musings among friends. This is one of the beautiful things about problem solving, programming, and statistics. When you are curious enough and excited about a problem, you may find yourself exploring every caveat just to prove your point. (*Be careful though, don't forgo sleep. We cover the detrimental effects of missing sleep on your coding performance in another lesson.*) While it is very important to tackle real problems that are genuinely affecting people's lives, first we have to learn to wrangle statistics and ask the right kinds of questions. Learning this skill will help you to ask critical questions when it really counts, like when building software for hospitals, news sites, financial institutions, or maybe the next rocket to Mars. One of the best ways to teach these skills is to lock on to a personally meaningful, but low-stakes, question; **and answer it**.

2.2 Playing with Data

In this lesson, you will learn to meaningfully play with data. So often in our academic careers we are expected to perfectly follow along and do things the "right" way, in order to build our skills to become an essential and productive member of a team/company/society. We can get through our math classes by "plugging-it-in" to equations, get through English by practicing flash cards, and get through History by memorizing dates. But we can gain a lot through creatively expressing ourselves in our learning. So this lesson will demonstrate

how to make queries to archived GitHub data, generate plots of interest, and answer our own curiosities (sometimes frivolous, but who cares?) You will have several chances to creatively express yourself in this lesson, while still being guided by examples.

2.3 GHTorrent and BigRQuery

We need to begin by getting **data**. Luckily the GHTorrent project has been archiving data from the **GitHub API** for the past several years, and we can get access to *commits, users, forks, repos, locations* (see **GitHub** in glossary) and more. While there are several ways to load this data into R, we are going to rely on `bigrquery` to interface with Google BigQuery and the public GHTorrent data on the Google Cloud. Here are the steps you need to take:

- Set up an account with Google Cloud. They do require a credit card number, which is ridiculous. But they will *not* auto-charge you when your free trial is up.
- Create a project from the Google Cloud console. Name it using something you'd like to use to access the project with `bigrquery`
- Make sure to enable the BigQuery API for the project. Go to the APIs & Services card on the console, and navigate to "Enable APIs and Services" where you can search for 'BigQuery' and enable it.
- You will be able to test out SQL queries on here
- The GHTorrent project for BigQuery is located here, with a small tutorial here

The next piece of this is to not only understand that the data is being held on the Google Cloud (that we can access with **SQL queries**, but to understand how we can do that from our local machines, in R. We will use the `bigrquery` package, with a tutorial here. `bigrquery` allows us to link our R code to our project on the cloud. You will be asked to authenticate. For the record, I did not know what the "cloud" really meant for a while, and was too afraid to ask.

2.3.1 DataFiles for this Lesson

For this lesson, we actually stored all of the queries in datafiles in this repository. Accessing the data required a credit card, which is unreasonable to expect from anyone just trying to use these lessons to learn. If you *do* want to perform different queries of your own, there are several ways to do that. I have included how you could do it using **BigQuery** and `bigrquery` (that's the R package, see the r in there?). Any time you see `query_exec()` it will be commented out and you will use the data we have provided from the result of that query as of September 2019. That way, you can see how the query would be executed but also you don't need to fiddle with APIs to follow along in this lesson.

```
library(bigrquery)
library(ggplot2)
```

```
#- bq_auth(path="servicetoken.json") # you will need to get your own service token from your acco
#- project <- "gitstats" #your project name here
```

2.4 Alan Turing's Birthday Examples (learn some SQL queries)

Here are some examples for using `bigrquery` to access GHTorrent data. Here, we take a look at Alan Turing's birthday (June 23) for the year 2016.This should help you get the hang of some SQL commands and collecting data. I chose Alan Turing because I'm an Artificial Intelligence nerd, and once even had a hamster named Turing. You can plug in your own birthday, or the birthday of someone else you admire. The point is to start playing.

There is a lot to unpack in your first SQL command. Here's a small recipe for reading the examples:

- the first line indicates what columns will come out in the final dataframe (`select  p.language as language, count(c.id) as num_commits`)
- `WHERE` is similar to `if` logic, and it grabs anything that fits the specified condition
- in the following example, we care about anything in the ghtorrent database `WHERE date(created_at) = date('2016-06-23))`
- that means that the commit data from ghtorrent has a `created_at` property, and we want data from where that date equals Alan Turing's birthday in 2016
- the `language` property of those commits to GitHub are stored in the ghtorrent-bq.ght.projects database, and we want to report all the languages used on the day we selected, except for if the `language` property is `null` because that's not helpful to us (unless you care about that)
- we `group_by` language, meaning that we want to see the number of commits for each language *separately*. We are going to compare the different languages and their relative number of commits on this day.
- and finally, we order everything in the data by the number of commits, in *descending order*. We want to see the *most* commits first.
- Do you have any predictions for which language was the most committed to?

```
#- # walking through an SQL example
#- # language with most commits on Alan Turing's birthday in 2016
#- sql_birthday <- "SELECT  p.language as language, count(c.id) as num_commits
#- from [ghtorrent-bq.ght.project_commits] pc join
#-      (SELECT id, author_id, created_at FROM [ghtorrent-bq.ght.commits] WHERE
#-      date(created_at) = date('2016-06-23') )c on pc.commit_id = c.id join
#-      (SELECT id, language, description
#-      FROM [ghtorrent-bq.ght.projects] WHERE language != 'null')p on p.id = pc.project_id join
```

```
#-      (SELECT login,  id
#-      FROM [ghtorrent-bq.ght.users]) u on c.author_id = u.id,
#- group by language
#- order by num_commits desc;"
```

2.5 Execute the query

We use `query_exec` to execute the query with `bigrquery` to the ghtorrent database, hosted on BigQuery. We get back a dataframe, stored in `bday_commits`.

```
#- # executing the query if you aren't using data provided
#- bday_commits  <- query_exec(sql_birthday, project = project, useLegacySql = FALSE) 

# reading in data for the above query that we stored earlier
bday_commits <- read.csv("data/curiosity_and_play/bdaycommits.csv")

kable(head(bday_commits)) %>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

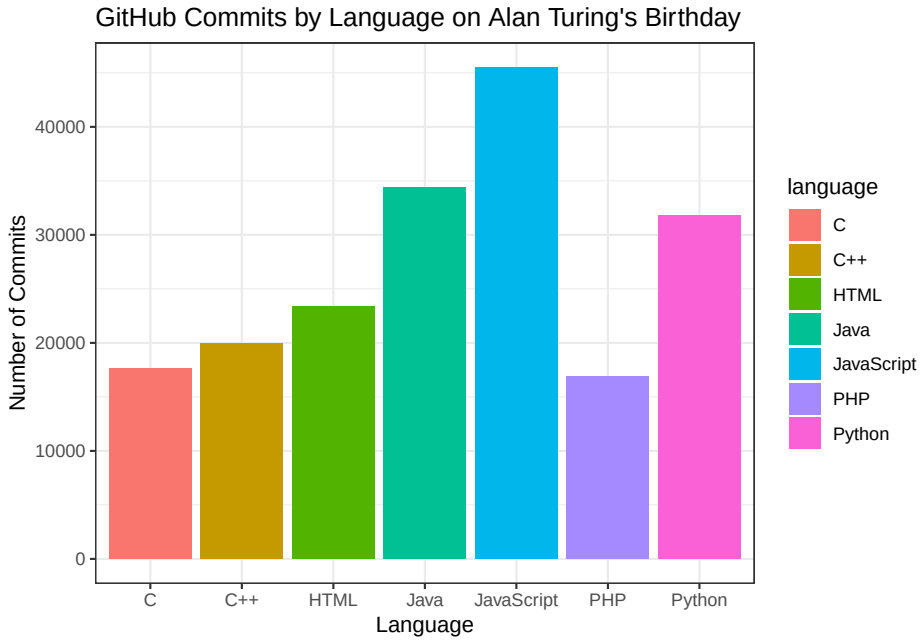
language	num_commits
JavaScript	45497
Java	34379
Python	31760
HTML	23366
C++	19956
C	17601

2.6 Visualize the Data

The following is a `ggplot` of the top 7 languages used on Alan Turing's birthday, by number of commits to each.

- `bday_commits[1:7,]` selects the first 7 rows in the dataframe
- `aes()` is the "aesthetic": what is the x and y that you are plotting?
- `fill = language` is simply how we get the pretty colors to show up
- `geom_bar` indicates that we want a bar plot. We could also use `geom_point()` for a scatter plot
- `stat="identity"` indicates that the bar plot should use the raw value of num_commits instead of a different statistical transformation
- `xlab`,`ylab`,and `ggtitle` are the x-axis label, y-axis label, and title, respectively theme_bw()‘ is "Theme Black and White" and I just like it better

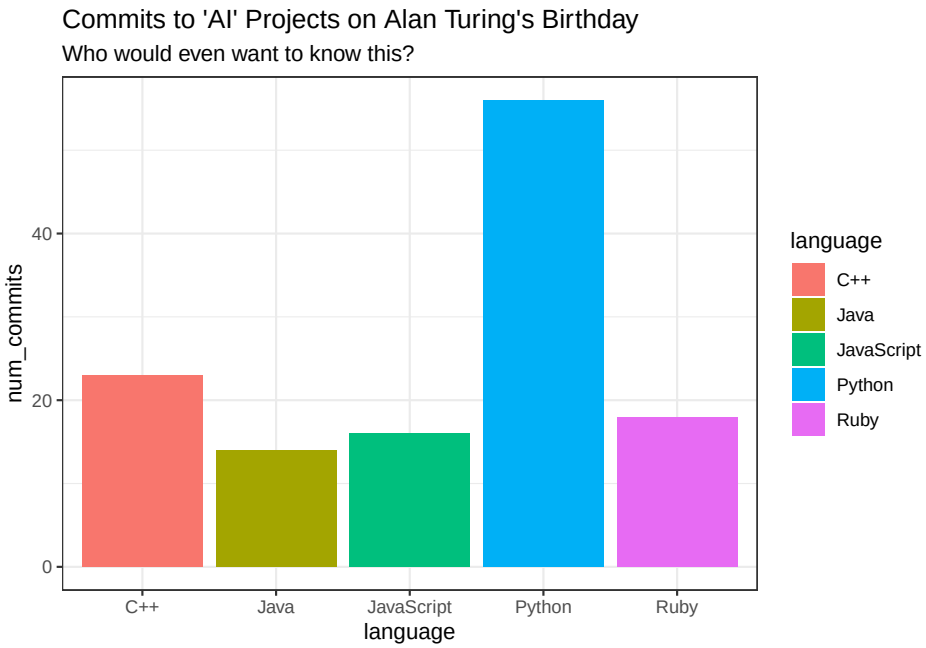
```
plt = ggplot(bday_commits[1:7,],aes(language,num_commits,fill=language))+
  geom_bar(stat="identity")+
  xlab("Language")+
  ylab("Number of Commits")+
  ggtitle("GitHub Commits by Language on Alan Turing's Birthday")+
  theme_bw()
plt
```



I can't imagine that Alan Turing would have been the biggest JavaScript fan. Let's take a look at projects where the project description includes "AI". Here, we see Python emerge as top commits for the day.

```
#- # languge with most commits on Alan Turing's birthday in 2016
#- sql_example2 <- "SELECT p.description as description, p.language as language, count(c.id) as n
#- from [ghtorrent-bq.ght.project_commits] pc join
#-       (SELECT id, author_id, created_at FROM [ghtorrent-bq.ght.commits] WHERE
#-       date(created_at) = date('2016-06-23') )c on pc.commit_id = c.id join
#-       (SELECT id, language, description
#-       FROM [ghtorrent-bq.ght.projects] WHERE language != 'null' and description LIKE '%AI%')p o
#-       (SELECT login,  id
#-       FROM [ghtorrent-bq.ght.users]) u on c.author_id = u.id,
#- group by description,language
#- order by num_commits desc;"

#- # executing the query if you aren't using data provided
```

```
#- example2  <- query_exec(sql_example2, project = project, useLegacySql = FALSE)

# reading in data for the above query that we stored earlier
example2 <- read.csv("data/curiosity_and_play/example2.csv")

plt = ggplot(example2[1:6,],aes(language,num_commits,fill=language))+
  geom_bar(stat="identity")+
  ggtitle("Commits to 'AI' Projects on Alan Turing's Birthday",subtitle="Who would ever
  theme_bw()
plt
```




```
python_desc <- example2[example2$language=='Python',]
python_desc <- python_desc[order(-python_desc$num_commits),]

kable(python_desc[1:10,])%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

	description
1	Record for playing with OpenAI Gym
5	rllab is a framework for developing and evaluating reinforcement learning algorithms, fully compatible with O
8	AI of bomb iss(almost tetris) on NES
9	:alien: An AI project. :alien:
16	AI for 2048 using simple reinforcement learning
27	SIMA and RAIN comparison
35	The OAI Harvest module handles metadata gathering between OAI-PMH v.2.0 compliant repositories.
37	This depot is used for learning of AI for robotics in Udacity
46	Website version of the AIS Network Mapper project
53	Gen-6 Pokemon Battling Sim and AI

2.7 Language Growth over Time

Let's investigate another question. Do you know anyone who is obsessed with a certain language? Maybe they can't stop talking about how Julia (*programming language*) is gonna rule the world, or how Golang is the future because of all that multithreading (I've never used Golang). Or maybe you're a straight up *fan* of some language and you want to make sure that other people know how awesome it is (Scala, anyone?) This is a fun exercise in seeing how languages grow over time, by the number of commits for those projects on GitHub. In this example:

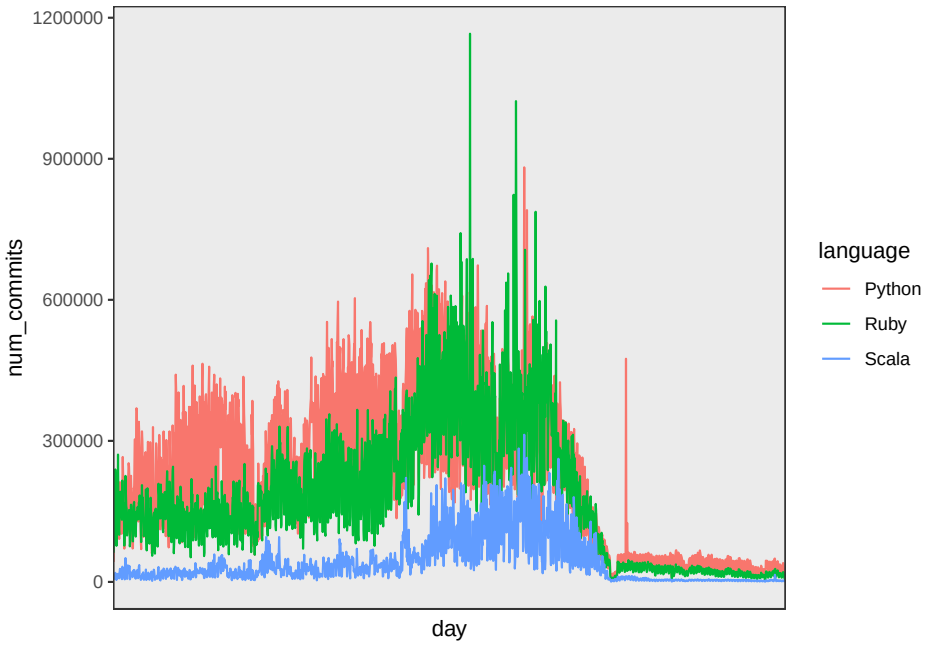
- we `SELECT` language, day, and num_commits to make a dataframe where we can monitor commits over time for different languages
- we use `WHERE language == 'Python'` to select all projects where the language is Python
- the `||` syntax indicates a logical OR symbol. We are collecting Python *or* Scale *or* Julia *or* Ruby… *or* whatever else you'd like to plug in.
- We `group_by(language,day)` to get the number of commits, by language, by day, over time.
- And finally, we order the data by the greatest number of commits first

```
#- language_sql <- "SELECT  p.language as language, date(created_at) as day,count(c.id) as num_co
#- from [ghtorrent-bq.ght.project_commits] pc join
#-      (SELECT id, author_id, created_at FROM [ghtorrent-bq.ght.commits] WHERE
#-      date(created_at) between date('2012-01-01')
#-                          and  date('2016-09-05') )c on pc.commit_id = c.id join
#-      (SELECT id, language, description
#-      FROM [ghtorrent-bq.ght.projects] WHERE language == 'Scala' || language == 'Python' || lan
#-      (SELECT login,  id
#-      FROM [ghtorrent-bq.ght.users]) u on c.author_id = u.id,
#- group by language,day
#- order by num_commits desc;"
#- # executing the query if you aren't using data provided
```

```
#- lang_growth  <- query_exec(language_sql, project = project, useLegacySql = FALSE)

# reading in data for the above query that we stored earlier
lang_growth <- read.csv("data/curiosity_and_play/lang_growth.csv")

kable(head(lang_growth))%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

language	day	num_commits
Ruby	2014-06-26	1166068
Ruby	2014-10-21	1022681
Python	2014-11-11	881825
Ruby	2014-10-15	823475
Python	2014-11-18	790873
Ruby	2014-12-10	787341

```
# I want the top 3

summary <- lang_growth %>%
             group_by(language)%>%
             summarise(total_num_commits = sum(num_commits))%>%
             arrange(-total_num_commits)%>%
             head(3)

top_langs <- lang_growth[lang_growth$language %in% summary$language,]

plt =ggplot(top_langs,aes(day,num_commits,color=language,group=language))+
  geom_line()+
  theme_bw()+
  theme(axis.text.x=element_blank(),axis.ticks.x=element_blank())
plt
```



2.8 Commits to GitHub: Canada Day vs. July 4th

Alright, now that we have explored how to use `bigrquery` and generate some simple queries and plots, let's approach another curiosity. It occurred to me that I was curious about holidays. This most recent 4th of July I checked the GitHub data to see if people commit differently on the 4th of July than they do normally. It then occurred to me that I might compare it to Canada Day, which is very close in date to American Independence Day. The queries will get more complex as we continue to explore, and refine what we want to know. You may be developing an intuition for what you believe to be true; of course people don't commit on a holiday. They're not working! But what about the case where company repos are kept private, and the data that GHTorrent has access to is actually "for fun" repos, and a day off is the perfect day for that programmer to keep making their alien game. Intuition is certainly important when asking statistical questions, as it guides us towards different factors to account for and questions to ask. But remember, your intuition may be the exact opposite of someone else's.

2.8.1 Total Commits

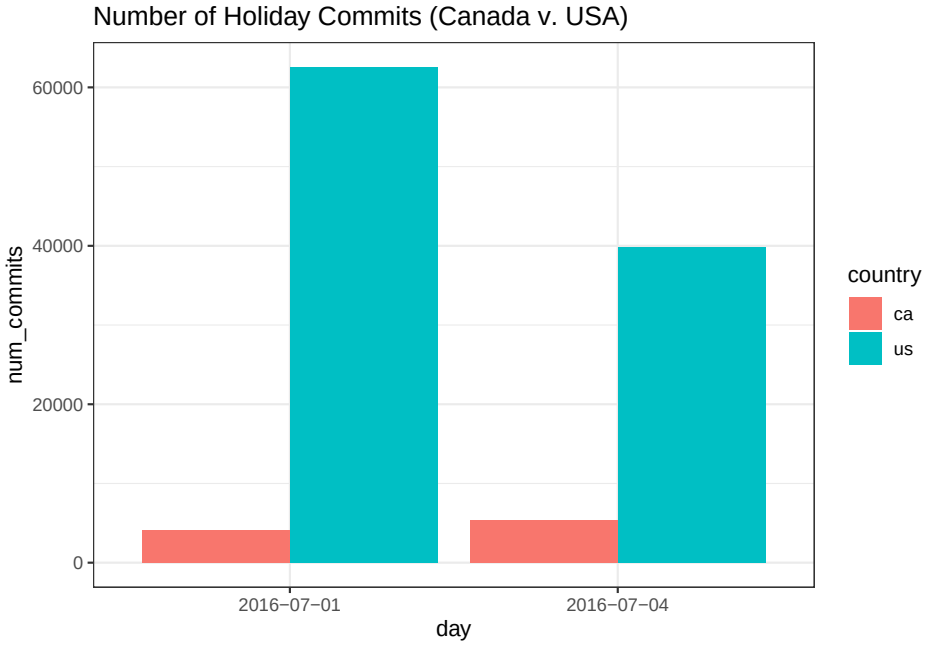
We start by taking a look at the total commits to GitHub on either Canada Day or Independence Day, from either Canada or the United States. We immediately discover that raw count is *not* going to be helpful in making any meaningful

comparsions. The US has *orders of magnitude* more commits on any given day than Canada does. While that is good to know, we need to account for it if we actually want to make comparisons.

```
#- data_sql <- "select  u.country_code as country, date(c.created_at) as day, count(c.
#- from [ghtorrent-bq.ght.project_commits] pc join
#-      (SELECT id, author_id, created_at FROM [ghtorrent-bq.ght.commits] WHERE
#-      date(created_at) = date('2016-07-01')
#-                            or date(created_at)= date('2016-07-04') )c on pc.commit_i
#-      (SELECT id,
#-      FROM [ghtorrent-bq.ght.projects]) p on p.id = pc.project_id join
#-      (SELECT login, location, id, country_code,
#-      FROM [ghtorrent-bq.ght.users]
#-      WHERE country_code = 'us' or country_code = 'ca') u on c.author_id = u.id,
#- group by country, day
#- order by num_commits desc;"

#- # executing the query if you aren't using data provided
#- total_commits  <- query_exec(data_sql, project = project, useLegacySql = FALSE)

# reading in data for the above query that we stored earlier
total_commits <- read.csv("data/curiosity_and_play/total_commits.csv")

kable(total_commits)%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

country	day	num_commits
us	2016-07-01	62582
us	2016-07-04	39811
ca	2016-07-04	5305
ca	2016-07-01	4096

```
plt = ggplot(total_commits, aes(day,num_commits,fill=country))+
  geom_bar(stat="identity",position="dodge")+
  ggtitle("Number of Holiday Commits (Canada v. USA)")+
  theme_bw()
plt
```

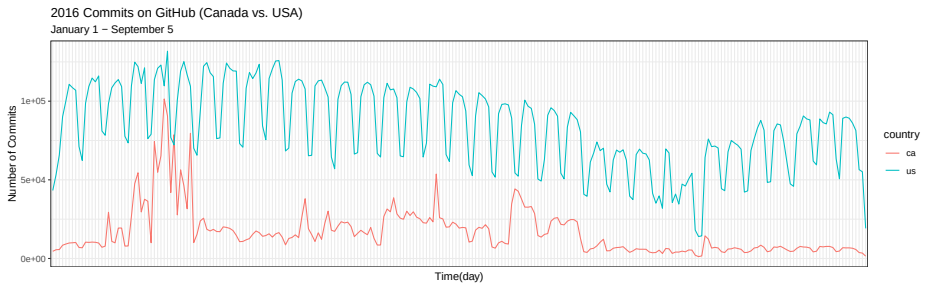


2.8.2 Yearly Commits

What we actually need to care about is, *on average* do number of commits differ on a holiday? In order to do that, let's look at number of commits per day across a year sample, between Canada and USA.

```
#- data_sql <- "select  u.country_code as country, date(c.created_at) as day, count(c.id) as num_
#- from [ghtorrent-bq.ght.project_commits] pc join
#-      (SELECT id, author_id, created_at FROM [ghtorrent-bq.ght.commits] WHERE
#-      date(created_at) between date('2016-01-01')
#-                           and  date('2016-09-05') )c on pc.commit_id = c.id join
#-      (SELECT id,
#-      FROM [ghtorrent-bq.ght.projects]) p on p.id = pc.project_id join
#-      (SELECT login, location, id, country_code,
#-      FROM [ghtorrent-bq.ght.users]
#-      WHERE country_code = 'us' or country_code = 'ca') u on c.author_id = u.id,
#- group by country, day
#- order by num_commits desc;"

#- # executing the query if you aren't using data provided
#- year_commits  <- query_exec(data_sql, project = project, useLegacySql = FALSE)

# reading in data for the above query that we stored earlier
year_commits <- read.csv("data/curiosity_and_play/year_commits.csv")
```

```
plt = ggplot(year_commits,aes(day,num_commits,colour=country))+
  geom_line(aes(group=country))+
  theme_bw()+
  ylab("Number of Commits")+
  xlab("Time(day)")+
  theme(axis.text.x=element_blank(),axis.ticks.x=element_blank())+
  ggtitle("2016 Commits on GitHub (Canada vs. USA)", subtitle="January 1 - September 5
plt
```



2.8.3 What are those fluctuations?

Look at those patterns, and take a guess at what you are seeing. There appears to be a regular, periodic trend in the data. That means that it seems to rise and fall in a consistent way. It seems less prominent for Canada, but that could also be due to scaling. What is your hunch about these curves? What are we looking at?

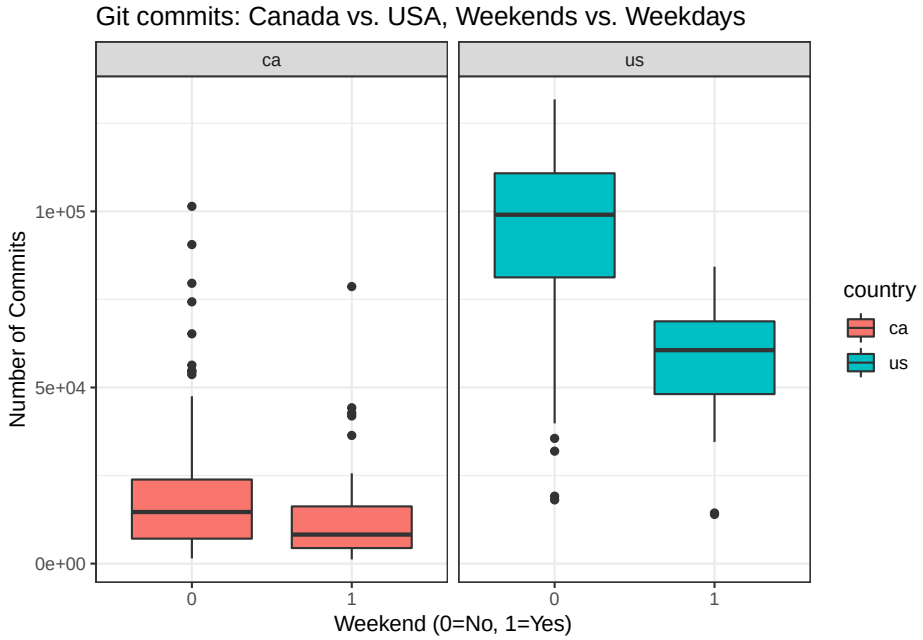
> The answer: *weekends*

If you check the weekday for the dips in Number of Commits, you can see that on Saturdays and Sundays, the Number of Commits is less than during a weekday.

Let's make sure:

```
year_commits$weekday <- weekdays(as.Date(year_commits$day))
year_commits$isWeekend <- 0
year_commits$isWeekend[year_commits$weekday=='Saturday'|year_commits$weekday=='Sunday'
```

```
ggplot(year_commits,aes(as.factor(isWeekend),num_commits,fill=country))+
  geom_boxplot()+
  facet_wrap(~country)+
  xlab("Weekend (0=No, 1=Yes)")+
  ylab("Number of Commits")+
  ggtitle("Git commits: Canada vs. USA, Weekends vs. Weekdays")+
  theme_bw()
```




```
summary <- year_commits %>%
  group_by(country) %>%
  summarise(avg = mean(num_commits),sd=sd(num_commits))

kable(summary)%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

country	avg	sd
ca	16707.84	15061.61
us	83648.84	26215.89

```
kable(ddply(year_commits,c('country','isWeekend'),summarise,mean = mean(num_commits),sd(num_commi
  kable_styling(bootstrap_options = c("striped", "hover"))
```

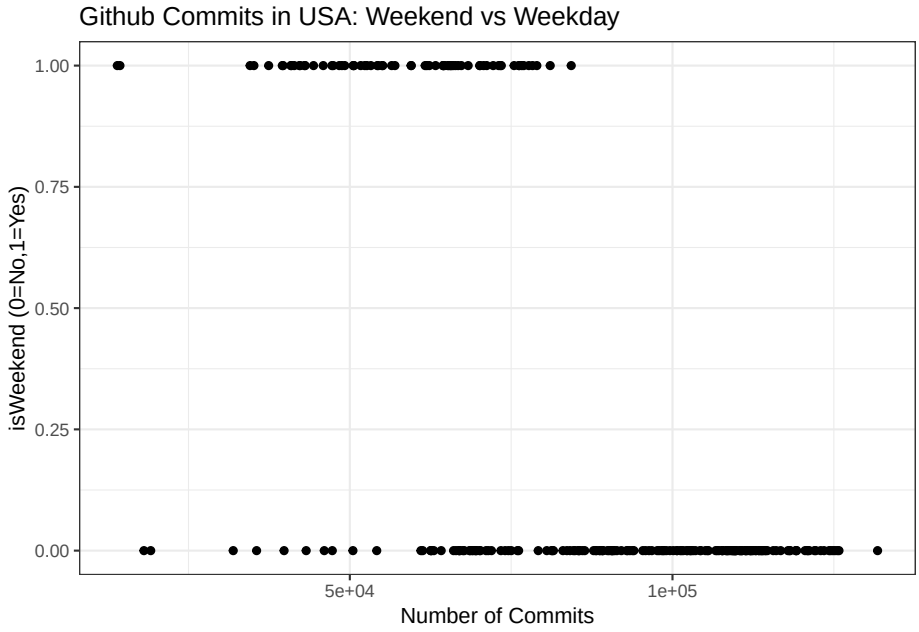
country	isWeekend	mean	sd(num_commits)
ca	0	18400.07	15722.57
ca	1	12547.75	12446.70
us	0	94069.36	22410.14
us	1	58031.75	14815.83

2.8.4 Logistic regression for predicting your weekend commits

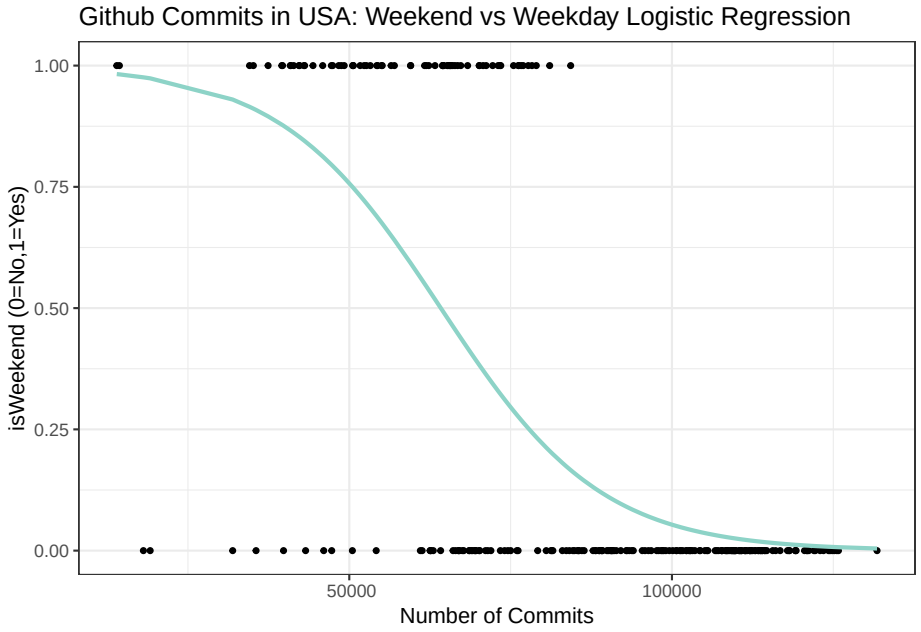
A useful skill that you may want to return to is something called **logistic regression**. You may have seen "line of best fit" before, where you try to calculate the best fitting line across a scatter plot of numbers. What if you only have a 0 or 1 variable to map? In our case, we are dealing with "weekend" or "not a weekend". Is there any way that we could look at the number of commits, and infer whether it was a weekend or not? We can use something called logistic regression. If you look up logistic regression, you will see lots and lots of plots like the one below. It looks like two bars of points, one clustered at the "1" (weekend), and another clustered across "0" (not a weekend). If the bars were exactly the same, that would mean that across our variable `num_commits`, there is no difference between weekend or not a weekend. The number of commits would be totally unrelated to that **binary** variable, and it might not even be worth measuring, because it doesn't tell us very much. But we *know* there's a difference, and we can see it in the logistic plot as well. There tend to be more commits (higher `num_commits` values) in the "1" category as opposed to "0". So could we go backwards? What if we knew that there were 100,567 commits one day in the USA. Can we take a guess at whether it was a weekend or not? We use logistic regression to map the probability across the `num_commits` variable. This is tough to process at first. How could we go from 0 or 1 to a continuous line? Well, it has to do with how the points are clustered and distributed across the 0 or 1 variables.

By the way, this is all part of learning "machine learning". I see so much hype around this subject, and "Big Data", but I just want you to know that you are already on your way.

```
ggplot(year_commits[year_commits$country=='us',],aes(num_commits,isWeekend))+
  geom_point()+
  xlab("Number of Commits")+
  ylab("isWeekend (0=No,1=Yes)")+
  ggtitle("Github Commits in USA: Weekend vs Weekday")+
  theme_bw()
```




```
logistic <- glm(isWeekend ~ num_commits, data=year_commits[year_commits$country=='us',],family =

vals <- predict(logistic,data.frame(num_commits=year_commits[year_commits$country=='us',]$num_com

plt <- ggplot(year_commits[year_commits$country=='us',],aes(num_commits,isWeekend))+
  geom_point(size=1)+
  geom_line(aes(num_commits,vals),color="#8DD3C7",size=1)+
  theme_bw()+
  scale_x_continuous(labels = function(x) format(x, scientific = FALSE))+
  xlab("Number of Commits")+
  ylab("isWeekend (0=No,1=Yes)")+
  ggtitle("Github Commits in USA: Weekend vs Weekday Logistic Regression")
plt
```



2.8.5 Making Predictions

We now have a curve that demonstrates the likelihood that given a certain number of commits, if it is a weekend or not a weekend. So, if there were 100,000 commits, is it a weekend or not a weekend? According to this `predict` function on our **test data**, the likelihood that 100,000 commits was done on a Saturday or Sunday is only 5%. But if we see 50,000 commits, suddenly it is a 75% that it's a weekend, because that is much more in line with the kinds of commits happening on a weekend. We use our data to create a useful, predictive model.

```
newdata = data.frame(num_commits=100000)
predict(logistic,newdata,type="response")
```

```
##           1
## 0.05353886
```

```
newdata = data.frame(num_commits=50000)
predict(logistic,newdata,type="response")
```

```
##         1
## 0.7572345
```

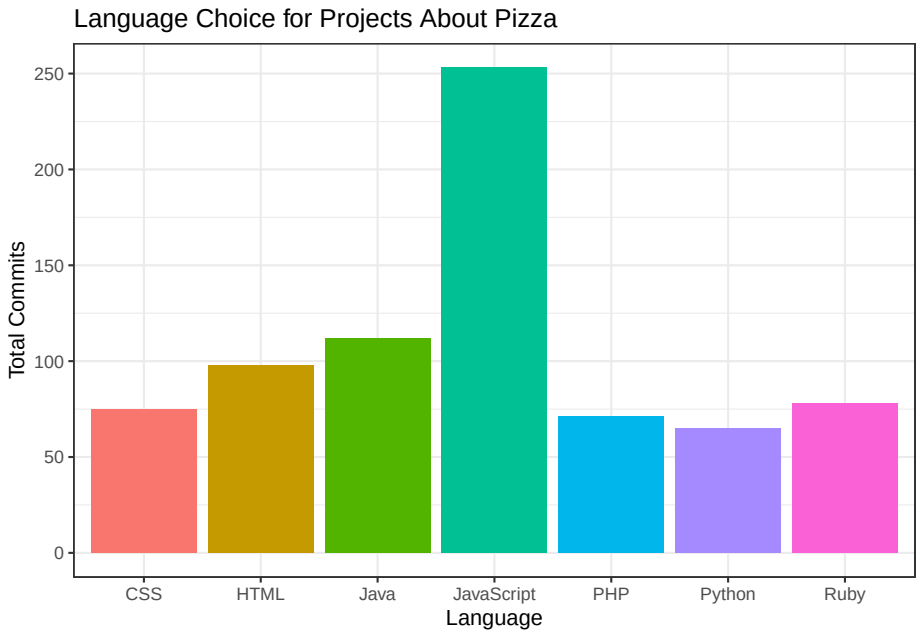
2.9 Time for a Cooldown: Pizza & Cats

Okay, you've gotten through a lot already today. We are going to finish up the lesson with some fun. One thing we have access to in the ghtorrent database is *project descriptions*. We saw this earlier when we tried to find Artificial Intelligence projects. Now, we will explore more words that we want to play with. This should be a low-stakes investigation, where you plug in some of your silliest words into the SQL query and generate some statistics about your findings. Think of everything you've accomplished today. SQL queries to the cloud, barplots, boxplots, comparisons, time series, statistical testing, logistic regression, and more. Now it's time to just look up stupid project descriptions and laugh about them.

```
#- pizza_sql <- "SELECT p.description as description, p.language as language, count(c.id) as num_
#- from [ghtorrent-bq.ght.project_commits] pc join
#-       (SELECT id, author_id, created_at FROM [ghtorrent-bq.ght.commits] WHERE
#-       date(created_at) between date('2012-01-01')
#-                            and  date('2016-09-05') )c on pc.commit_id = c.id join
#-       (SELECT id, language, description
#-       FROM [ghtorrent-bq.ght.projects] WHERE language != 'null' and description LIKE '%pizza%')
#-       (SELECT login,  id
#-       FROM [ghtorrent-bq.ght.users]) u on c.author_id = u.id,
#- group by description,language
#- order by num_commits desc;"

#- # executing the query if you aren't using data provided
#- pizza  <- query_exec(pizza_sql, project = project, useLegacySql = FALSE)

# reading in data for the above query that we stored earlier
pizza <- read.csv("data/curiosity_and_play/pizza.csv")

pizza <- pizza[order(-pizza$num_commits),]

kable(head(pizza))%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

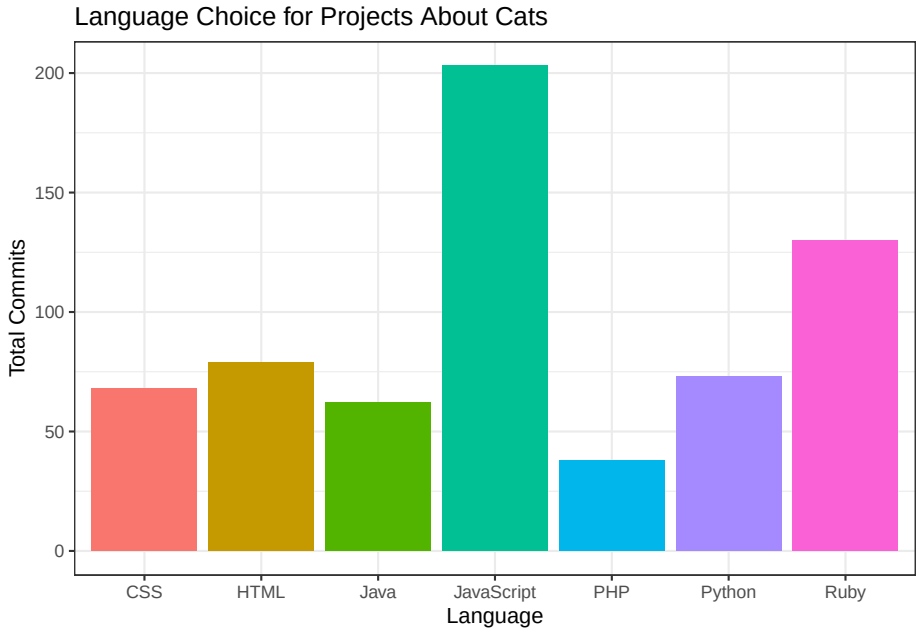
description	language	n
A Jekyll-powered site generated by jekyll:pizza:pizza	CSS	
allows group orders on pizza.de // ermöglicht Gruppenbestellungen auf pizza.de	Ruby	
Crazy Uncle Buck's pizza ordering system.	Ruby	
This is a node.js wrapper for the dominos pizza apis	JavaScript	
Where is the best :pizza: in a given city?	Ruby	
Organisation of a student-run pizza seminar in mathematics at University of Augsburg, Germany	Tcl	

```
summary <- pizza %>%
 count(language)
summary <- summary[order(-summary$n),]

plt = ggplot(summary[1:7,],aes(language,n,fill=language))+
  geom_bar(stat="identity")+
  theme_bw()+
  xlab("Language")+
  ylab("Total Commits")+
  ggtitle("Language Choice for Projects About Pizza")+
  theme(legend.position = "none")
plt
```




```
#- cats_sql <- "SELECT p.description as description, p.language as language, count(c.i
#- from [ghtorrent-bq.ght.project_commits] pc join
#-      (SELECT id, author_id, created_at FROM [ghtorrent-bq.ght.commits] WHERE
#-      date(created_at) between date('2012-01-01')
#-                           and  date('2016-09-05') )c on pc.commit_id = c.id join
#-      (SELECT id, language, description
#-      FROM [ghtorrent-bq.ght.projects] WHERE language != 'null' and description LIKE
#-      (SELECT login,  id
#-      FROM [ghtorrent-bq.ght.users]) u on c.author_id = u.id,
#- group by description,language
#- order by num_commits desc;"
```

```
#- # executing the query if you aren't using data provided
#- cats  <- query_exec(cats_sql, project = project, useLegacySql = FALSE)

# reading in data for the above query that we stored earlier
cats <- read.csv("data/curiosity_and_play/cats.csv")

cats <- cats[order(-cats$num_commits),]

summary <- cats %>%
  count(language)

summary <- summary[order(-summary$n),]
plt = ggplot(summary[1:7,],aes(language,n,fill=language))+
  geom_bar(stat="identity")+
  theme_bw()+
  xlab("Language")+
  ylab("Total Commits")+
  ggtitle("Language Choice for Projects About Cats")+
  theme(legend.position = "none")
plt
```



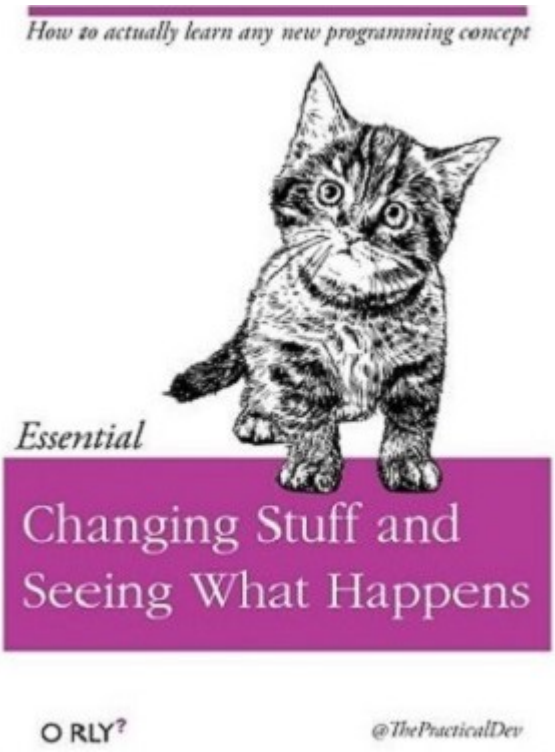
Chapter 3

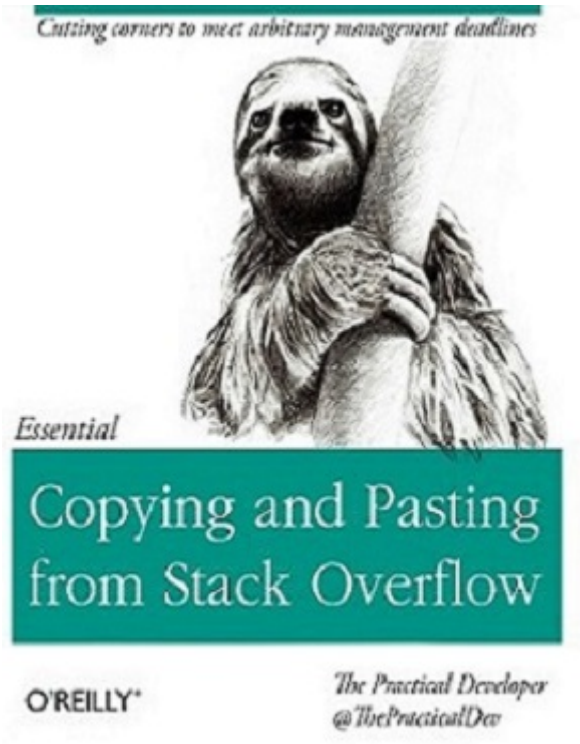
Stack Overflow Queries

3.1 I am thankful for Stack Overflow

You've come through for me in my dark moments of swearing at Tetris, forgetting any and all JavaScript syntax, and more. I dedicate this lesson to all the Stack Overflow contributors who didn't make me feel like a moron, and to all the new students fixing their bugs one at a time with 26 tabs open.


Software can be chaotic, but we make it work
Expert
Trying Stuff Until it Works
O RLY?
The Practical Developer
@ThePracticalDev


How to actually learn any new programming concept
Essential
Changing Stuff and Seeing What Happens
O RLY?
@ThePracticalDev



Fake book covers from *the Practical Developer*, mimicking O'Reilly textbooks

3.2 Stack Exchange Queries

One of the coolest things about learning Data Science and statistics is that we can apply our methods to any dataset that interests us. This is an excellent opportunity for creativity and expression. But it can be difficult to know where to begin; especially because our questions might be too advanced, or difficult to get data for. These lessons approach this problem by analyzing data from Software Engineering research, and guiding the learner through the findings. We also use sources like Stack Overflow, GitHub, and Eclipse (Java IDE) Bug Reports. For this lesson, it's all about Stack Overflow tags. Stack Overflow is a forum where developers can ask and answer questions about code. There are other features to the site, like gaining reputation and earning badges, but above all else they market the site as "ask questions, get answers, no distractions". There is certainly a culture to be reckoned with; as some people can be unkind to beginners on the site. But never fear, you are a coder and you belong.

In this lesson we use the Stack Exchange API to get data from Stack Overflow. We use this query to get a **.csv file** that pulls data from Stack Overflow. We can actually download that file to our local machine and work with the data

from there.

3.3 Load Your Libraries

```
library(kableExtra)
library(ggplot2)
library(dplyr)
library(tm)
```

3.4 Load the Data

The data already exists in the `data/stackoverflow` folder, but if you download any of your own data from Stack Exchange, you will need to put that csv file in the correct location so that this R code can reach it in the correct path.

```
data <- read.csv("data/stackoverflow/top10tags.csv")

#this is the same as calling head(data,13) which shows the top 13 rows, but put inside
kable(head(data,13))%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

Month	TagName	X
2008-07-01 00:00:00	c#	3
2008-07-01 00:00:00	html	1
2008-08-01 00:00:00	android	4
2008-08-01 00:00:00	c#	513
2008-08-01 00:00:00	c++	164
2008-08-01 00:00:00	html	111
2008-08-01 00:00:00	ios	9
2008-08-01 00:00:00	java	220
2008-08-01 00:00:00	javascript	161
2008-08-01 00:00:00	jquery	28
2008-08-01 00:00:00	php	162
2008-08-01 00:00:00	python	124
2008-09-01 00:00:00	android	9

```
#you can also use the View command to inspect the entire data frame
#View(data)
```

3.5 Summarize the Variables

Here we have the top 10 tags used on StackOverflow over time, with the number of questions containing that tag per month since 2008. Let's summarize a bit

more of what we have by going over each variable and making some tables. This is just a nice way to get yourself situated with the data.

```
summary(data$TagName)
```

```
##    android         c#        c++       html        ios       java
##        133        134        133        134        133        133
## javascript     jquery        php     python
##        133        133        133        133
```

```
summary(data$X)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       1    4182    8364    8675   12476   24202
```

```
kable(tail(data,13))%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

	Month	TagName	X
1320	2019-07-01 00:00:00	jquery	4129
1321	2019-07-01 00:00:00	php	8429
1322	2019-07-01 00:00:00	python	23483
1323	2019-08-01 00:00:00	android	5262
1324	2019-08-01 00:00:00	c#	5679
1325	2019-08-01 00:00:00	c++	2460
1326	2019-08-01 00:00:00	html	4397
1327	2019-08-01 00:00:00	ios	1820
1328	2019-08-01 00:00:00	java	7089
1329	2019-08-01 00:00:00	javascript	10873
1330	2019-08-01 00:00:00	jquery	2119
1331	2019-08-01 00:00:00	php	4748
1332	2019-08-01 00:00:00	python	13024

3.6 Total Tags and dplyr Intro

We've explored the data a little bit, but there are important insights in summarizing that data into totals. We do that by using `dplyr`, part of the `tidyverse`. It is a package that allows for verb-driven data manipulation. Basically, that means that there are slightly more intuitive names for stuff you can do to your data. In the below example, we care about collapsing over `Month` and gathering up the total number of questions with the given tag, across all time values.

We still want to separate the top 10 tags from one another. Let's look at what happens if we don't include `group_by(TagName)` in our code:

```
not_so_helpful <- data %>%
  summarise(sum=sum(X))
```

```
head(not_so_helpful)
```

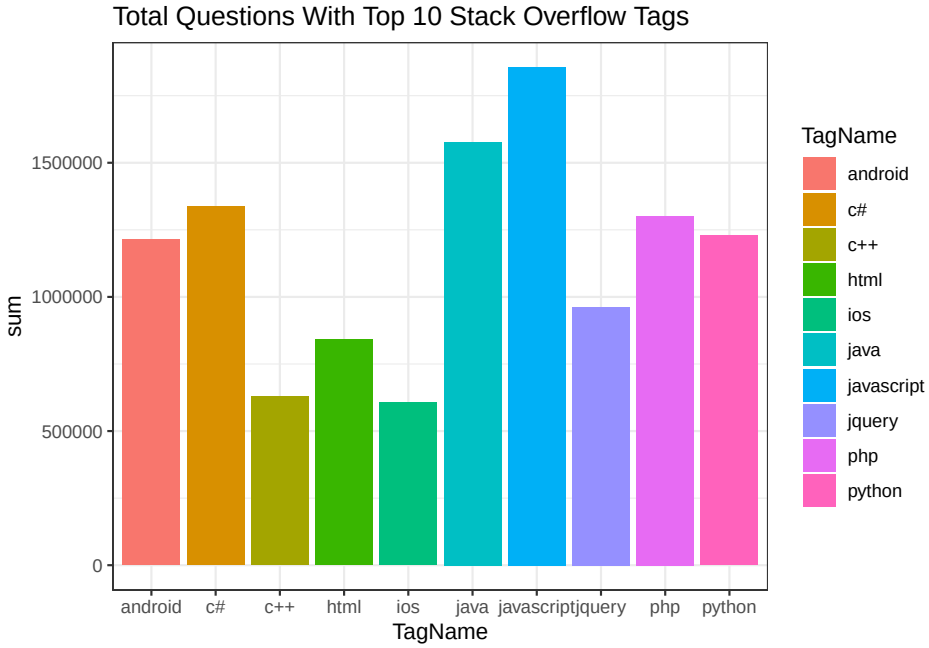
```
##        sum
## 1 11555029
```

It actually only gives us one number back. This is the total number of questions collected in our dataset as a whole. This is across tags *and* Month. But we would still like to know about the different tags, so we use the following `dplyr` code to achieve that, including a step where we `arrange()` the data in descending order. Next we create a `ggplot` bar plot to show the total questions with the top 10 Stack Overflow tags.

```
#summarizing up the totals, collapsing over Month
summary <- data %>%
  group_by(TagName) %>%
  summarise(sum=sum(X))%>%
  arrange(desc(sum))

kable(summary)%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

TagName	sum
javascript	1856183
java	1577823
c#	1336488
php	1301057
python	1228792
android	1214594
jquery	962366
html	842133
c++	629479
ios	606114

```
ggplot(summary,aes(TagName,sum,fill=TagName))+
  geom_bar(stat="identity")+
  ggtitle("Total Questions With Top 10 Stack Overflow Tags")+
  theme_bw()
```

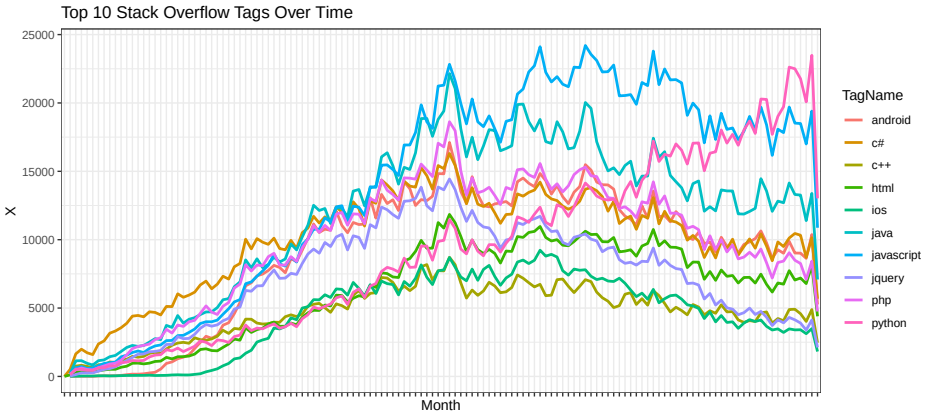


3.7 Tags Over Time By Month

Let's reintroduce the `Month` variable and take a look at how the tags differ from month to month, over time. Note, this is *not* total over time, but the number of questions using that tag on that given day. Therefore, the curves will not be smooth or always increasing.

NOTE: oh crap are we just getting the data for that specific day of the month but not collecting all days? this isn't total totals over time then it's a sample oops

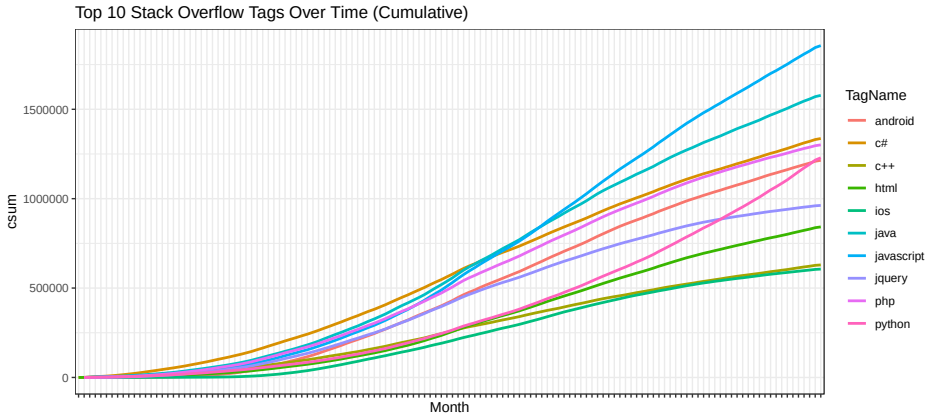
```
plt = ggplot(data,aes(Month,X,group=TagName,color=TagName))+
  geom_line(size=1)+
  ggtitle("Top 10 Stack Overflow Tags Over Time")+
  theme_bw()+
  theme(axis.text.x=element_blank())
plt
```



3.8 Cumulative Growth Over Time

If we want to get the cumulative sum, we will use `dplyr` again with the `mutate()` command. `mutate()` creates a new column in the dataframe. The `cumsum()` command refers to **cumulative sum**, similar to a `+=` in a typical for loop. The last piece of this code is to order the dataframe by the Month. You'll note that these curves are much smoother than the month to month fluctuations, and that's because a cumulative sum guarantees that kind of smoothness (and no decreasing is possible).

```
# total growth over time, holding on to the cumulative sum
cumulative <- data %>%
  group_by(TagName) %>%
  mutate(csum=cumsum(X)) %>%
  arrange(Month)

plt = ggplot(cumulative,aes(Month,csum,group=TagName,color=TagName))+
  geom_line(size=1)+
  ggtitle("Top 10 Stack Overflow Tags Over Time (Cumulative)")+
  theme_bw()+
  theme(axis.text.x=element_blank())
plt
```



3.9 Curiouser and Curiouser...

What we see above is cumulative growth curves for each of the top 10 tags on Stack Overflow. Each one seems to follow a similar trend, though with different magnitudes. All except one... the `python` tag. While the other growth curves appear sigmoidal [LINK TO GLOSS], the `python` growth looks like it isn't reaching a plateau in the time period we are looking at. It is more likely that `python` is still an actively growing area for questions on Stack Overflow, in a different manner than the other tags. In my own mind, it's easy to look at this plot and think of `python` as a car trying to rapidly cut across the highway as it changes multiple lanes, while everyone else is trying to mind their own business.

3.10 Time vs. Totals

If we look at the summary, we see that `python` and `php` have similar total tagged questions:

```
kable(summary %>% filter(TagName=="php" | TagName == "python")) %>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

TagName	sum
php	1301057
python	1228792

But there is more *information* in looking at the growth over time. Despite the `python` tag being similar to, and actually less than, the `php` tag, predictive models might show that `python` is growing much faster than `php`. If Stack Overflow were to choose which tags to pay more attention to, in order to help most of its users, which one do you think they ought to choose?

3.11 Kolmogorov-Smirnov what?!

Despite the name of this test sounding like a strong drink, it's a statistical method tracing back to the 1930s that compares two distributions from possibly different samples. Basically, it is what we can use to compare *curves*. We look at the difference between the curves, and test if they come from similar data-producing processes.

Let's set up our **null** and **alternative hypotheses**.

> Null Hypothesis (H0): There is no significant difference between these curves. They are likely the same pattern.
>
> Alternative Hypothesis (HA): There is a significant difference between these curves. They are likely different patterns.

Remember, the **p-value** is telling us about the likelihood of the null hypothesis. A *low p-value* means that the null hypothesis is not likely, and we favor the alternative hypothesis. A *high p-value* means that the null hypothesis is likely, and we fail to reject the null hypothesis. If the p-value is less than .05, we can reject the null hypothesis.

```
ks.test(cumulative$csum[cumulative$TagName=='python'],cumulative$csum[cumulative$TagNa
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  cumulative$csum[cumulative$TagName == "python"] and cumulative$csum[cumulati
## D = 0.18045, p-value = 0.02631
## alternative hypothesis: two-sided
```

In contrast, here is the test between two curves that look roughly similar to the eye. The probability that they are the same is very high.

```
ks.test(cumulative$csum[cumulative$TagName=='c#'],cumulative$csum[cumulative$TagName==
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  cumulative$csum[cumulative$TagName == "c#"] and cumulative$csum[cumulative$Ta
## D = 0.061441, p-value = 0.9627
## alternative hypothesis: two-sided
```

We don't have a causal explanation of what is happening with Python, but we can say that something significantly different is occurring with the Python tag than the PHP tag. This is just one example of something you can test on this data.

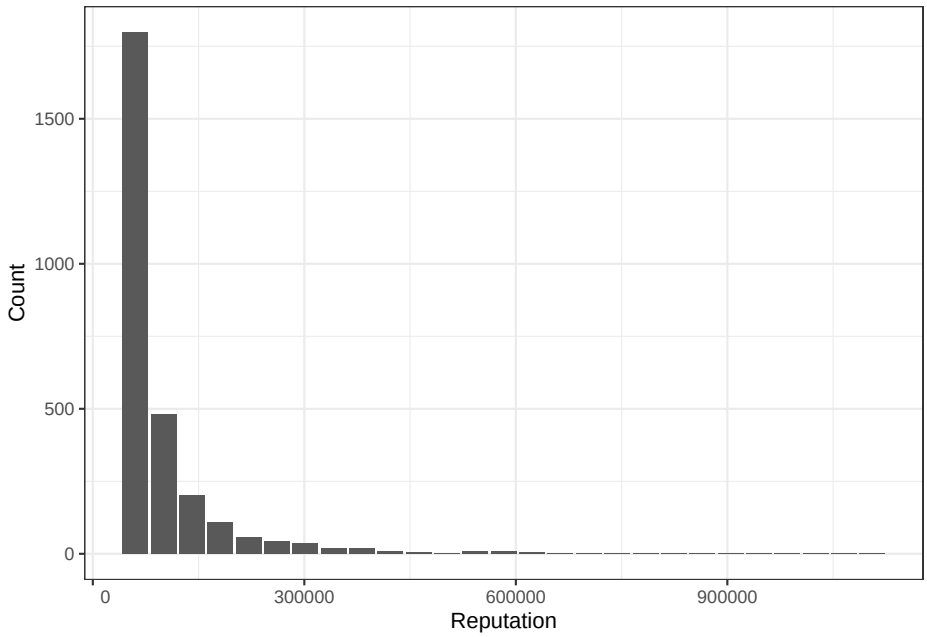
3.12 Reputation

You can read some official rules on reputation here. Basically, you are aiming for a higher reputation amount (it's not a *ranking*, it's just sheer volume of reputation points). The bigger the better. You gain reputation points by answering questions correctly and asking good questions. The reputation data comes from a query to the Stack Exchange, but the downloaded csv file is in `data/stackoverflow`. This data has the number of users with a given reputation (excluding very low reputation amounts because there are far too many users with hardly any reputation, and it would make it difficult to visualize the other reputation values).

```
reps <- read.csv("data/stackoverflow/reputation.csv")
kable(head(reps))%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

Reputation	Count
60300	1798
100500	482
140700	203
180900	107
221100	57
261300	44

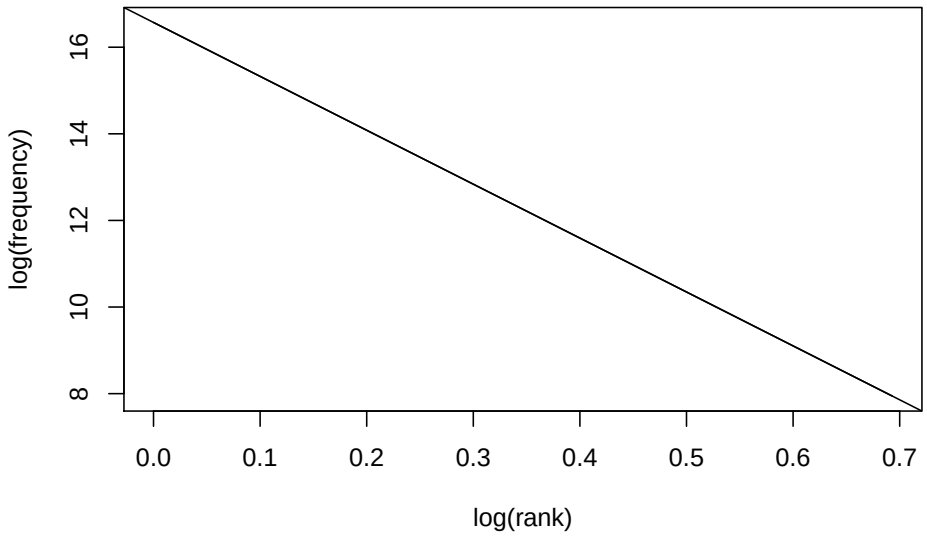
```
plt = ggplot(reps,aes(Reputation,Count))+
  geom_bar(stat="identity")+
  theme_bw()+
  scale_x_continuous(labels = function(x) format(x, scientific = FALSE))
plt
```



3.13 Log-Log Plots and Power Laws, Oh My!

We can see from the above histogram that many people have low reputation, and very few people have high reputation. This is expected, as high reputation is difficult to get and is probably only obtained by a rare few who answer questions all day. But it isn't just decreasing at one rate; it appears to be exponentially decreasing. If you plot this data on a **log-log scale**, it would appear to be a decreasing, straight line. Below is an example of a **Zipf plot**, or **power law** which shows this relationship. The super funky awesome interesting part is that Zipf's law occurs all over the place in natural settings. The English language is Zipfianly ditributed, and we see this kind of power-law dropoff in wealth disparities as well. It's interesting to find it in our own developer backyard, Stack Overflow.

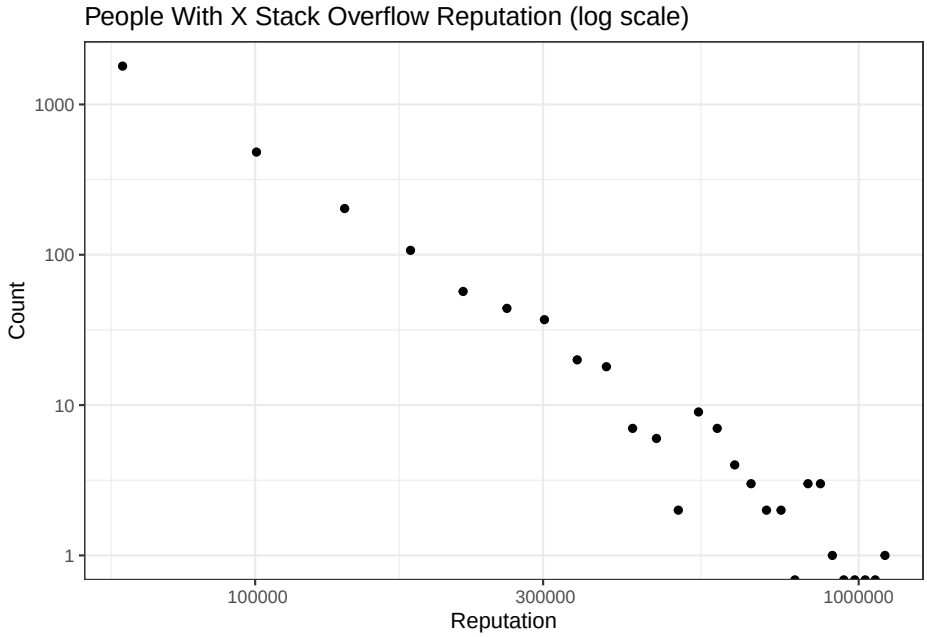
```
Zipf_plot(reps)
```




```
## (Intercept)           x
##    16.57161   -12.44834
```

```
plt = ggplot(reps,aes(Reputation,Count))+
  geom_point()+
  ggtitle("People With X Stack Overflow Reputation (log scale)")+
  theme_bw()+
  scale_x_continuous(trans='log10',labels = function(x) format(x, scientific = FALSE))+
  scale_y_continuous(trans='log10')
plt
```

```
## Warning: Transformation introduced infinite values in continuous y-axis
```


People With X Stack Overflow Reputation (log scale)
Count
1000
100
10
1
100000
300000
1000000
Reputation

Chapter 4

Analyze This!

4.0.1 Before this lesson:

- Read the paper: "Analyze This!"
- Get comfortable copying and pasting from this Markdown and running in your own R file
- Group discussion about how you might use statistics in software engineering

4.1 What is "Evidence"?

You're about to learn a term that hopefully won't haunt you like it haunted me: **epistemology**. I recently spent an entire year using the term to the point of irony, after having it repeated over and over in my first-year PhD courses. Epistemology is the study of knowing. I won't go too far into this, as it's literally PhDs worth of studying, but it's "how we know what we think we know".

> the theory of knowledge, especially with regard to its methods, validity, and scope. Epistemology is the investigation of what distinguishes justified belief from opinion

Miraculously, we can go through years of school and never question how we know anything. We learn from history books, practice math formulas, create artwork, and maybe conduct some neat experiments. All of that knowledge had to come from *somewhere*, and some of it is wrong.

Part of being a digital citizen, software engineer, and scholar is learning how to defend what you think you know. Whether that's in political arguments or convincing someone why pickles are a polarized issue, you will be caught over and over again in some form of argumentation. In software engineering, there are opinions abound. On the best language, best practice, hottest new package, best workflow, etc.

Through these lessons, you will develop your own ability to question how we know what we know; walking through how we can use data and statistics to make conclusions about the software industry. Some people believe that statistics are the ground truth, while others believe that numbers could never capture the nuance of a problem. Both of these views are dangerous. We are about to embark on a journey into Statistical Wonderland, where some things are nonsense, some things are useful, some things are wrong, and some things are awesome. The goal of this curriculum is to give you a toolbox to find your *own* answers; to learn to read and dissect academic findings and to apply the useful results to your own practice of software engineering.

4.2 Surveying a Population

Sometimes we don't know anything about anything. We all have to start somewhere. The paper "Analyze This!" approaches an unexplored research area with a certain elegance. They asked 1500 Microsoft engineers the following:

> Please list up to five questions you would like a team of data scientists who specialize in studying how software is developed to answer

They then asked a new sample of 2500 Microsoft engineers to prioritize the questions.

So yeah, they just *asked*. If you want to know what the most important things that Data Scientists should work on, *just ask*. This is an incredible, high-powered starting point for distilling how we should investigate what is most important to the stakeholders involved. Let's take a look at how this sample prioritized things, while also getting a lesson in R.

4.2.1 Loading Libraries

Libraries may also be referred to as **packages**. They are collections of **functions** that someone else has written that are not part of the base programming language, but have functions for a specific purpose. So for instance, R has a `plot` function, but we use `ggplot2` because it has better graphics and more customization. It's like an "add-on" or "expansion pack" to a programming language.

```
library(readxl)
library(ggplot2)
library(kableExtra)
library(dplyr)
```

4.2.2 Reading in the Data

Here we have an Excel spreadsheet, but many data files will be **Comma Separated Values** (`.csv`). We are using the `readxl` library to convert the spread-

sheet into a **dataframe** that R can work with. Here I have printed out the head and tail of this dataframe, but you should run the `View(data)` command in R to see the entire thing. You can run that command from the **console** after highlighting it in your R code.

```
data <- read_excel("data/145Questions.xlsx",skip=3) #skip the first three rows because they have
```

4.2.3 Renaming Columns

The columns had names that weren't as conducive to the code we will write, so here's how to rename columns in a dataframe. You do need to include all of the names using this technique.

```
colnames(data) <- c("QuestionID","Category","Question","Essential",  "Worthwhile"   ,     "Unimpor

kable(head(data[1:8])) %>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

QuestionID	Category	Question
1	Bug measurement	What is the impact and/or cost of findings bugs at a certain stage in the develo
2	Bug measurement	What kinds of mistakes do developers make in their software? Which ones are
3	Bug measurement	In what places in their software code do developers make the most mistakes?
4	Bug measurement	What kinds of mistakes are caught by static analysis?
5	Bug measurement	How many new bugs are introduced for every bug that is fixed?
6	Bug measurement	Is the number of bugs a good measure of developer effectiveness?

```
kable(tail(data[1:8])) %>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

QuestionID	Category	Question
140	Testing practices	How do we measure test coverage with unit tests?
141	Testing practices	How can we create and run unit tests whose code and test inputs can be shared
142	Testing practices	Should we do Test-Driven Development?
143	Testing practices	What are benefits of Test-Driven Development for Microsoft?
144	Testing practices	How should we do Test-Driven Development while prototyping?
145	Testing practices	When do I maintain or update a test vs. remove it?

4.2.4 Descriptive Statistics

Whenever we have data, we first report on **descriptive statistics**, which are things like the average values, the ranges of those values (highest and lowest), and other facts about the data that was collected. Descriptive statistics are contrasted with **inferential statistics**, which use statistical tests to draw conclusions about differences between groups, or fits of models, or other things that

we can use to make sense of various phenomena. Let's stick with descriptive statistics for now.

> Note that the data we have access to is already in aggregate form (we don't have access to each of the 2500 Microsoft engineer ratings, due to privacy). I will demonstrate how to get aggregate statistics across the categories with `dplyr`. This would work even if we hadn't already aggregated the data:

```
aggregate <- data %>% # this symbol is referred to as a "pipe". it "pipes" the data in
              group_by(Category) %>%
              summarise(AverageEssential = mean(Essential))

kable(aggregate) %>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

Category	AverageEssential
Bug measurement	36.28571
Customers and requirements	45.88889
Development Best Practices	24.88889
Development Practices	31.28571
Evaluating quality	33.93750
Productivity	27.92308
Reuse and Shared Components	42.66667
Services	36.62500
Software development process	30.78571
Software lifecycle; Time allocation	35.00000
Teams and collaboration	34.63636
Testing practices	26.35000

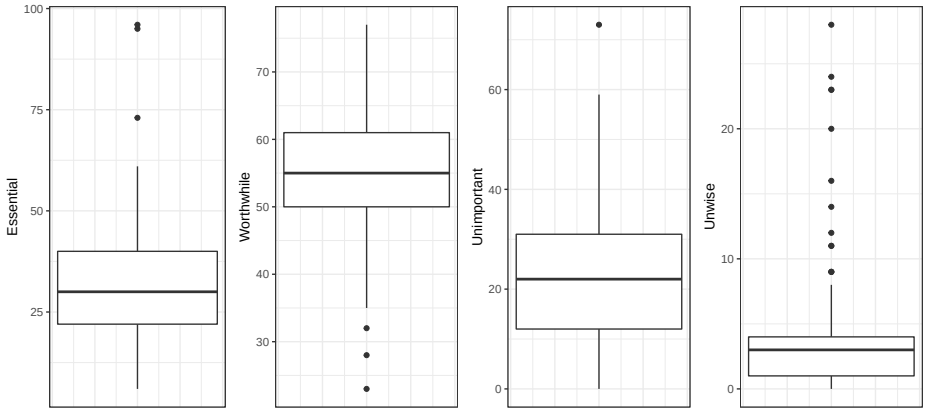
4.2.5 First Visualization

We are going to make some plots. You may or may not have learned about **box-and-whisker plots** before, but if you haven't, there are several pieces to these plots that help us visualize descriptive statistics. The bar in the middle of the box is the **mean**, and any points outside of the box-and-whisker plot are **outliers**, meaning they are significantly above or below the **inter-quartile range**. We can't actually know too much from these plots, as they simply show us, across all categories, how willing the participants were to assign a certain label to a question. We can observe that people seemed more willing to label something as "Worthwhile" than they did "Unwise" or "Essential". This actually makes sense, because it's easier to label something with a less-extreme judgment. "Unwise" is seriously suggesting that something should not be done, and "Essential" is making a serious judgment call on the value of something. "Worthwhile" is more relaxed, and it seems that more people were willing to use that label instead of strongly committing on most of the topics. It's lucky for

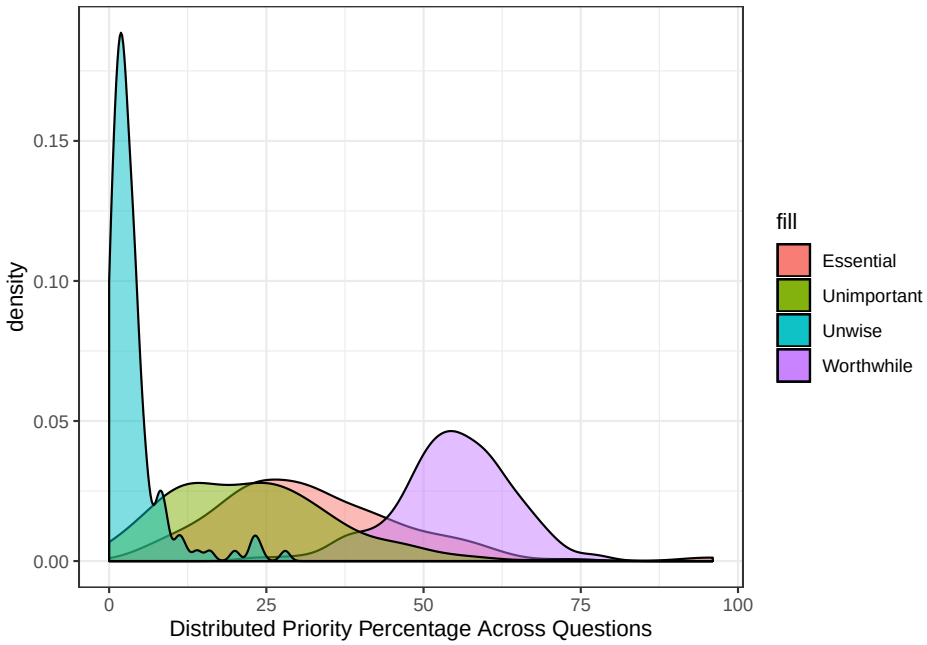
us that everything wasn't labeled "Essential" or we wouldn't even have a better idea of where to start researching.

> Note that I'm using something called `ggplot` and `cowplot` to make these graphs. `ggplot` is a cornerstone of data visualization, whereas `cowplot` simply allows me to line them up horizontally. I've used `theme(axis.text.x=element_blank()` to remove the x-axis labels. I use `theme_bw()` which stands for "Theme Black and White" because it looks better to me. You should play around with your own preferences.

```
essential <- ggplot(data,aes(y=Essential,))+
  geom_boxplot()+
  theme_bw()+
  theme(
        axis.text.x=element_blank(),
        axis.ticks.x=element_blank())

worthwhile <- ggplot(data,aes(y=Worthwhile))+
  geom_boxplot()+
  theme_bw()+
  theme(
        axis.text.x=element_blank(),
        axis.ticks.x=element_blank())

unimportant <- ggplot(data,aes(y=Unimportant))+
  geom_boxplot()+
  theme_bw()+
  theme(
        axis.text.x=element_blank(),
        axis.ticks.x=element_blank())

unwise <- ggplot(data,aes(y=Unwise))+
  geom_boxplot()+
  theme_bw()+
  theme(
        axis.text.x=element_blank(),
        axis.ticks.x=element_blank())

#lining up all the boxplots horizontally using cowplot
cowplot::plot_grid(essential, worthwhile, unimportant,unwise ,
                   ncol = 4, rel_heights = c(1, 1),
                   align = 'h', axis = 'lr')
```

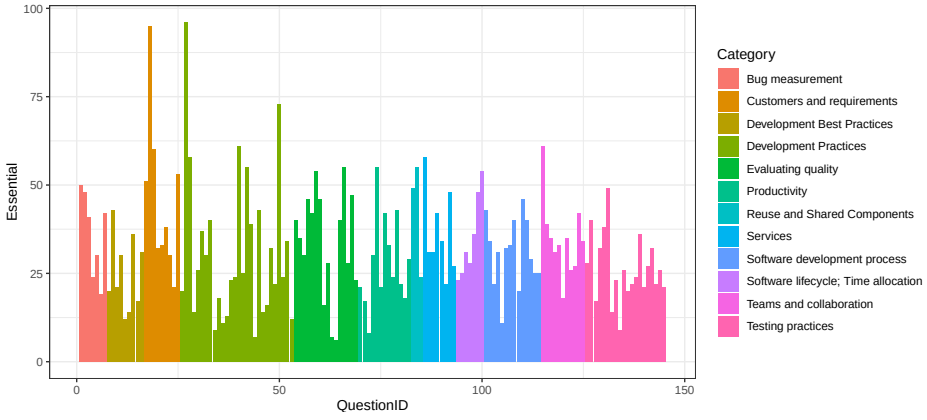



```
# How much was labeled Essential, Worthwhile, Unimportant, Unwise across the different

# FIXME: IDK what the hell I'm trying to show here
ggplot(data,aes())+
  geom_density(aes(Essential,fill="Essential"),alpha=.5)+
  geom_density(aes(Worthwhile,fill="Worthwhile"),alpha=.5)+
  geom_density(aes(Unimportant,fill="Unimportant"),alpha=.5)+
  geom_density(aes(Unwise,fill="Unwise"),alpha=.5,)+
  xlab("Distributed Priority Percentage Across Questions")+
  theme_bw()
```



4.2.7 Most Essential Question

```
ggplot(data,aes(QuestionID,Essential,fill=Category))+
  geom_bar(stat="identity")+
  theme_bw()
```

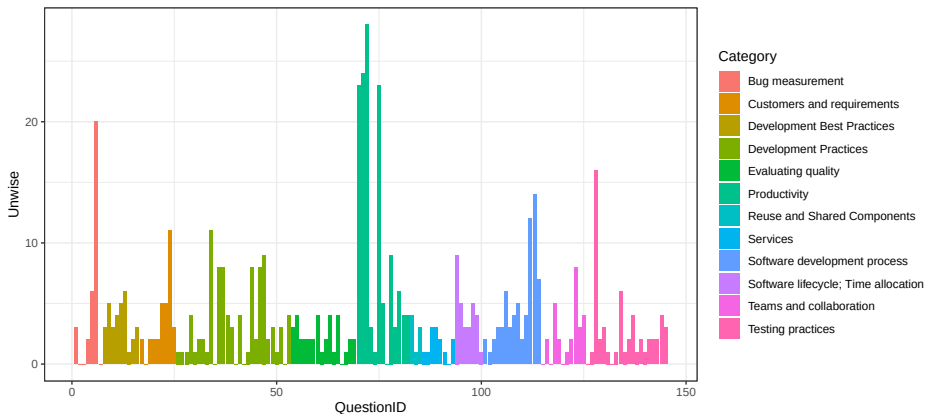



```
# FIXME: this R syntax is grabbing the Question cell wherever the Essential value is matching the
data$Question[data$Essential==max(data$Essential)]
```

```
## [1] "How do users typically use my application?"
```

4.2.8 Most Unwise Question

```
ggplot(data,aes(QuestionID,Unwise,fill=Category))+
  geom_bar(stat="identity")+
  theme_bw()
```




```
data$Question[data$Unwise==max(data$Unwise)]
```

```
## [1] "Which individual measures correlate with employee productivity (e.g., employee
```

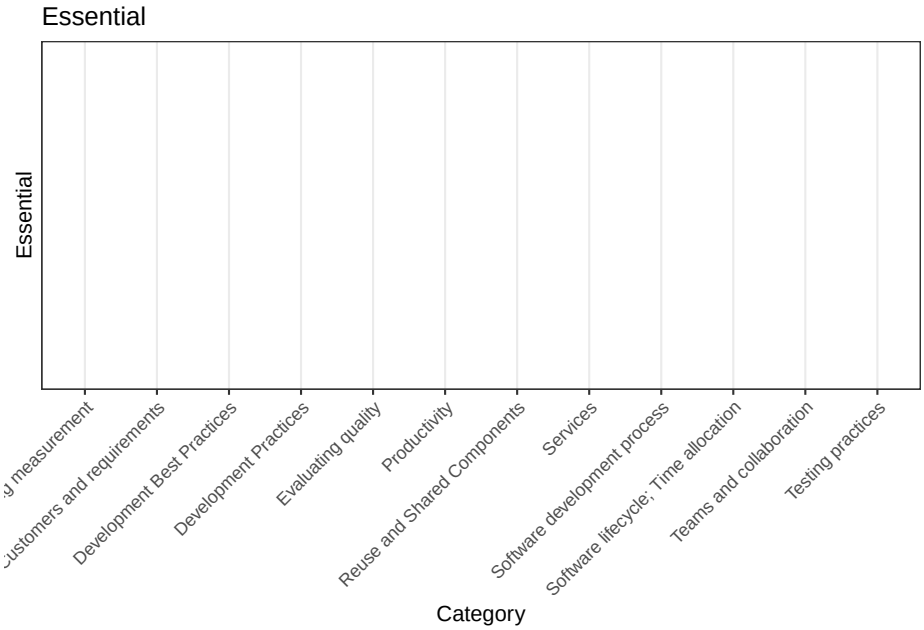
4.3 Customers Matter the Most

Specifically, people want to know about the user. Their experiences and their opinions on the tool. Do you think that academics would value this also? Or is there something about having to hit a bottom line that matters? I predict that everyone cares more about the user than we are currently catering for. In Computer Science curriculums, students are often building games or programs without ever evaluating how a user interacts with the program they've built; even though that's one of the most important parts of software engineering. Perhaps CS curriculums can also take a hint from these findings.

FIXME: I think the lesson will be stronger if we encourage readers to draw this conclusion themselves...

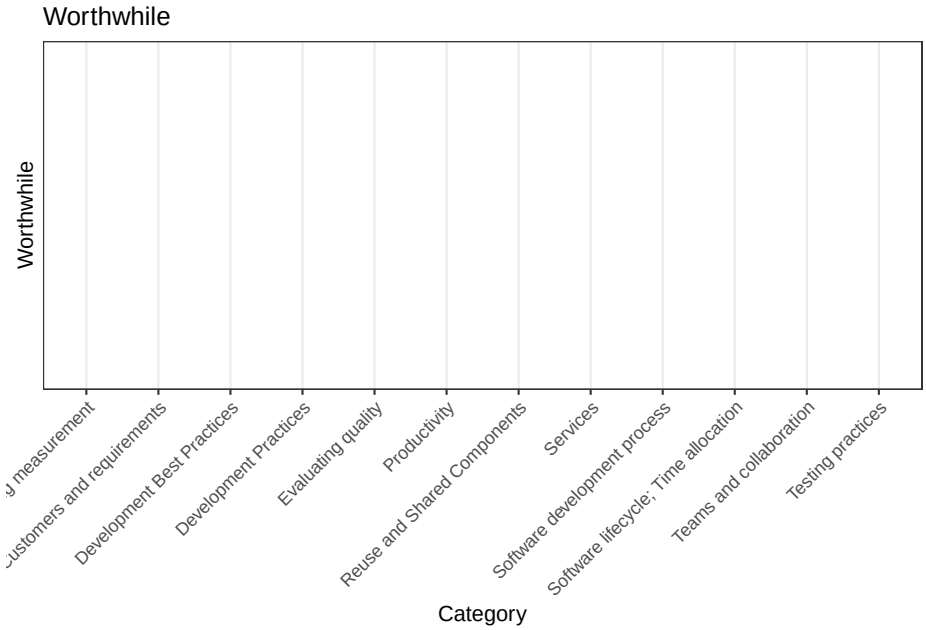
```
ggplot(data,aes(Category,Essential,color=Category))+
  stat_summary(fun.data=mean_cl_boot)+
  ggtitle("Essential")+
  theme_bw()+
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

```
## Warning: Computation failed in `stat_summary()`:
## Hmisc package required for this function
```




```
ggplot(data,aes(Category,Worthwhile,color=Category))+
  stat_summary(fun.data=mean_cl_boot)+
  ggtitle("Worthwhile")+

  theme_bw()+
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

```
## Warning: Computation failed in `stat_summary()`:
## Hmisc package required for this function
```

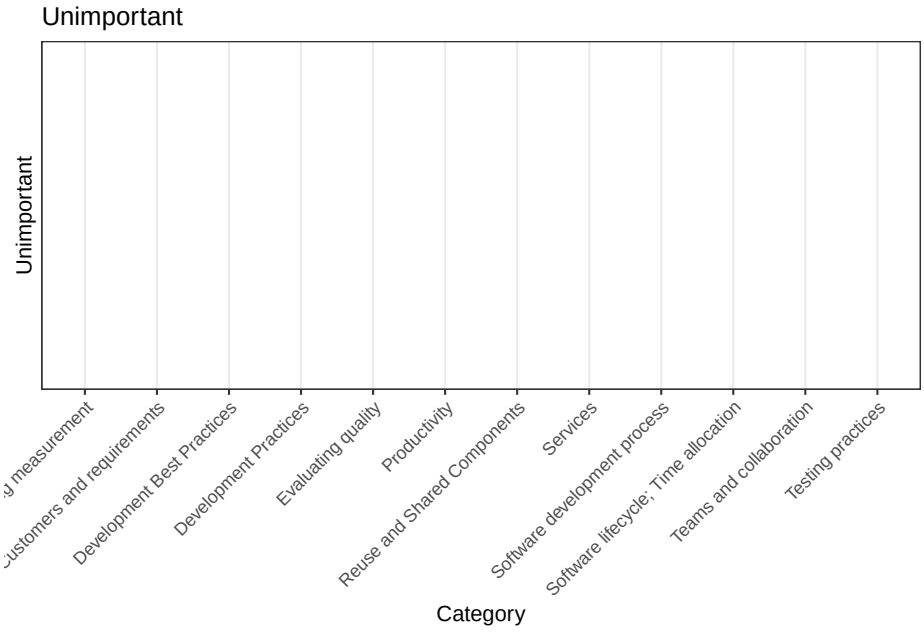


4.4 Developer Practices Don't Matter

It's actually in line with the research, there's not a ton of need to measure (for possible intervention probably), dev practice and testing. You'll see in some of the future lessons that we delve into developer folklore about the importance of different dogmas in software development, and how it doesn't matter as much as many people think it does.

```
ggplot(data,aes(Category,Unimportant,color=Category))+
  stat_summary(fun.data=mean_cl_boot)+
  ggtitle("Unimportant")+
  theme_bw()+
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

```
## Warning: Computation failed in `stat_summary()`:
## Hmisc package required for this function
```



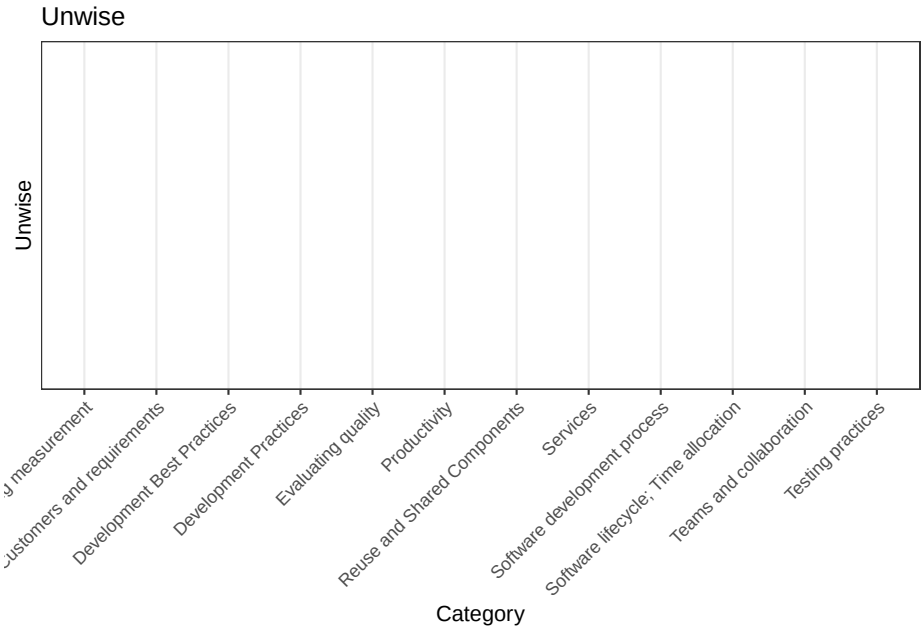
4.5 Don't Measure Productivity

As we saw from our "Most Unwise" question, we should *not* be measuring productivity. Later on, you'll see a lesson on measuring developer performance and all of the things that can go wrong. Some things should be measured, while others should be left out of the discussion. We do not need some dystopian algorithm determining if your IQ is high enough for you to do well at your job.

FIXME: again, I think the point will be stronger if we put the pieces on the table and let them decide what to eat.

```
ggplot(data,aes(Category,Unwise,color=Category))+
  stat_summary(fun.data=mean_cl_boot)+
  theme_bw()+
  ggtitle("Unwise")+
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

```
## Warning: Computation failed in `stat_summary()`:
## Hmisc package required for this function
```



4.6 How Unwise is it compared to the other categories: Test of Significance

We are moving beyond descriptive statistics to inferential statistics. Here we will use a test of significance to see if Productivity is truly more Unwise to study than the other categories. According to the above graphs, we can use visual reasoning to make a judgment about this. Does the Prouctivity mean look different than the others, on average? But sometimes it is much more subtle than that, or the confidence intervals are very large (the vertical bars radiating from the mean points). That could mean that the data is really widespread, has some outliers, or doesn't have a very large sample size. We use a **Mann-Whitney U Test** to compare the two groups. We will go further into detail about that test in future lessons. But for now, know that we are comparing two groups, and want to know if they are significantly different or not.

- Unwise ratings for all of the questions in the Productivity category vs. Unwise ratings for all the questions in the other categories

Our **null hypothesis** is what we start with. We begin with the assumption that the two groups are *not* different. If the result of our statistical test is a **p-value** less than .05, it means that there is only a 5% change our null hypothesis is wrong. This would lead us to suggest that the null hypothesis does not hold, and that the groups may truly be different after all.

```
mean(data$Unwise[data$Category=='Productivity'])
```

```
## [1] 10.23077
```

```
mean(data$Unwise[data$Category!='Productivity'])
```

```
## [1] 3.204545
```

```
wilcox.test(data$Unwise[data$Category=='Productivity'],data$Unwise[data$Category!='Productivity']
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  data$Unwise[data$Category == "Productivity"] and data$Unwise[data$Category != "Producti
## W = 1269.5, p-value = 0.004025
## alternative hypothesis: true location shift is not equal to 0
```

4.7 Academic and Industry Partnership for Good Research

FIXME: I'd cut this entire section for now.

In academic research, we are reminded that our projects should be *"novel, feasible, and impactful"*.

> I often make the joke that on your first day of PhD school, you're basically just faced with an empty desk and the task of developing something brand new and brilliant that no one else has been able to do. Good luck!

It's actually much more structured, supportive, and gentle than that, but that's still the crux of it. The goal is to contribute to science with something that is new, actually possible, but also helps people (my version of impactful) and contributes to what we know. This paper points out that one of the best ways to achieve all of those is to do some of your research outside of academia. Asking industry professionals what they need is *good research*. Asking any population of interest what they need is *good practice* and needs to be taken into account. We often have opinions about what we think is important, novel, feasible, etc. But our number one step as researchers is to go find out if our "hunches" are actually founded, and if anyone else sees value in what we are envisioning. Some people will be naysayers to everything; but let's apply what we are learning about statistics to how we feel about our own ideas: if a large sample of people seem to think your work will be a good idea and will be impactful and important, (and they can't think of someone who is already doing it), then you've already started on your statistical journey of knowing something.

Chapter 5

How Many Repositories Are There on GitHub?

5.1 Have You Ever Wondered...

We come across statistical questions all the time, even if we don't immediately realize it. From "whoah, how did Amazon know I was looking for a collection of rubber ducks that look like cowboys?" to more applied questions like "When will this server run out of storage space?" or "When will this project be com-

pleted?" Fortunatley, we live in a data-driven society; with access to billions of data sources from all over the world, we should be able to answer anything, right? Data comes in all kinds of forms; heartrate data every second from your fitness watch, location check-ins from that pizza place you went to last week, transactions, clicks, views, posts, tags… It's all digital data that can tell us something about the world. But making sense of it all requires statistical thinking and data-wrangling skills.

5.2 How many repositories are there on GitHub?

Consider this question. The immediate response is to go Google it; voila! You have an answer (it seems to say that there are 100 million repositiories). But how would you actually arrive at that answer? And how would you know you were correct? Similarly to a straightforward programming task, seemingly simple questions can devolve into tons of unforseen caveats, constraints, and workarounds.

5.2.1 Our Data (GHTorrent)

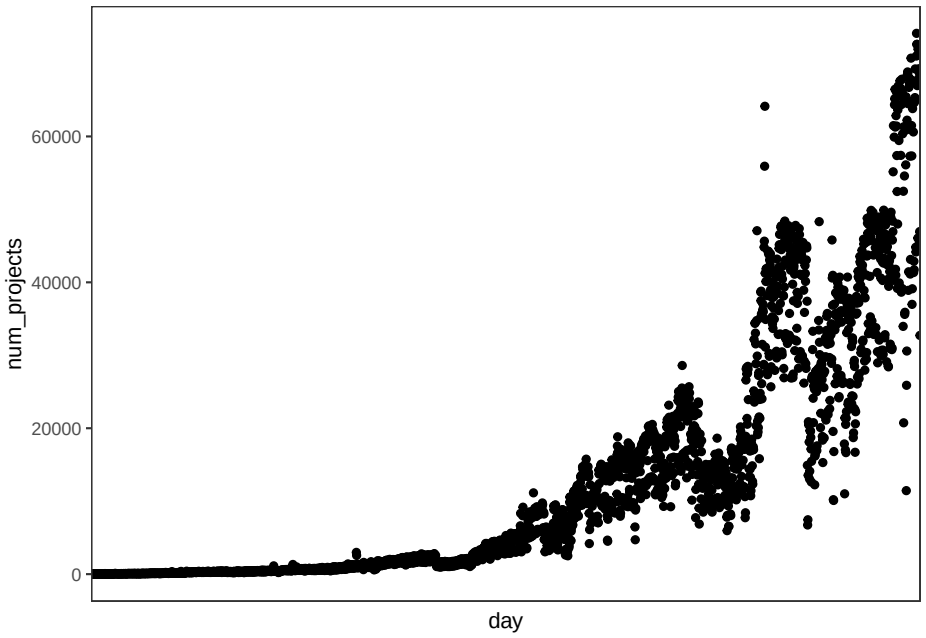
An important thing to understand early on in your programming career is about APIs and access to data. A lot of the time, you'll get data via a `.csv` or `.txt` file and can run everything on your local machine. But when working with really large datasets, there's just no way you can run it all on your own laptop. If you wanted to use a program to count the number of repositories on GitHub, you might consider downloading all of them and keeping a counter of each new one. Well, that's *terrabytes* of data, so it's a no-go. So for our question, we are using something called **GHTorrent** (Gousios, 2013). GitHub itself keeps track of all the behavior happening across the site. Commits, pushes, pull requests, new repositories, issues; and you can access those things through the GitHub API when you have a query. GHTorrent holds on to all that data, through the years. So now we can get a good picture of how fast GitHub is growing over time. Below, we query the ghtorrent project on Google Big Query, where `id` is a `project id` or **repository** with a `created_at` value. Grouping by day, we see the number of projects created each day. We also have access to other information about those repositories, like *commits, users, language, description* and more. For now, let's look at the simplest count.

5.2.2 DataFiles for this Lesson

For this lesson, we actually stored all of the queries in datafiles in this repository. Accessing the data required a credit card, which is unreasonable to expect from anyone just trying to use these lessons to learn. If you *do* want to perform different queries of your own, there are several ways to do that. I have included how you could do it using **BigQuery** and `bigrquery` (that's the R package,

see the r in there?). Any time you see `query_exec()` it will be commented out and you will use the data we have provided from the result of that query as of September 2019. That way, you can see how the query would be executed but also you don't need to fiddle with APIs to follow along in this lesson.

```
library(bigrquery)
library(ggplot2)
#- set_service_token("servicetoken.json") # you will need to get your own service token from your
#- project <- "gitstats" #your project name here


#- # an sql query string
#- sql <- "select count(id) as num_projects,date(created_at) as day
#-                 FROM [ghtorrent-bq.ght.projects]
#-                 group by day"


#- # executing the query if you aren't using data provided
#- data  <- query_exec(sql, project = project, useLegacySql = FALSE)

# reading in data for the above query that we stored earlier
data <- read.csv("data/how_many_repos/data1.csv")

data <- data[data$day!="2016-05-01",] #crazy outlier? #like unreasonably so

# number of projects created each day (it's going up, but its not the running sum! don't mix them
plt = ggplot(data,aes(day,num_projects))+
  geom_point()+
  theme_bw()+
  theme(axis.text.x=element_blank(),axis.ticks.x=element_blank())+
  theme(panel.grid.major = element_blank(), panel.grid.minor = element_blank())
plt
```




```
data <-data[order(data$day),]
data$n <-seq.int(nrow(data)) #adding in indexing for the days, so they're numeric as o

#what we care about is the number of projects total, over time
data$num_projects_sum <- cumsum(data$num_projects)


TOTAL <- sum(data$num_projects) #here's our first answer
```

5.2.3 Different Strokes for Different Folks

Could it be that depending on what we investigate and take into account, our final result changes? Does a repository count if it's not code? What if it hasn't been committed to in over a year? Do we care about forked repositories? It depends on what we care about… Do we care about *active code on GitHub*? or do we care about the sheer amount of storage space being taken up on the site? If the latter is the case, do we need information about the size of the repository; maybe even the lines of code? If we care about the former, do we need to look at which files are getting committed to, and which ones aren't being touched? Let's keep track of some of our findings in a table.

```
library(knitr)
library(kableExtra)
library(dplyr)
```

Table 5.1: different results obtained from different setups of the problem

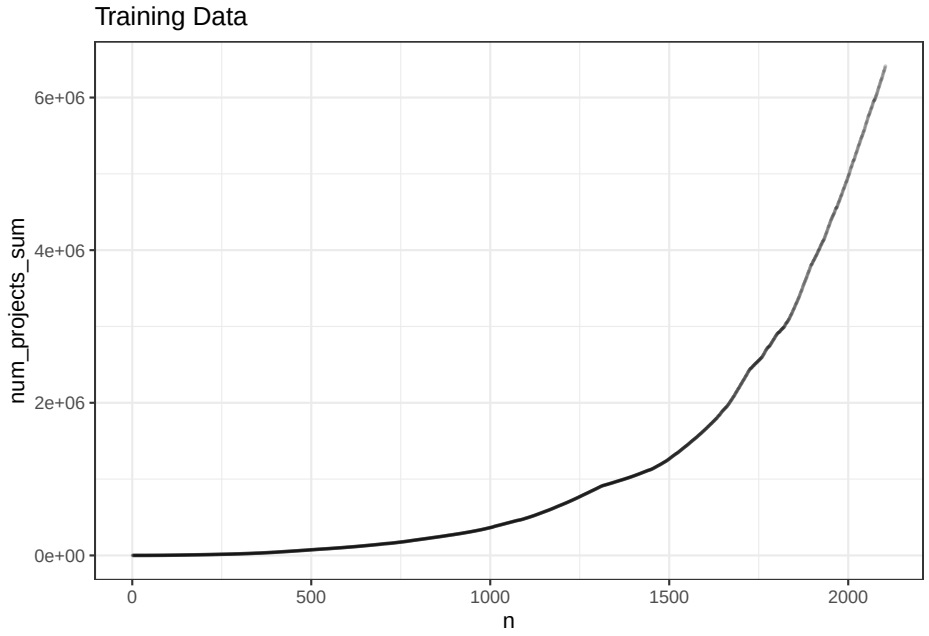
Final_Result	Description
37473073	simplest raw sum

```
Final_Result <- c(TOTAL)
Description <- c("simplest raw sum")
different_results <- data.frame(Final_Result,Description)
kable(different_results,caption="different results obtained from different setups of the problem'
  kable_styling(bootstrap_options = "striped", full_width = T)
```

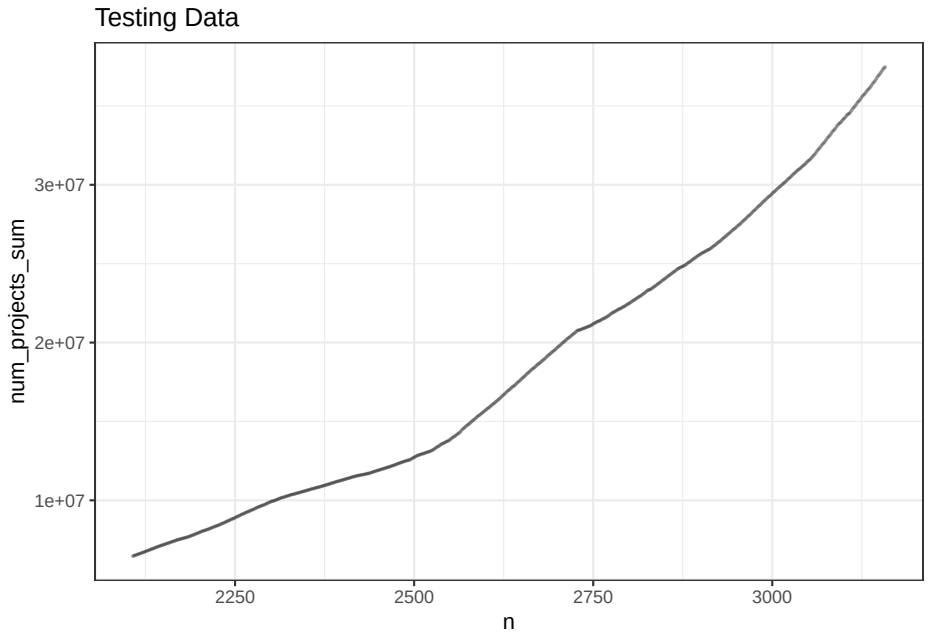
5.2.4 Training Data and Testing Data

You may have a heard about models being "trained". Sometimes this conjures up images of a baby robot learning its ABCs. But training a **model** basically means building an equation or program off of some data, so that it can reliably generalize to *new* data. We try to find an equation that "fits" the data while also being flexible enough to generalize into the future. If you've heard of the term "trend line" or "line of best fit", this is the concept we will start with when discussing training and test data. On our training data, we find a good fitting trend line. In order to evaluate how good that trend line is, we can compare how it performs on our test data. The benefit of reserving test data is that we can immediately evaluate our model, instead of simply waiting for it to be deployed in the world and either failing or succeeding (trust me, that's not a good idea). So we hold on to some data where we know the **true** values, and compare to what our trained model would predict. Below, we split our data into a training and test set, with a 70/30 split. With some data, you would sample in order to get a wide range of test data from across the entire set. However, because our model is *over time* we will reserve the last 30% of true data in order to evaluate how effective our models are. **You do not always want to just grab the last bit of data, as this can be biased for some problems where time is not the independent variable.** (Imagine you were trying to fit a model to determine the genre of a movie, and your test set was accidentally all holiday movies because you pulled from December for your test set). In our case, we will grab the most recent 30%.

```
third <- length(data$n)%/%3

test <- tail(data,third)
train <- head(data,third*2)

ggplot(train,aes(n,num_projects_sum))+
  geom_point(size=.2,alpha=.2)+
  ggtitle("Training Data")+
  theme_bw()
```



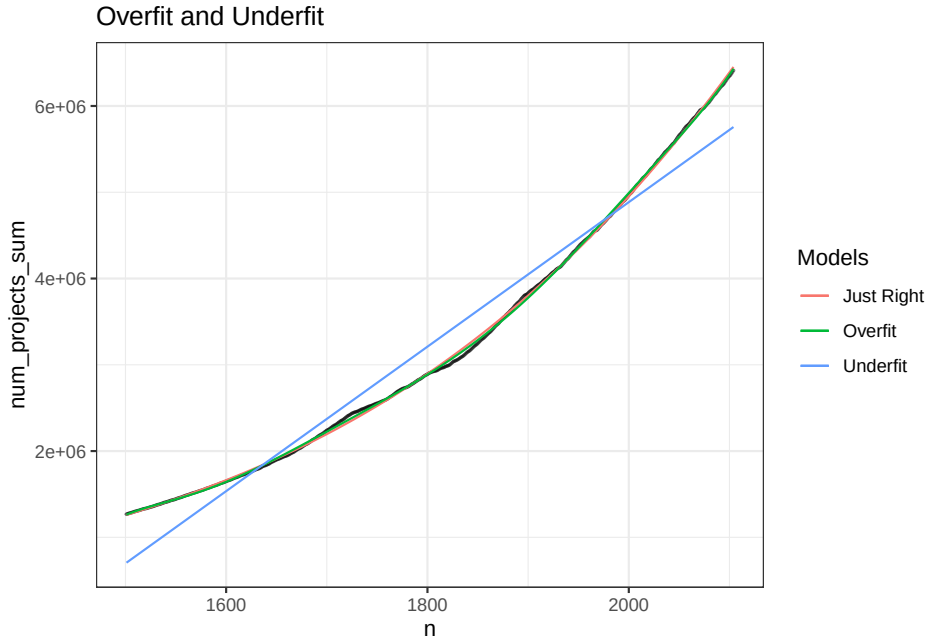

```
ggplot(test,aes(n,num_projects_sum))+
  geom_point(size=.2,alpha=.2)+
  ggtitle("Testing Data")+
  theme_bw()
```



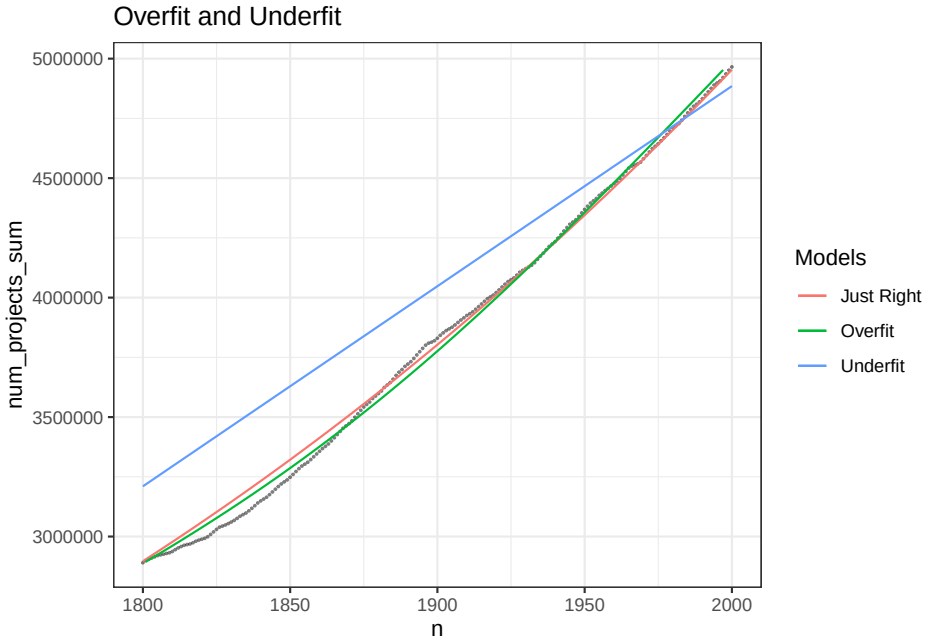
5.3 Overfitting and Underfitting

I guarantee it, you can probably fit a model to anything. It doesn't mean it's a *good* model, and it's most certainly not guaranteed to be a *useful* one. The reason why we have training and testing data is so that we can evaluate how useful our model is at predicting new information. A big danger of fitting a model to your training data is **overfitting**. All data is messy, and contains **noise**. That means, tiny dips and jumps that are just natural in the data. But we want to capture the main trends, the main patterns, in the data we have. So we want to make sure that we aren't creating models to fit those tiny dips that don't really matter. Let's zoom in on our data, and look at a *very* close fit vs. a not so great fit:

```
zoom <-1500
#zoom in on a window of our data
small <- train[train$n>zoom,]

underfit <- lm(num_projects_sum~poly(n,1,raw=TRUE),data=small) #just one parameter, it will be a
overfit <- lm(num_projects_sum~poly(n,10,raw=TRUE),data=small) #10 parameters, this will be a ver
just_right <- lm(num_projects_sum~poly(n,3,raw=TRUE),data=small) # 3 parameters, this one seems t

#generate predictions from the models
pred.underfit<- predict(underfit,data.frame(n=small$n))
pred.just_right<- predict(just_right,data.frame(n=small$n))
pred.overfit<- predict(overfit,data.frame(n=small$n))

#plot the prediction lines
ggplot(train[train$n>zoom,],aes(n,num_projects_sum))+
  geom_point(size=.2,alpha=.5)+
  geom_line(aes(n,pred.just_right,color="Just Right"))+
  geom_line(aes(n,pred.overfit,color="Overfit"))+
  geom_line(aes(n,pred.underfit,color="Underfit"))+
  scale_color_discrete(name="Models")+
  ggtitle("Overfit and Underfit")+
  theme_bw()
```




```
#zooming in even closer
ggplot(train[train$n>zoom,],aes(n,num_projects_sum))+
  geom_point(size=.2,alpha=.5)+
  geom_line(aes(n,pred.just_right,color="Just Right"))+
  geom_line(aes(n,pred.overfit,color="Overfit"))+
  geom_line(aes(n,pred.underfit,color="Underfit"))+
  scale_color_discrete(name="Models")+
  ggtitle("Overfit and Underfit")+
  theme_bw()+
  xlim(zoom+300,zoom+500)+
  ylim(train$num_projects_sum[zoom+300],train$num_projects_sum[zoom+500])
```

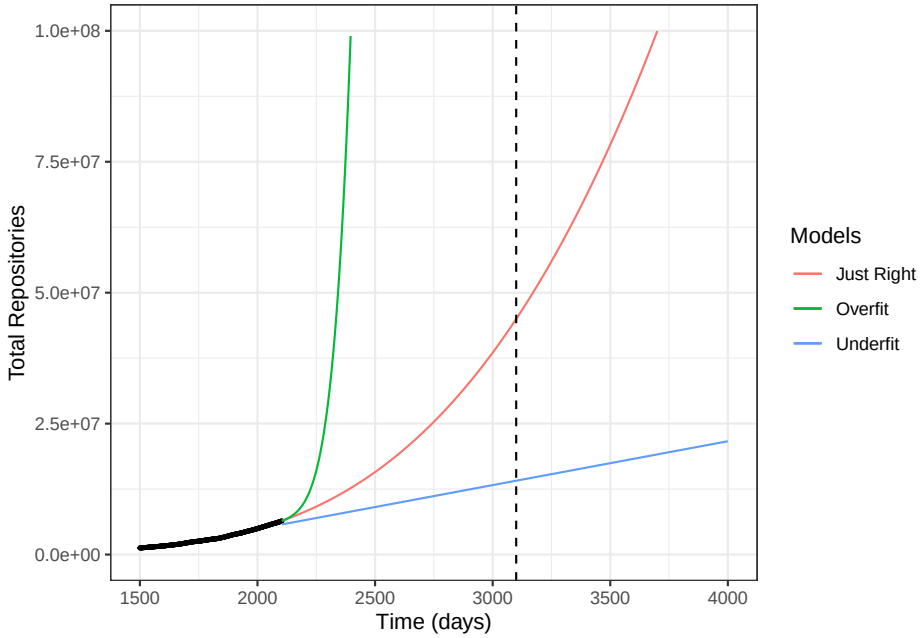


The model with more **parameters** sure does seem to fit the data best! You can imagine going to GitHub and announcing *"I've found the exact predictive model of how fast you are growing!"* Your fit is so great, that it captures the small little nuances of how the growth fluctuates. You must be so proud.

Unfortunately, let's look at the extrapolated trajectory of the models we have. Extrapolating into the future, we see that the best fitting model actually starts going in a crazy direction. It wasn't capturing the main trend, and is then subject to totally bombing on new data. **Overfitting**: using too complex a model, with too many unjustified parameters, to accidentally fit to the noise in your data instead of capturing the main trend. **Underfitting**: not complex enough of a model, missing out on valuable information about the patterns in the data (like a straight line attempting to fit an exponential). We want the "sweet spot" where we have a reasonable number of parameters, but are also robust to real world data, and can handle noisy inputs.

```
#extrapolate the model onto future data
extrap<- as.data.frame(predict(overfit,data.frame(n=seq(tail(train$n, 1),4000,1))))
extrap$n <- seq(tail(train$n, 1),4000,1)
colnames(extrap) <- c("overfit","n")
extrap$underfit <-predict(underfit,data.frame(n=seq(tail(train$n, 1),4000,1)))
extrap$justright <-predict(just_right,data.frame(n=seq(tail(train$n, 1),4000,1)))

merged <- merge(extrap,train,all=TRUE)
```

```
#plot those lines and see how well they will do on future data
plt = ggplot(merged,aes(n,num_projects_sum))+
  geom_point(size=.5,aes(color="True Data"),color="black")+
  geom_line(aes(n,justright,color="Just Right"))+
  geom_line(aes(n,overfit,color="Overfit"))+
  geom_line(aes(n,underfit,color="Underfit"))+
  theme_bw()+
  geom_vline(xintercept = 3100,linetype="dashed")+
  xlim(zoom,NA)+
  ylim(-1000,NA)+
  xlab("Time (days)")+
  ylab("Total Repositories")+
  scale_color_discrete(name = "Models")+
  ylim(NA,1.0e+8)
plt
```



Here we have the story of the Three Bears, Machine Learning style. The first equation *sort of* gets the trend (at least it's increasing?!) but really isn't capturing the nonlinearity of the data. A line can only do so much. That's an example of **underfitting**. Then, we have a 10-term polynomial that beautifully fits the data we have; too perfectly. Are *10 terms* really necessary to capture the basic trend? And beyond just being excessive, the complex model may fail miserably on future data.

5.4 Slow Down! How can we fit a curve to this?

We are trying to understand the true trend in the world (in our case, the number of repositories on GitHub and how that is changing over time). Simply understanding trends is nice, as we might be able to infer some causal factors that help a company grow, or we may be able to show that some phenomenon is, in fact, changing over the course of several years. But the real value comes in being able to reliably predict how that trend will continue in the future. The world is filled with patterns and statistical questions, from asking how many repositories there are on GitHub to predicting who needs financial aid or how many people will go vote each year.

Sometimes, we can simply *look* at the data and describe the trend line; "it's linear and decreasing", "there's very little or no correlation here", "that's exponential". But things get complicated when we care about the details (which we do!), or if we are dealing in more dimensions than we can see (whoah! don't worry we will get to that in another lesson). For now, let's talk about some of the details we might care about in *our* problem, that we need modeling to help us determine:

- can a model even capture the regularities in the data we have?
- how consistently is the number of repos growing over time?
- is it linear growth? exponential? could the growth go back down like a sine wave? maybe it will plateau?
- if we count all the different forks, is the growth different? what if we don't consider them at all?
- the exact slope of the growth, helping us predict what will happen next year

If we can fit a model, it means that we have an equation that helps us understand the true phenomenon happening in our data. It will *never* be perfect (you might have heard the term "noise" and that's basically describing the beautiful unpredictableness of life). But what we hope to achieve is an equation that gets us close enough to what is happening so that it is useful. So for our data, what we hope to have by the end of this is an equation that helps us map a line through the growth of repositories on GitHub. For any year, we can look up in our equation how close we got to the *actual* number.

So where do we start? I find it really helpful to imagine the worst model you could possibly think of, and start there. Think of this like a creative exercise; what would be the *worst* way to solve this problem? Some great terrible ideas might include:

- Always predict "7" as the answer
- Always predict the mean across all years
- Guess randomly, using different equations, until one sticks

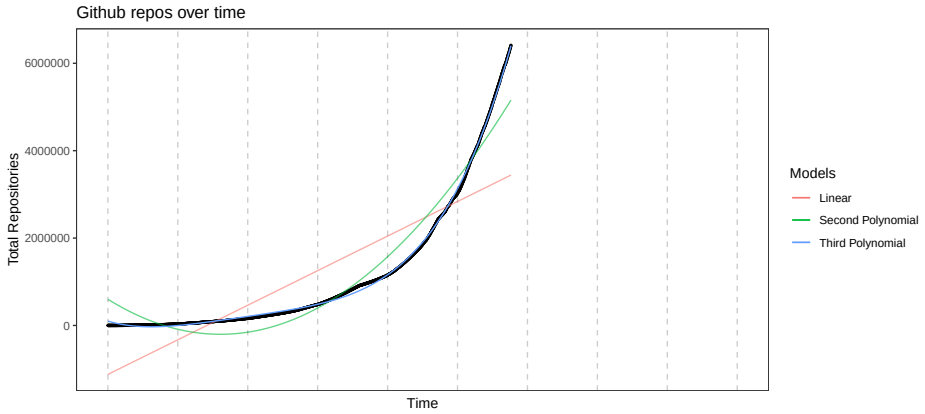
Luckily, the field of statistics and data science has come a tad further than these methodologies. So here are some guiding questions to help us determine which

model to start with.

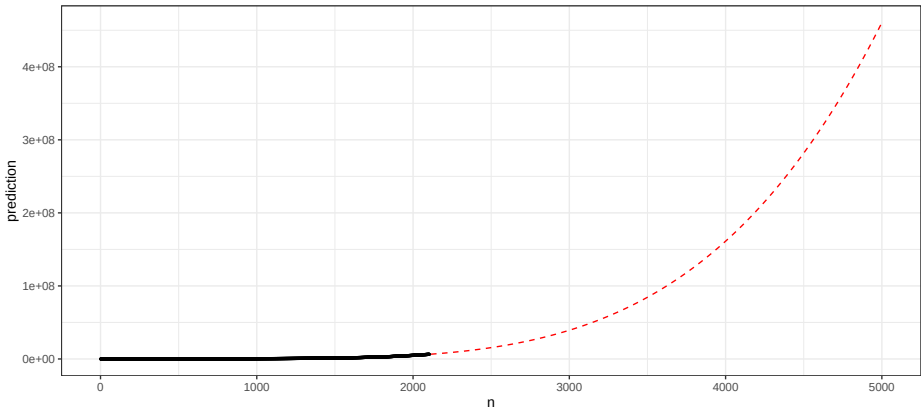
- What is a reasonable guess at the shape for this growth? Keep in mind that simpler is often better.
- What do we know about the world that makes that shape reasonable? For instance, we often see that exponential growth tends to taper off eventually, like in the case of X (example here). We also know that there really is only so much storage space, and other constraints that affect that happens in the world. As always, we don't *just* have a bunch of numbers we are looking at about GitHub repos; but a result of complex interworking parts from behavior in the real world.

5.5 What kind of curve should we fit?

```
linear_fit <- lm(num_projects_sum~n,data=train)
second_polynomial <- lm(num_projects_sum~poly(n,2,raw=TRUE),data=train)
third_polynomial <- lm(num_projects_sum~poly(n,4,raw=TRUE),data=train)



# testing out different shapes, from linear to third-order polynomial
train$linear <- predict(linear_fit,data.frame(n=train$n))
train$second_polynomial <- predict(second_polynomial, data.frame(n=train$n))
train$third_polynomial <- predict(third_polynomial, data.frame(n=train$n))



#ggplot those curves
plt <- ggplot(train,aes(n,num_projects_sum))+
  geom_point(size=.5)+
  ggtitle("Github repos over time")+
  geom_line(aes(n,linear, color="Linear"),size=.5,alpha=.6)+
  geom_line(aes(n,second_polynomial,color="Second Polynomial"),size=.5,alpha=.6)+
  geom_line(aes(n,third_polynomial,color="Third Polynomial"),size=.5)+
  geom_vline(xintercept = seq(0,3300,by=365),linetype="dashed",alpha=.2)+
  theme_bw()+
  theme(axis.text.x=element_blank(),axis.ticks.x=element_blank())+
  theme(panel.grid.major = element_blank(), panel.grid.minor = element_blank())+
  xlab("Time")+
  ylab("Total Repositories")+
  scale_y_continuous(labels = function(x) format(x, scientific = FALSE))+
  scale_color_discrete(name="Models")
plt
```

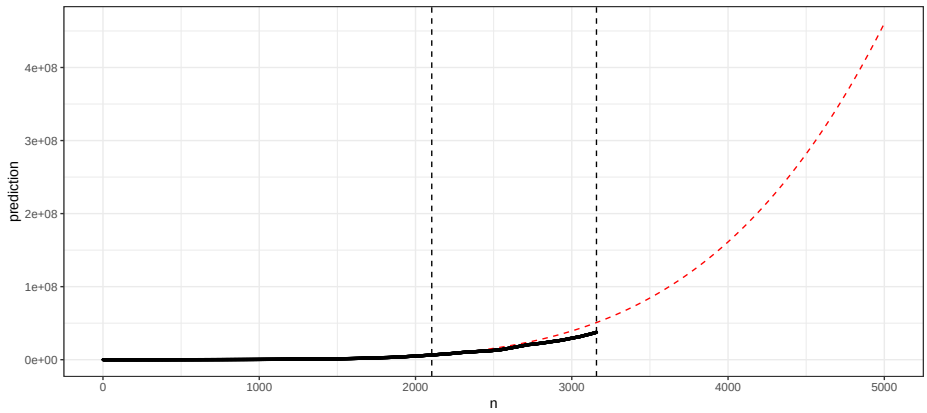



```
future <- as.data.frame(predict(third_polynomial, data.frame(n=seq(0,5000))))
future$n <- seq(0,5000)
colnames(future) <- c("prediction","n")

future <- merge(train,future,all=TRUE)

ggplot(future,aes(n,prediction))+
  geom_line(color="red",linetype="dashed")+
  geom_point(aes(n,num_projects_sum),color="black",size=.5)+
  theme_bw()
```




```
future_and_test <- merge(test,future,all=TRUE)
#predictions for testing period
library(data.table)
future_and_test<-setDT(future_and_test)[, lapply(.SD, na.omit), by = n]


ggplot(future_and_test,aes(n,prediction))+
  geom_line(color="red",linetype="dashed")+
```

```
  geom_point(aes(n,num_projects_sum),color="black",size=.5)+
  geom_vline(xintercept=tail(train$n,1),linetype="dashed")+
  geom_vline(xintercept=tail(test$n,1),linetype="dashed")+
  theme_bw()
```



5.6 Evaluation

Because we held on to some **test data**, we can evaluate how good our model is on actual prediction, as opposed to just looking at how well it fits the [**training data**](glossary.html#trainingdata] (that matters too, but not nearly as much). What we have now is two vectors that we can compare. The **true** data (our test data), and the **predicted** data (our model). We can compare how good our model did by recording the **root mean squared error**. One of the great things about R is that we can do this very straightforwardly:

```
predictions <- future_and_test %>%
               filter(n %in% (test$n))
predictions <-predictions[order(predictions$n),]
test <-test[order(test$n),]
head(test)
```

```
##      num_projects        day    n num_projects_sum
## 1749        12595 2013-10-19 2107          6459831
## 501         12179 2013-10-20 2108          6472010
## 2455        18097 2013-10-21 2109          6490107
## 925         18414 2013-10-22 2110          6508521
## 500         18160 2013-10-23 2111          6526681
## 510         18949 2013-10-24 2112          6545630
```

```
residuals <- test$num_projects_sum-predictions$prediction
head(residuals)
```

```
## [1] 11933.068  8412.371 10777.812 13428.346 15792.929 18914.514
```

```
RMSE = function(predictions, true_values){
  sqrt(mean((predictions - true_values)^2))
}

RMSE(predictions$prediction,test$num_projects_sum)
```

```
## [1] 6035449
```

Chapter 6

When Will I Be Done With This Project?

6.1 Inconsistency of Measurement

We have talked a bit about how it's important to properly **operationalize** what we **measure**. Not to add more to your plate, but even if we get the right *measurement*, and do the correct *statistics*, when you're studying people, the *reliability of measurement* also matters. This lesson will highlight both a software engineering question (predicting when a project will be done), and how to think of reliability of what you measure. After walking through the *Need for Sleep* lesson, you're surely realizing how careful we have to be with our experimental setup. You can imagine that the best way to know if something is "true" is to test a concept over and over again, under lots of different circumstances, and see if your results hold true. Imagine testing participants on multiple occasions, trying to not only test your **hypothesis**, but seeing how steady that hypothesis holds over time. All of that is ideal, of course. Most of the time, there is limited resources, limited participants, limited time, and a rush towards a deadline. But convenience should never get in the way of good science. So let's take a look at how unreliable measurements might be.

In the following study, *Inconsistency of expert judgment-based estimates of software development effort*, expert participants rated how much effort (work-hours) they estimate for sixty software projects. They unknowingly rated six of those projects *twice*, helping to answer questions about the reliability of professional judgments about software effort estimation. The paper tries to answer the following research question:

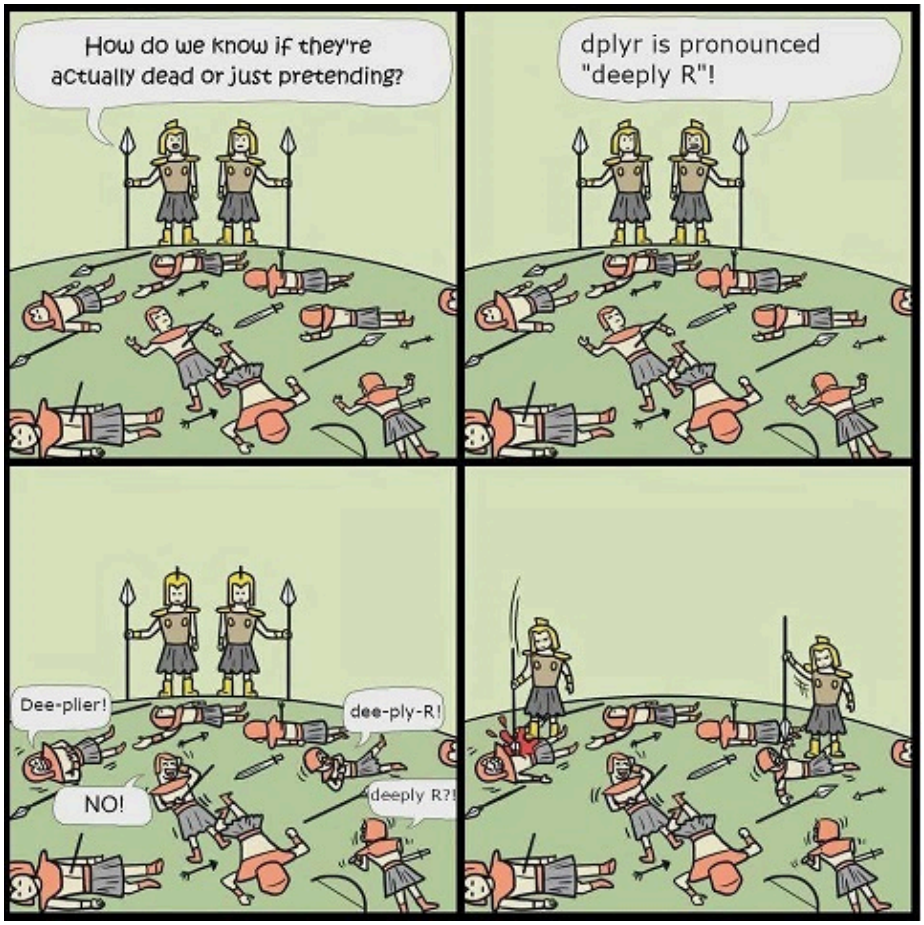
> *How consistent are software professionals' expert judgment-based effort estimates?*

This lesson is going to explore how we estimate how long software will take to finish (or your homework, for that matter). We will explore several statistical and scientific concepts:

- reliability of measurement
- generalization of results
- features that could matter
- data wrangling
- accuracy

```
source("data/ESEUR_config.r") # FIXME

library("plyr")
library("dplyr")
library("reshape2")
library("ggplot2")
```

6.3 Data Wrangling



Whatever your problem, you will come across the need to wrangle your data into different shapes. At first, it can seem either pointless or too complicated to be worth it. I remember first hearing the terms "melt" or "longform data" and wondering if I could just avoid ever doing that. Turns out, everyone was right and it's way better to wrap your head around a good `melt` command and move on with your life. Let's get a quick briefing on **long form**,**wide form**, **reshaping**, and **melting** data.

6.3.1 Original Data

```
incon=read.csv(paste0(ESEUR_dir, "data/estimation/inconsistency_est.csv.xz"), as.is=TRUE)
```

6.3.2 Wide Form

This data format "stretches" the results by each Task across several columns. Each task has its own column. I've even demonstrated the *even wider* version of the data, where the **Order** variable is also "stretched" across the columns, instead of being in long form. What we are left with is simply the Subject ID number and then a separate column for each subject's ratings for each Task in each Order.

```
head(incon)
```

```
##   Order Subject T1  T2 T3  T4 T5 T6
## 1     1       1 32 8.0 32 4.0 16  6
## 2     2       1 30 8.0 28 4.0 40 10
## 3     1       2  6 6.0  7 1.0 10  4
## 4     2       2 13 2.5 11 2.5 15  3
## 5     1       3  5 5.0  4 2.0  6  1
## 6     2       3  5 2.0  5 2.0 16  3
```

```
wider <-reshape(incon, direction="wide", idvar=c("Subject"), timevar="Order")

head(wider)
```

```
##    Subject T1.1 T2.1 T3.1 T4.1 T5.1 T6.1 T1.2 T2.2 T3.2 T4.2 T5.2 T6.2
## 1        1 32.0    8   32    4   16  6.0   30  8.0   28  4.0 40.0 10.0
## 3        2  6.0    6    7    1   10  4.0   13  2.5   11  2.5 15.0  3.0
## 5        3  5.0    5    4    2    6  1.0    5  2.0    5  2.0 16.0  3.0
## 7        4  7.5    5    7    2    7  1.5    7  6.0    4  4.0  5.5  1.5
## 9        5  8.0    4   16    3   80  1.0   25  6.0    8  2.0 30.0  1.0
## 11       6  7.0    6    7    2   40  3.0   20  8.0   10  1.0 40.0  1.0
```

6.3.3 Melt

Personally, I never understood the "melt" terminology but basically we will take the wide form data and transform it to long form. I guess it kind of goes from being stretched to dripping down and melting together? Whatever helps you to think about it, here's an example of "melting" down `tasks`.

```
tasks=melt(incon, measure.vars=paste0("T", 1:6),
        variable.name="Task", value.name="Estimate")
```

6.3.4 Long Form

Here you can see that tasks has been melted down to now represent each Task (T1, T2, T3..) as a factor of `Task` in one column. You can also see a difference with `Subject`, as it has to be recorded *twice* per column, in order to demonstrate the *repeated measure* for each task.

```
head(tasks)
```

```
##   Order Subject Task Estimate
## 1     1       1   T1       32
## 2     2       1   T1       30
## 3     1       2   T1        6
## 4     2       2   T1       13
## 5     1       3   T1        5
## 6     2       3   T1        5
```

6.3.5 Reshape

Yet another reshape will allow us to manipulate the data by whichever variable we choose. In this case, I would like to plot the *First Estimate* by the *Second Estimate*. To this day, it still helps me to draw out the exact kind of dataframe I need in order to make the comparisons or plots I have visualized in my mind. I realized that I had **Order** recorded, but those two groups of **Estimate**s were not easily comparable. I needed to manipulate my data further to have a mix of long and wide form to be the most amenable for `ggplot`.

```
# reshape data
tasks <-reshape(tasks, direction="wide", idvar=c("Subject", "Task"), timevar="Order")

head(tasks)
```

```
##    Subject Task Estimate.1 Estimate.2
## 1        1   T1       32.0         30
## 3        2   T1        6.0         13
## 5        3   T1        5.0          5
## 7        4   T1        7.5          7
## 9        5   T1        8.0         25
## 11       6   T1        7.0         20
```

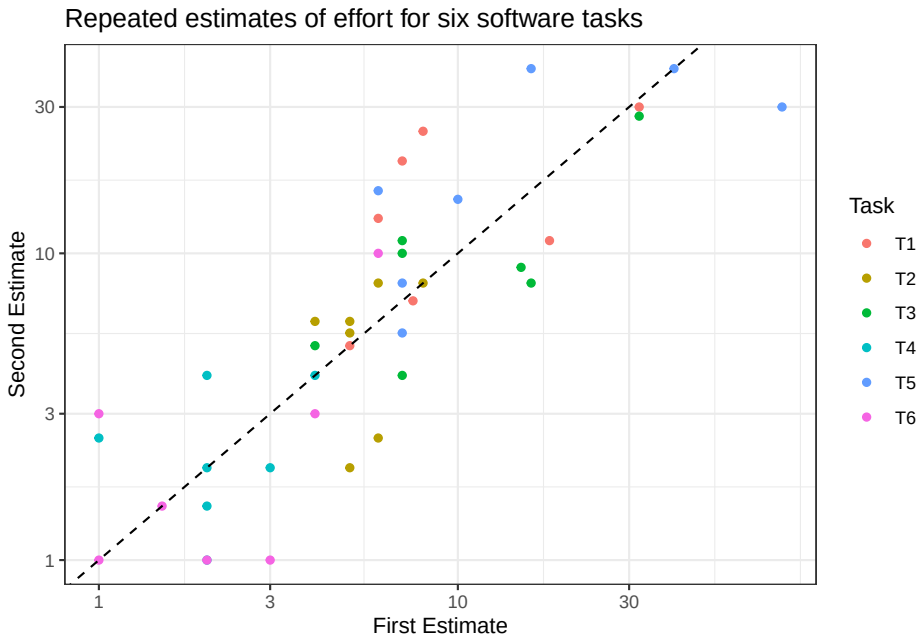
I'm truly not exaggerating when I once tried to avoid any and all reshaping/reformatting of my data. It was a difficult concept to wrap my head around, and I hoped I could just format my data in one way and stick to it. Turns out that it's much easier to commit to learning these tools and apply them to make your life easier when answering statistical questions. I'm also not exaggerating when I say I've come across almost every configuration of data you can imagine; *long form, wide form, half-melted-who-knows-what form.* If I can swallow my pride and learn a `melt` command, I believe you can too.

6.3.6 Log Plot

Here we include a **reference line**. This is the $y==x$ line, and it is describing a hypothetical scenario where each participant perfectly reliably rates each software task. On the x axis we have their first estimate for each task, and on the

y axis we see the second estimate. If they were perfectly reliable, the $y==x$ line would represent the data. If they are not perfectly reliable, the points will deviate from the $y==x$ line. The following plot also uses **log scale**, a common technique to visualize data with a spread that would otherwise be difficult to interpret. Try it without the log transformation and see which plot is more interpretable...

```
plt = ggplot(tasks,aes(Estimate.1,Estimate.2,color=Task))+
  geom_point()+
  theme_bw()+
  ggtitle("Repeated estimates of effort for six software tasks")+
  xlab("First Estimate")+
  ylab("Second Estimate")+
  geom_abline(slope=1,linetype="dashed")+
  scale_x_continuous(trans='log10') +
  scale_y_continuous(trans='log10')
plt
```



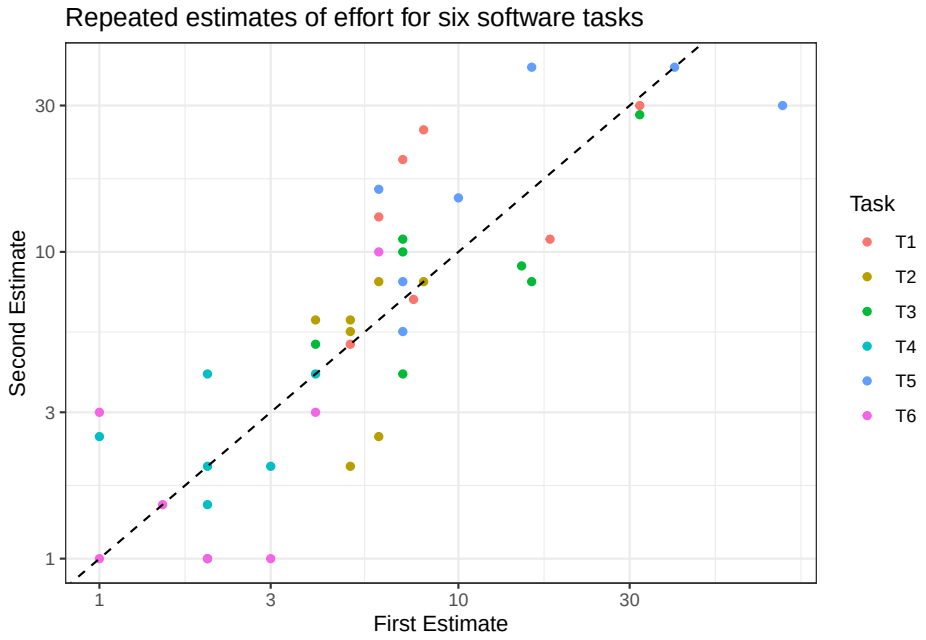

```
shapiro.test(tasks$Estimate.1)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  tasks$Estimate.1
## W = 0.55771, p-value = 4.725e-10
```

```
shapiro.test(tasks$Estimate.2)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  tasks$Estimate.2
## W = 0.77446, p-value = 1.32e-06
```

```
plt = ggplot(tasks,aes(Estimate.1,Estimate.2,color=Task))+
  geom_point()+
  theme_bw()+
  ggtitle("Repeated estimates of effort for six software tasks")+
  xlab("First Estimate")+
  ylab("Second Estimate")+
  geom_abline(slope=1,linetype="dashed")+
  scale_x_continuous(trans='log10') +
  scale_y_continuous(trans='log10')
plt
```



6.4 Correlation (Misleading)

We have explored the idea of looking at the strength of a linear relationship through (glossary.html#correlation). It might seem intuitive that correlation between the **First Estimate** and **Second Estimate** could tell us something about rater accuracy. Let's look into this correlation. First of all, we test the

normality of our responses. Neither of the distributions are normally distributed, meaning that we need to employ a rank-based (nonparametric) method when looking at correlation. Here we use a method referring to **Kendall's Tau**. Our result is `.69` which indicates a moderately strong correlation between the first and second estimate. So, they must be pretty close then, right? If they're strongly correlated, it's gotta be that they're related enough to be accurate.

```
shapiro.test(tasks$Estimate.1)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  tasks$Estimate.1
## W = 0.55771, p-value = 4.725e-10
```

```
shapiro.test(tasks$Estimate.2)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  tasks$Estimate.2
## W = 0.77446, p-value = 1.32e-06
```

```
# Correlation is a misleading method of comparing accuracy
cor.test(tasks$Estimate.1, tasks$Estimate.2,method="kendall")
```

```
## Warning in cor.test.default(tasks$Estimate.1, tasks$Estimate.2, method =
## "kendall"): Cannot compute exact p-value with ties
```

```
##
##  Kendall's rank correlation tau
##
## data:  tasks$Estimate.1 and tasks$Estimate.2
## z = 6.1993, p-value = 5.671e-10
## alternative hypothesis: true tau is not equal to 0
## sample estimates:
##      tau
## 0.69277
```

```
# Percentage difference?
# This is one of those awkward cases... TODO: switch brain
# library("boot")
```

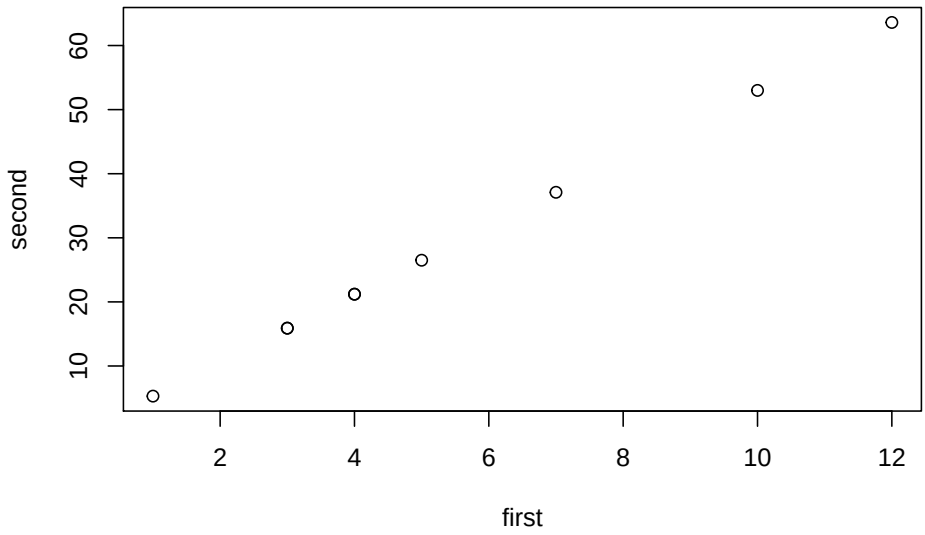
6.4.1 Don't fall for my logic!

Let's fabricate a scenario in which the two estimates are perfectly correlated but wildly different from each other. Clearly, correlation doesn't say anything about how reliable the estimates are. But it probably is saying *something* about the developer's ability to judge a project. What does a strong correlation indicate?

```
first <- c (1,4,3,7,5,4,10,3,12)

second <- first *5.3 #any scalar ensures they will be perfectly correlated

second
```

```
## [1]  5.3 21.2 15.9 37.1 26.5 21.2 53.0 15.9 63.6
```

```
plot(first,second)
```




```
cor(first,second)
```

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

```
#we could do it negatively too; the relationship would still be very strong
negative <- first *-5.3
cor(first,negative)
```

```
## [1] -1
```

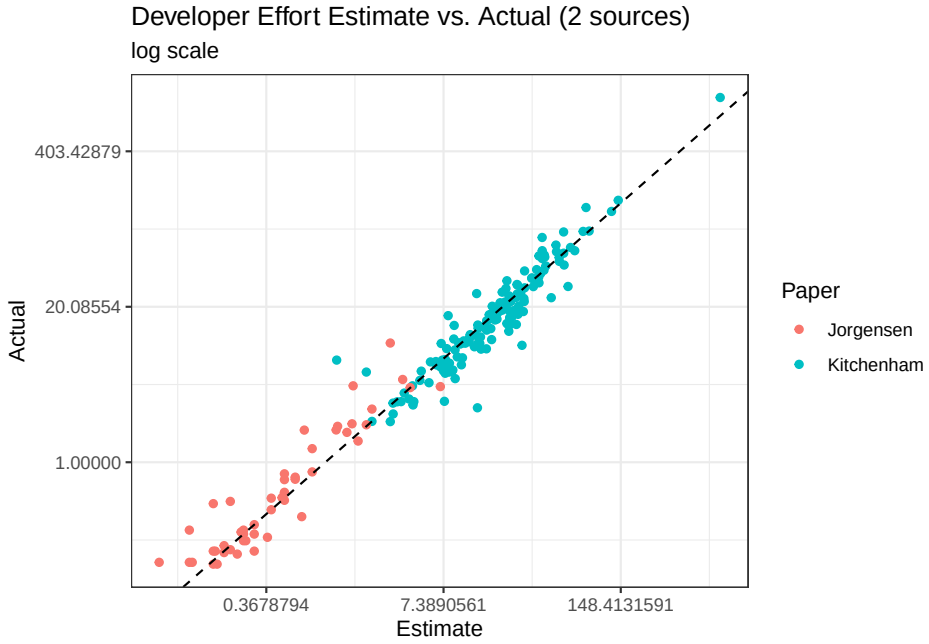
```
error <- abs(mean(first) - mean(second))
error #the error is pretty bad. on average, the second estimate was 34 hours off from
```

```
## [1] 23.41111
```

So, we've looked into *reliability* of measurement, but we still haven't compared those subjective estimates to *what actually happened*. Whenever we measure anything, we want to know if the masure was **reliable** and if the measure was **valid**. Validity of a measure means *it is actually measuring what you think it's measuring*. So, if we want to know about developer productivity, is it actually worth it to *ask* developers how long something will take? We see from the previous case that they are not reliable from estimate to estimate. Are any of their estimates actually *accurate* though? How good are devs at estimating how long a project will take? The following two studies report data on this, with develpers reporting in *hundreds of hours* how long they think a project will take to finish, against how long it actually took in reality. Both Jorgensen and Kitchenham data are plotted together. It's very clear in the software world that *underestimated* projects are far more common than *overestimated* projects, but these mishaps are also pushed around a bit by company culture.

6.5 Effort Estimation: Accuracy and Bias

```
# Regression Models of Software Development Effort Estimation Accuracy and Bias
#
# Example from:
# Empirical Software Engineering using R
# Derek M. Jones



library("foreign") #just to read in the arff file



kitch=read.arff(paste0(ESEUR_dir, "data/estimation/82507128-kitchenham.arff.txt"))
jorg=read.csv(paste0(ESEUR_dir, "data/estimation/Regression-models.csv.xz"), as.is=TRUE)


data <- kitch %>%
  select(First.estimate,Actual.effort)
data$Paper <- "Kitchenham"

jorg <- jorg %>%
  select(Estimated.effort,Actual.effort)
jorg$Paper <- "Jorgensen"
colnames(jorg) <- c("First.estimate","Actual.effort","Paper")

data<-rbind(data,jorg)

plt = ggplot(data,aes(First.estimate/100,Actual.effort/100,color=Paper))+
  scale_x_continuous(trans='log') +
  scale_y_continuous(trans='log')+
  ggtitle("Developer Effort Estimate vs. Actual (2 sources)",subtitle="log scale")+
  xlab("Estimate")+
  ylab("Actual")+
  geom_point()+
  geom_abline(slope=1,linetype="dashed")+
  theme_bw()
plt
```



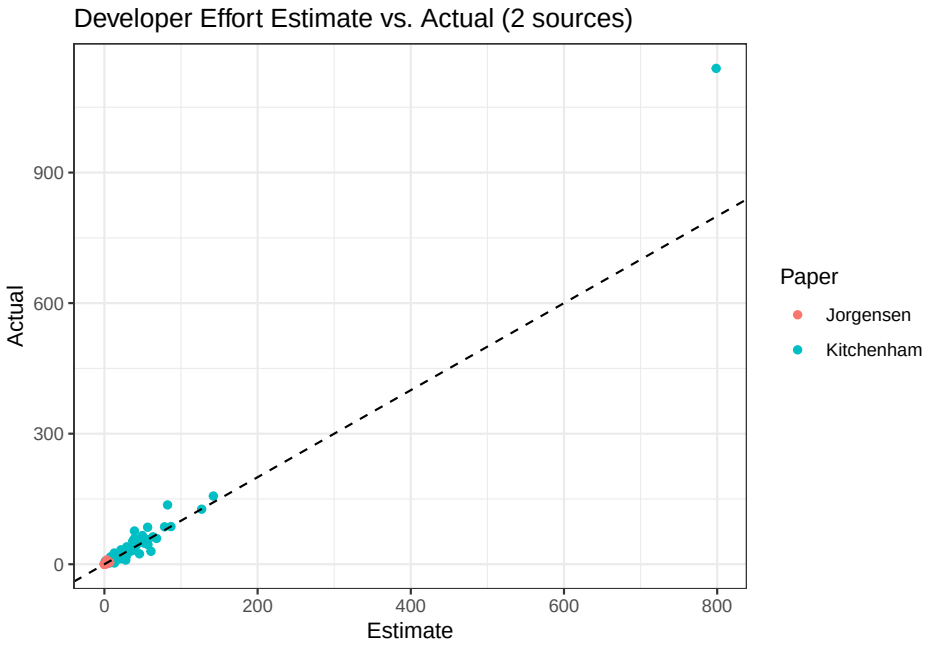
6.6 Data Visualizations: A tricky statistical tool

Remember when we took a look at the First vs. Second estimate data? We chose to plot using log/log scale, and we have just done the same for the Estimate vs. Actual data, above. Just remember that visualizations are also statistical communication tools; a plot can guide your story in ways you might not even anticipate. So for instance, in the plot above, we do see some variation around the *ideal line* (dashed), but it doesn't seem drastic. We also see that the slopes are roughly similar between the Jorgensen and Kitchenham data, just by using our eye. We can say a lot of things by simply looking at the red and blue dots and their shape, but the visualization can easily prime us to agree with one hypothesis over another. So it is always important to delve into the data, and explore it on different scales, different transformations, paying attention to some of the things we have already taught; like normality testing and curve fitting. There is nothing *wrong* with how this data has been presented, it is just important to see how the communication changes over different forms of visualization. So let's take a look at this plot *not* using a log transformation of the axes:

6.7 Without a Log Transformation

```
plt = ggplot(data,aes(First.estimate/100,Actual.effort/100,color=Paper))+
  ggtitle("Developer Effort Estimate vs. Actual (2 sources)")+
```

```
  xlab("Estimate")+
  ylab("Actual")+
  geom_point()+
  geom_abline(slope=1,linetype="dashed")+
  theme_bw()
plt
```



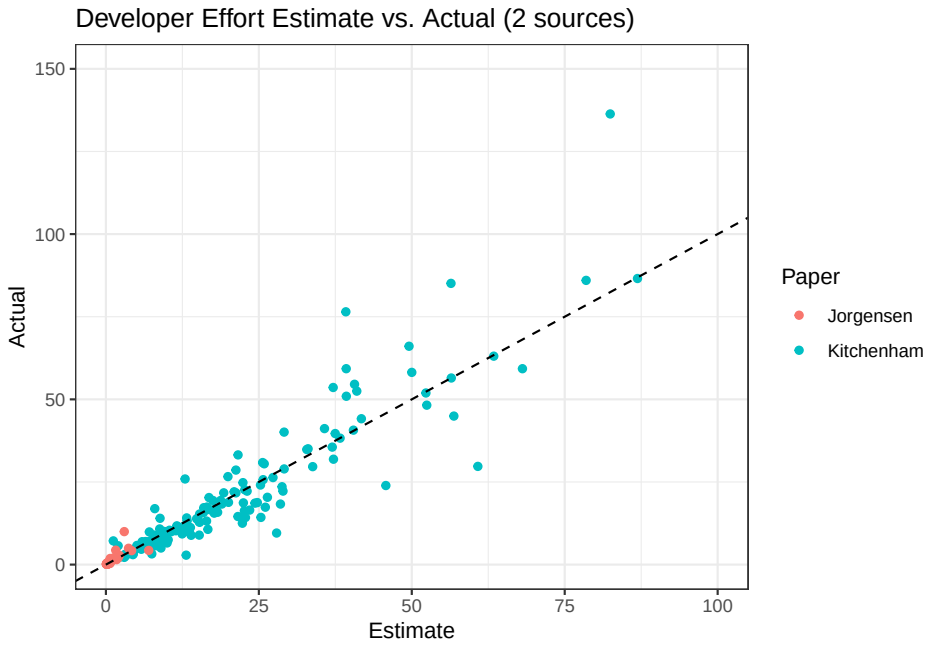
Well, there's a significant outlier that becomes much clearer in this representation. It is still visible in the original visualization, but log scale warps the *orders of magnitude*, affecting the distance between points that in raw form are actually much further from each other. It's quite difficult to see the rest of the data, and since there is no actual reason to remove that outlier (it's still a perfectly true estimate), the only way to see the rest is to "zoom in". The log transformed plot does not have this problem. Let's take a look, even closer on the raw form.

6.7.1 Zoom In

```
plt = ggplot(data,aes(First.estimate/100,Actual.effort/100,color=Paper))+
  ggtitle("Developer Effort Estimate vs. Actual (2 sources)")+
  xlab("Estimate")+
  ylab("Actual")+
  geom_point()+
  xlim(0,100)+
```

```
  ylim(0,150)+
  geom_abline(slope=1,linetype="dashed")+
  theme_bw()
plt
```

```
## Warning: Removed 3 rows containing missing values (geom_point).
```



Note how the distance from the *ideal line* is more exaggerated now. You can more clearly see that developer *estimates* differed from the *actual* hours. We have yet another layer of "orders of magnitude" that is slightly hidden from us. Remember that the measurement we are working with is *hundreds of hours*. For the actual visualizations, this doesn't matter, but for the seriousness of the actual phenomenon, **it matters**. In the scaled plot (hundreds of hours), the worst underestimate was 1.2 hours for a 7.2 hour job. 6 hours is just an afternoon, right? Well, the worst underestimate was an inital estimate of `121` hours, with an actual time of `718` hours. This is *5.9 times more than the original estimate*, and a difference of 597 hours (maybe months, depending on how many people are working). From the original plot, and from the rescaling of the points, it was not as clear to me that some projects suffered from such delusion.

Let's briefly look at the most overestimated project and the most underestimated project. We can actually use the histogram visualization to note that the maximum underestimate is actually pretty bad; an outlier from the rest of the distribution. In fact, there are quite a few outliers. A reminder that outliers are abnormal values with regards to the mean and the distribution, but should still be investigated carefully. Not every outlier can simply be thrown

out; because they may be perfectly reasonable with regards to the underlying process. In our case, there was one project that was estimated to take `79870 hours`, which then used `113930 hours` to complete. This may be an outlier from *this* data distribution, but it is not unreasonable for the actual phenomena we are looking at. I'm sure there are plenty of projects within that range, or longer. Our initial sample just wasn't covering larger projects. There is nothing inherent about that outlier that is not relevant to the process we are investigating: estimated time vs. actual time. Below, we see an investigation into the histogram of the ratios (spread), and the boxplot representation (*horizontally aligned*).

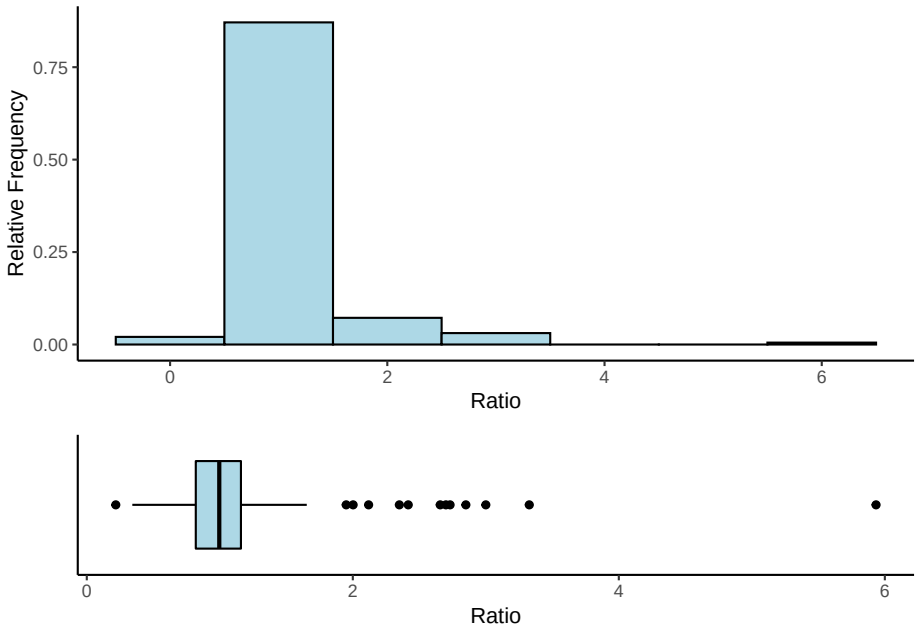
```
data <- data %>%
  mutate(Ratio= Actual.effort/First.estimate)

data[data$Ratio == max(data$Ratio),]
```

```
##    First.estimate Actual.effort      Paper    Ratio
## 69            121           718 Kitchenham 5.933884
```

```
data[data$Ratio == min(data$Ratio),]
```

```
##     First.estimate Actual.effort      Paper     Ratio
## 138           1312           286 Kitchenham 0.2179878
```

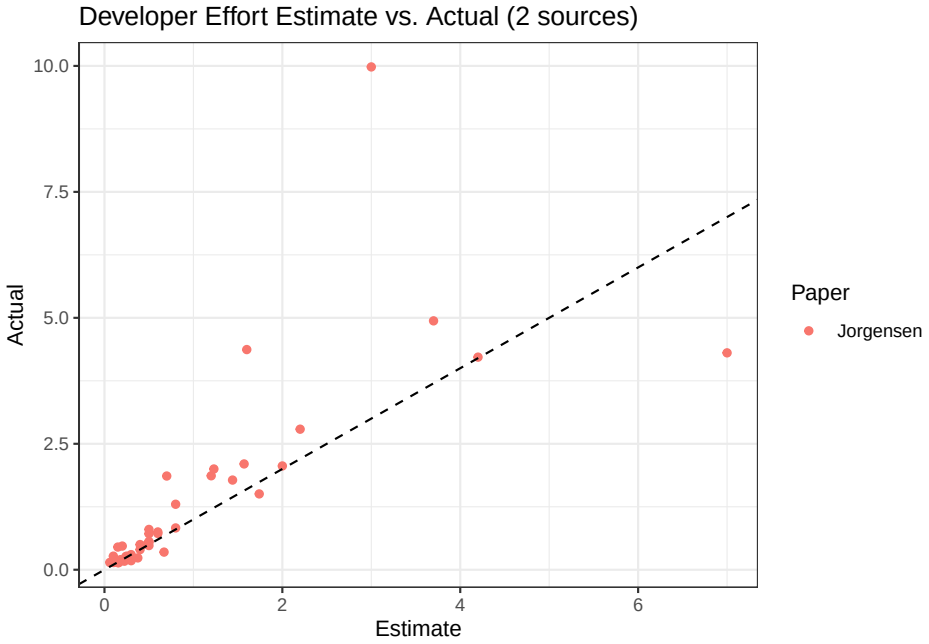
```
plt1 <- data %>% select(Ratio) %>%
  ggplot(aes(x="", y = Ratio)) +
  geom_boxplot(fill = "lightblue", color = "black") +
  coord_flip() +
  theme_classic() +
  xlab("") +
  theme(axis.text.y=element_blank(),
        axis.ticks.y=element_blank())

plt2 <- data %>% select(Ratio) %>%
  ggplot() +
  geom_histogram(aes(x = Ratio, y = (..count..)/sum(..count..)),
                 position = "identity", binwidth = 1,
                 fill = "lightblue", color = "black") +
  ylab("Relative Frequency") +
  theme_classic()



cowplot::plot_grid(plt2, plt1,
                   ncol = 1, rel_heights = c(2, 1),
                   align = 'v', axis = 'lr')
```



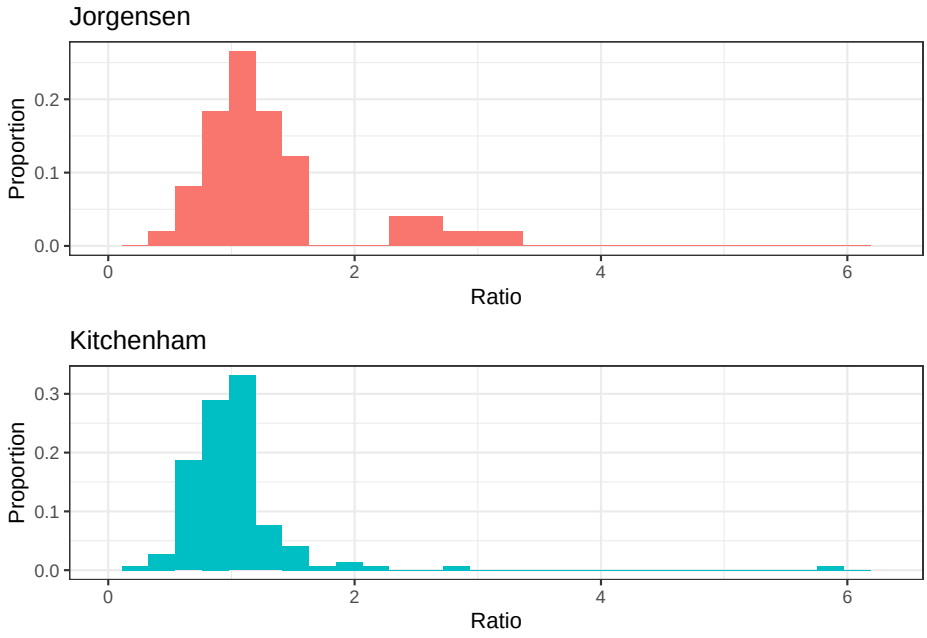
6.7.2 Zoom in on Jorgensen Data

The previous plots are still dominated by the Kitchenham data, simply because those projects were longer on average. Zooming in on just that paper, we might want to know if those two studies are actually comparable together. Judging by the histograms, after removing the outlier (just for this comparison, because we clearly know it's origins), the spread of ratios is pretty similar between the two studies, despite the studies being different scales.

```
plt = ggplot(data[data$Paper=="Jorgensen",],aes(First.estimate/100,Actual.effort/100,c
  ggtitle("Developer Effort Estimate vs. Actual (2 sources)")+
  xlab("Estimate")+
  ylab("Actual")+
  geom_point()+
  geom_abline(slope=1,linetype="dashed")+
  theme_bw()
plt
```

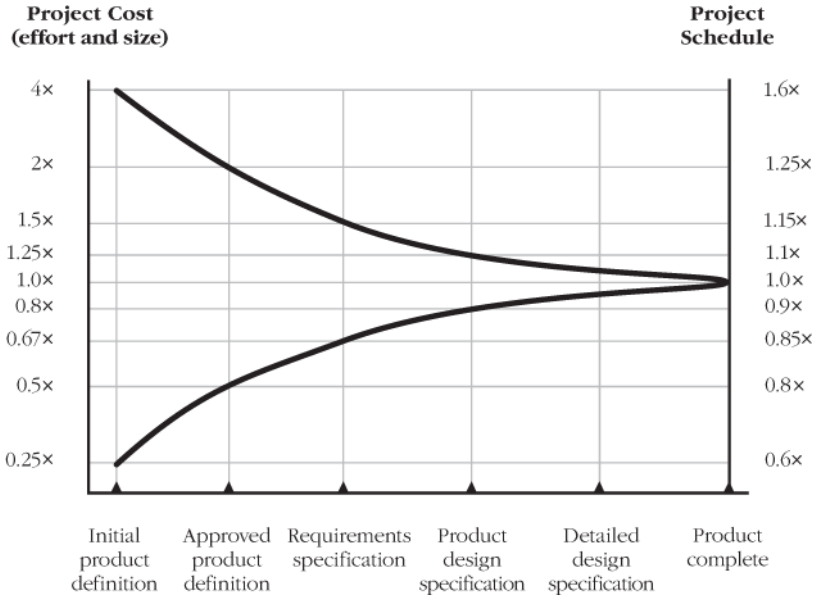



```
plt1 <- ggplot(data[data$Paper=="Jorgensen",],aes(Ratio))+
  #geom_density(alpha=.5)+
  geom_histogram(aes(y=(..count..)/sum(..count..)), position='dodge',fill="#F8766D") +
  theme_bw()+
  ggtitle("Jorgensen")+
  ylab("Proportion")+
  xlim(0,6.3)
plt2 <- ggplot(data[data$Paper=="Kitchenham",],aes(Ratio))+
  #geom_density(alpha=.5)+
  geom_histogram(aes(y=(..count..)/sum(..count..)), position='dodge',fill="#00BFC4") +
  theme_bw()+
  ggtitle("Kitchenham")+
  ylab("Proportion")+
  xlim(0,6.3)

cowplot::plot_grid(plt1, plt2,
                   ncol = 1, rel_heights = c(2, 2),
                   align = 'v', axis = 'lr')
```



6.8 So Can We Estimate Resources or Not?

What is the biggest problem with these studies? I'll give you a hint. Have you ever looked at a project overview, and said to yourself "hmm, seems simple enough" only to find yourself deep into a rabbit hole at 4am working on configuring some weird dependencies to allow you to do the nine other tasks you have to do before you can do the first thing for the project? How about the other way around; have you looked at a project spec and panicked, knowing it will take you *forever* until you realize that the package is actually pretty simple and you can probably finish this on time if the stars align just right? What I'm getting at is: *project estimation changes as you gain new information.* If I were to ask you how long something would take before you'd really started to try it, there's a good chance your estimation will be inaccurate. But what about when you've spent countless hours and are actually almost done? Will you be able to tell me how much you actually have left? Once you've worked with something for a while, you should probably get a better gauge of how long those types of problems take to complete. This concept is referred to as **the cone of uncertainty**. The idea is that as you get new information about a project, you can make more accurate judgments about the time to completion on that project. That makes some intuitive sense, but keep in mind that *by definition* you will converge to **1x** the project length by the end, because "it is what it is". Take a moment to discuss with a partner (or your cat, whatever) about this concept.



http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.630.6667&rep=rep1&type=pdf

https://github.com/Derek-Jones/ESEUR-code-data/blob/master/projects/Little-cone.R

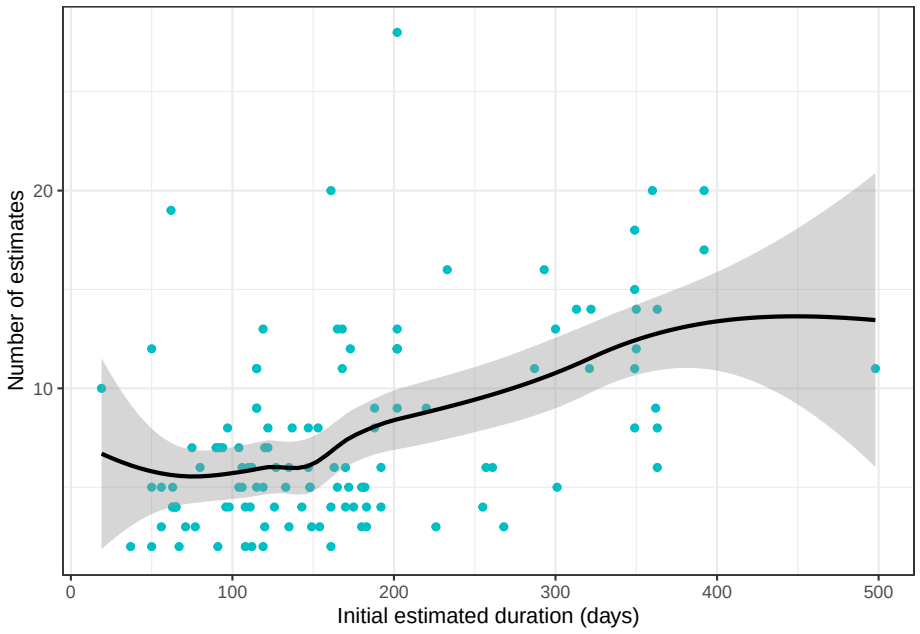
6.9 Estimate Updating

What if you were to take *multiple* estimates as the project progressed? It makes sense that perhaps you would get a better estimate as the project became closer and closer to finishing.

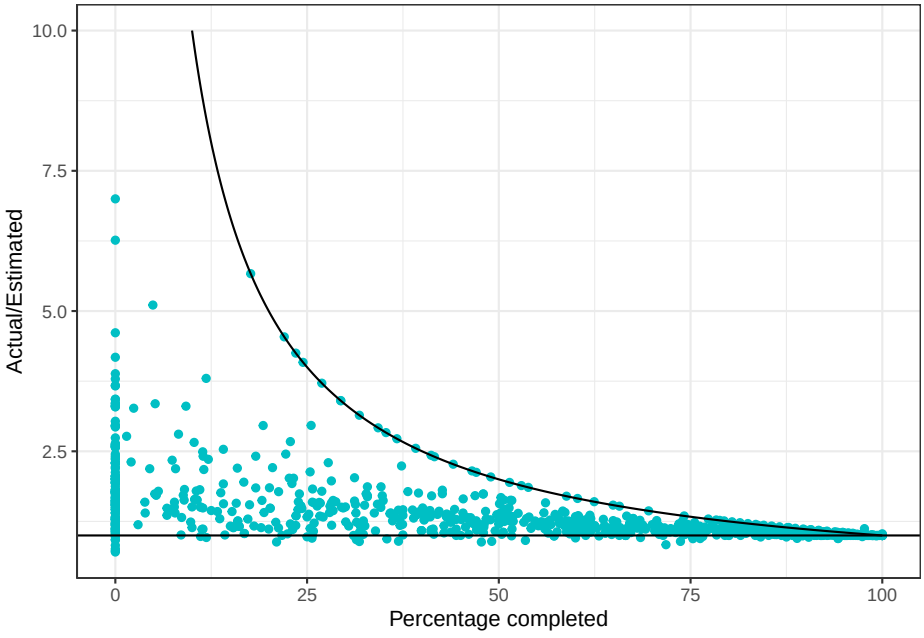
```
# Project Code,Week End Date,Target RelDate,Env,Plan,Dev,Stab,Mob,Adj Week,Est EndDate,First Est,
# 1,12/01/00,03/01/01,C,C,I,,,12/01/00,03/01/01,03/01/01,08/22/01,12/01/00,0.00,0.6770156819,2.95
est=read.csv(paste0(ESEUR_dir, "data/Little06.csv.xz"), as.is=TRUE)

est$Week.End.Date=as.Date(est$Week.End.Date, format="%m/%d/%y")
est$Target.RelDate=as.Date(est$Target.RelDate, format="%m/%d/%y")
est$Adj.Week=as.Date(est$Adj.Week, format="%m/%d/%y")
est$Est.EndDate=as.Date(est$Est.EndDate, format="%m/%d/%y")
est$First.Est=as.Date(est$First.Est, format="%m/%d/%y")
est$Actual.Release=as.Date(est$Actual.Release, format="%m/%d/%y")
est$Start.Date=as.Date(est$Start.Date, format="%m/%d/%y")
est$est_duration=est$First.Est-est$Start.Date
```

```
mk_target_unique=function(df)
{
# The same Target release date might be estimated again after a change of date,
# so duplicated cannot be used.  rle only works with atomic types
t=cumsum(c(1, head(rle(as.integer(df$Target.RelDate))$lengths, n=-1)))
return(df[t, ])
}

target_summary=function(df)
{
return(data.frame(est_duration=df$est_duration[1], num_reest=nrow(df)))
}


u_est=ddply(est, .(Project.Code), mk_target_unique)
t_sum=ddply(u_est, .(Project.Code), target_summary)

ggplot(t_sum,aes(est_duration,num_reest))+
  geom_point(color="#00BFC4")+
  xlab("Initial estimated duration (days)")+
  ylab("Number of estimates")+
  stat_smooth(method="loess",color="black")+
  theme_bw()
```

```
## Don't know how to automatically pick scale for object of type difftime. Defaulting t
```



```
pal_col=rainbow(2)
mk_target_unique=function(df)
{
# The same Target release date might be estimated again after a change of date,
# so duplicated cannot be used.  rle only works with atomic types
t=cumsum(c(1, head(rle(as.integer(df$Target.RelDate))$lengths, n=-1)))
return(df[t, ])
}

est$percent_comp=100*as.integer(est$Week.End.Date-est$Start.Date)/as.integer(est$Actual.Release-e
est$AE_ratio=as.integer(est$Actual.Release-est$Start.Date)/as.integer(est$Target.RelDate-est$Star
est=subset(est, Week.End.Date <= Actual.Release)
u_est=ddply(est, .(Project.Code), mk_target_unique)

x_vals=seq(from=10, to=100,by=((100 - 10)/(length(u_est$percent_comp) - 1))) #just some slight co

ggplot(u_est,aes(percent_comp,AE_ratio))+
  geom_point(color="#00BFC4")+
  geom_line(aes(x_vals,100/x_vals))+
  geom_hline(yintercept=1)+
  xlab("Percentage completed")+
  ylab("Actual/Estimated")+
  theme_bw()
```



Chapter 7

Sleep Deprivation and Software Engineering Performance

7.1 "Pulling an All-Nighter"

Whether you stocked up on coffee and snacks and holed up in the library to finish programming that project due tomorrow, or stayed up all night playing video games until your eyes can only squint, perhaps you are familiar with what the next day feels like after a night without sleep. Unfortunatlely, studies suggest that sleep deprivation is cognitively comparable to being drunk, with impairments to memory, reasoning, reaction time, and decision making. I think we can all agree that everyone would prefer that critical software was not made by a drunk programmer. But is it really *that bad*? Especially given the trope that programmers are up all night shouting into their gaming headsets or hacking the CIA, how bad could it really be to lose a little sleep? Answer: **bad**. It's really bad. This lesson will walk us through just how bad it is to be sleep deprived while trying to write code, and teach you how to interpret experiment design and results of an academic paper. Hopefully by the end, you'll understand statistical concepts like: **experimental design, hypothesis testing, mean, variance, standard deviation, the normal distribution, Type I and Type II errors, Bonferroni correction, correlation, Cliff's delta (effect size), and the Kruskall Wallis Test**. And hopefully you'll realize how abysmal your code will be if you forgo those extra hours to order 3am pizza.

```
# there is actually A LOT to unpack here. All mean/variance/sd/normal distribution stuff should b
# we get a little messy when we get to nonparametric tests, effect size, and particularly the bon
```

7.2 Research Question

This lesson will discuss and follow the methods of *Need for Sleep: the Impact of a Night of Sleep Deprivation on Novice Developers' Performance.*(Fucci et al., 2018) This will allow you to get practice reading academic research, while also testing out their analysis methods in R.

Given that we *know* sleep deprivation affects cognitive functioning, it wouldn't necessarily be the most interesting question to ask *if* sleep deprivation affects programming, but rather, *how much*? The authors form their first research question as the following:

> "To what extent does sleep deprivation impact developers' performance?"

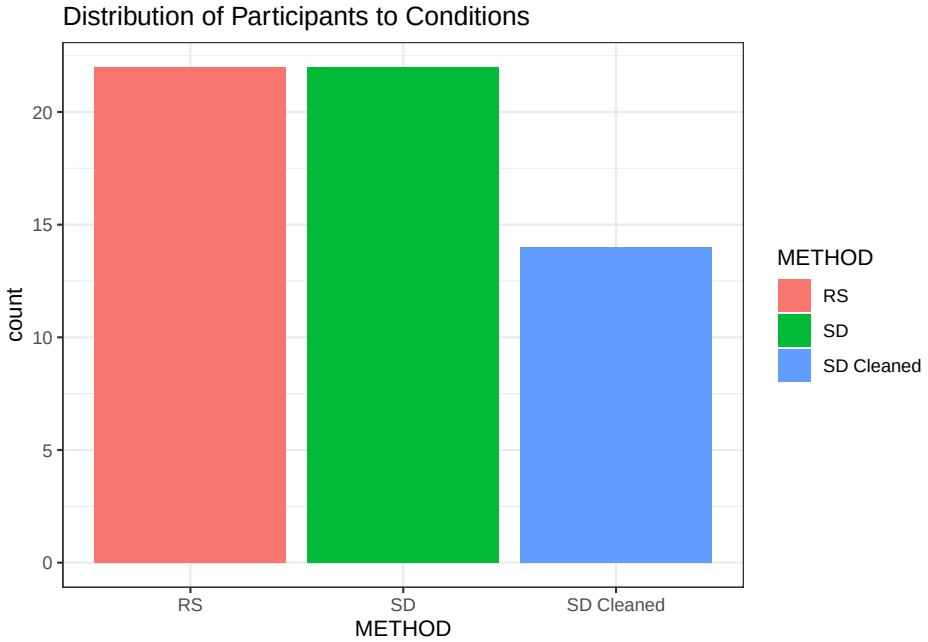
Immediately, you can tell that we are not going over a "Yes" or "No" response, but some kind of measure of *effect size*, and some kind of operationalization of what "performance" really is. Remember that every measure must be carefully defined; and no measure will perfectly capture a larger concept like "performance", "creativity", or "skill". Before we begin describing the methods in the paper, try to think of how *you* might measure developer performance. What would you have the participants do? How would you determine success or failure? Next, how would you check how much sleep deprivation was affecting that

measure?

7.3 Who are the participants?

Let's start by getting to know our participants, particularly between the conditions (Sleep Deprivaton vs. Regular Sleep)

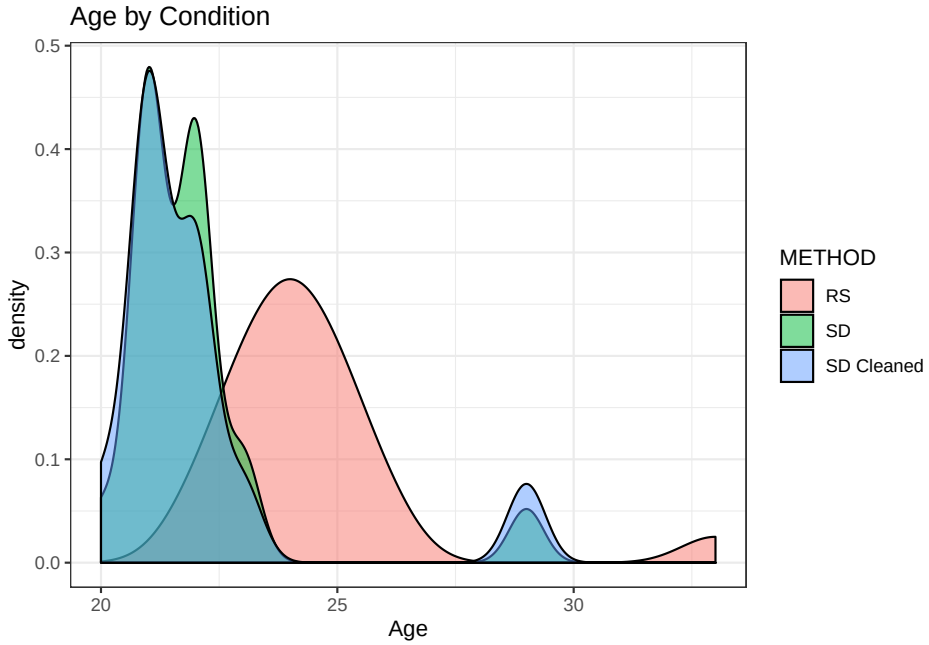
```
library(readxl)
library(ggplot2)

subTable <- function(data, nameCol, val){
  return (data[which(data[nameCol] == val),])

}

# loading data including the sleep deprived set where we remove some particpants

loadAllData <- function(fileName = "data/sleepDepr/piglatin.xlsx"){

  Exp <<-   read_xlsx(fileName)
  Exp_Cleaned <<- subTable(Exp, "PVT-remove", "NO")
  SlD <<- subTable(Exp, "METHOD", "SD")
  NOSD <<- subTable(Exp, "METHOD", "RS")
  SD_Cleaned <<- subTable(SlD, "PVT-remove", "NO")
  NOSD_Cleaned <<- subTable(NOSD, "PVT-remove", "NO")
}
loadAllData()

# this is the post-questionnaire
post <- read_xlsx("data/sleepDepr/post-questionnaire.xlsx")

#merging things together so we can work with one dataframe
data <- merge(Exp,post,by="ID")
SD_Cleaned$METHOD= "SD Cleaned"
SD_Cleaned <- merge(SD_Cleaned,post,by="ID")
data <- rbind(data,SD_Cleaned)

#who is in what condition?
plt = ggplot(data,aes(METHOD,fill=METHOD))+
  geom_histogram(stat="count")+
  ggtitle("Distribution of Participants to Conditions")+
  theme_bw()
plt
```



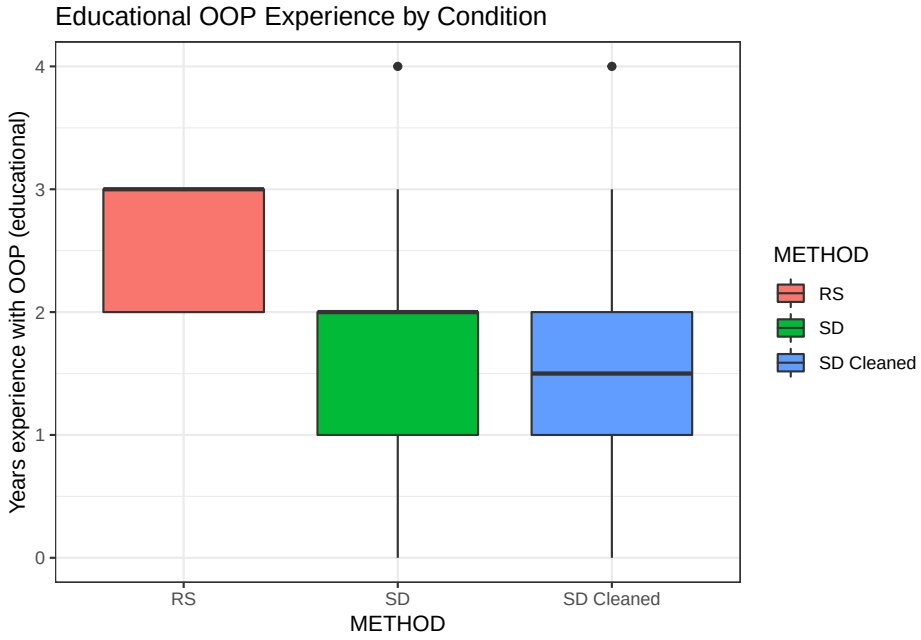

```
# 22 in RS, 22 in SD, 14 in cleaned set (different from in the paper, but just by 1)
kable(table(data$METHOD))
```

Var1	Freq
RS	22
SD	22
SD Cleaned	14

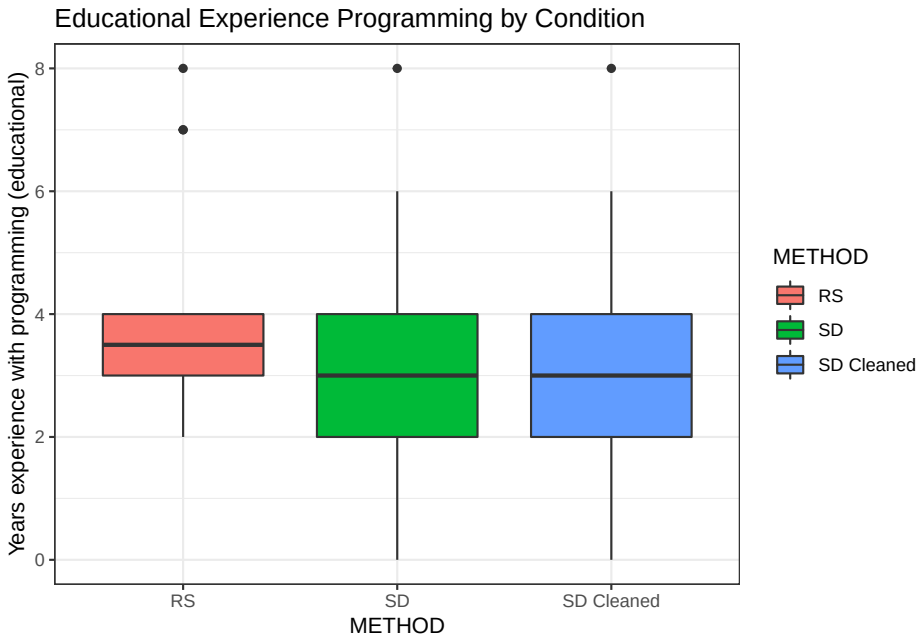
```
#density plot showing age distribution. We see that age is probably significantly different betwe
plt = ggplot(data,aes(Age,fill=METHOD))+
  geom_density(position="dodge",bins = 30,alpha=.5)+
  ggtitle("Age by Condition")+
  theme_bw()
plt
```



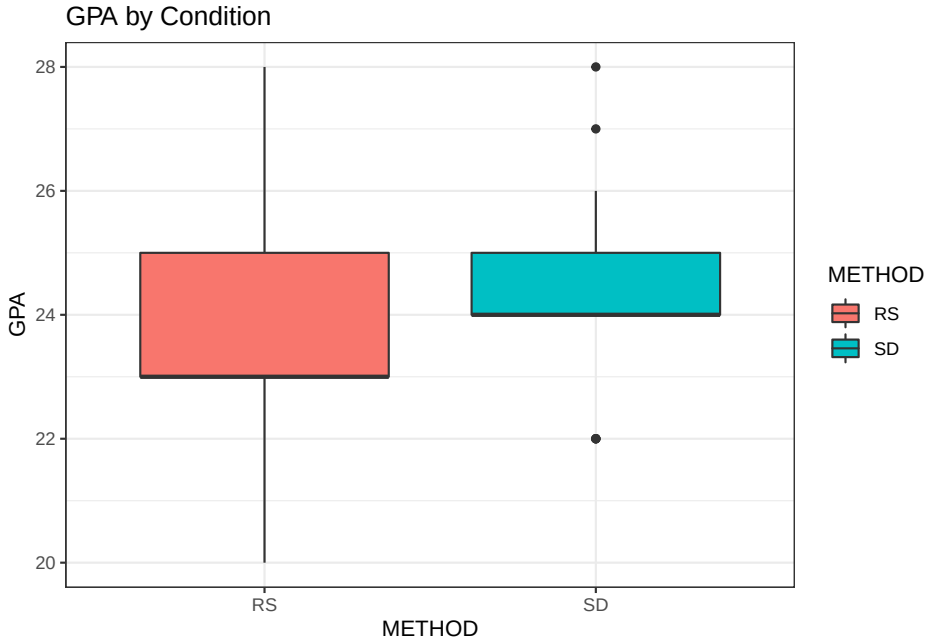

```
nas <- which(is.na(as.numeric(as.character(data$`During your education, how many years
data$`During your education, how many years of experience did you have with the Object

#were participants in the different groups experienced differently?
plt = ggplot(data,aes(METHOD,data$`During your education, how many years of experience
 geom_boxplot()+
  ylab("Years experience with OOP (educational)")+
  ggtitle("Educational OOP Experience by Condition")+
  theme_bw()
plt
```




```
plt = ggplot(data,aes(METHOD,as.numeric(as.character(data$`During your education, how many years
 geom_boxplot()+
  ggtitle("Educational Experience Programming by Condition")+
  ylab("Years experience with programming (educational)")+
  theme_bw()
plt
```




```
plt = ggplot(Exp,aes(METHOD,GPA, fill=METHOD))+
  geom_boxplot()+
  ggtitle("GPA by Condition")+
  theme_bw()
plt
```



7.4 Quasi-experimental setup

In most cases, the ideal experimental conditions would involve randomly selecting participants and assigning them into conditions (in our case, **Sleep Deprivation** or **Regular Sleep**). You can imagine this might be difficult to do with undergraduate students, and difficult to enforce. So this study relies on self-selected students who volunteered to pull an all-nighter before the programming task the next day. If you were to be asked *right now* to participate in the study, could you afford to not sleep tonight? Or would you need to be in the rested group? Perhaps you have an athletic tournament coming up, or just lost sleep the other night and can't afford another one. Or maybe, you planned on staying up tonight *anyways*, so you'd be happy to go in the sleep deprivation group (it will keep you accountable to cramming for that exam, right?). Using this kind of experimental design is referred to as **"quasi-experimental"**: the populations in the various conditions were not randomly assigned, but were self-selected into. These are perfectly fine studies to run, as long as you conduct a few tests to ensure the validity of your results. Can you guess the biggest issue with a quasi-experimental setup vs. a true experiment? The potential problem is that the self-selecting groups may be inherently different on *measures other than your independent variable**. So when you think you are measuring the effects of sleep deprivation on developer performance, perhaps you already have a significant difference of GPA or programming skill between your sleepers and non-sleepers. You can imagine coming up with a list of things that might differ between those groups, that might actually affect their developer performance

more than just the sleep deprivation itself.

7.5 How to Read Scientific Papers

Statistical literacy should be deeply connected to *empowerment*. As you gain more and more skills in statistics and hypothesis testing, you become able to critique the bombardment of "facts" thrown at you. Many news stories, videos, and articles report on scientific findings, using statistics to back up their conclusions or simply reporting what they found to be the most interesting about that study. If it's a good article, it should link the original study. With your newfound statistical prowess, you'll be able to look at the study itself and see if you agree with the interpretations from popular media. I'll outline a few important things to look for when you're reading about a study. Follow the checklist for our case study: *Need for Sleep*.

- What is the motivation for the paper? Why are they studying this?
- What is the research question?
- What is the experimental design? Are there groups to be compared? How many participants in each group? How were they assigned into a condition?
- What are they measuring? How do they define that measure?
- List the tests they run and why (including parametric vs. nonparametric destinction)
- What are the conclusions?
- Do you believe the results?

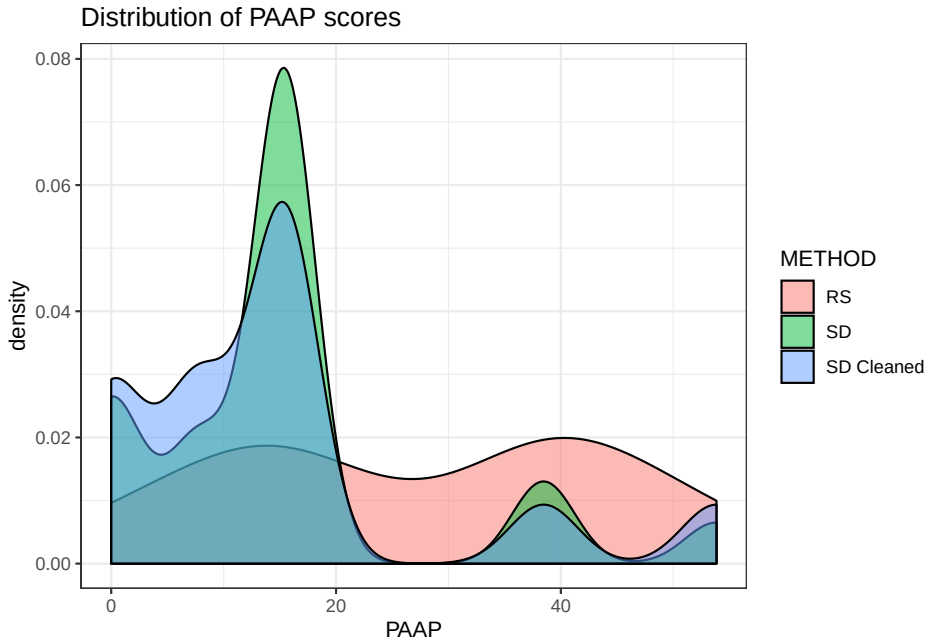
7.6 Percentage of Acceptance Asserts Passed

In our case, one of the major measures of the paper is the Percentage of Acceptance Asserts Passed (PAAP). An `assert` statement will trigger an error if it is not true. Therefore, you can set them up in your code to ensure that nothing is going wrong. For instance, if you were working with probabilities, you would want to assert that `sum(x) == 1`. Participants got 90 minutes to write code to pass the tests in the acceptance test suite, and the percentage of passed tests was recorded. Let's take a look at some descriptive statistics about PAAP.

```
#PAAP
summary <-ddply(data,~METHOD,summarise,mean=mean(PAAP),sd=sd(PAAP),max=max(PAAP),min=m
kable(summary)
```

METHOD	mean	sd	max	min
RS	27.27273	17.07347	53.84615	0
SD	15.38462	13.21730	53.84615	0
SD Cleaned	14.83516	14.92335	53.84615	0

```
plt = ggplot(data,aes(PAAP,fill=METHOD))+
  geom_density(alpha=.5)+
  ggtitle("Distribution of PAAP scores")+
  theme_bw()
plt
```




```
#test PAAP by method for normality
aggregate(formula = PAAP ~ METHOD,
          data = data,
          FUN = function(x) {y <- shapiro.test(x); c(y$statistic, y$p.value)})
```

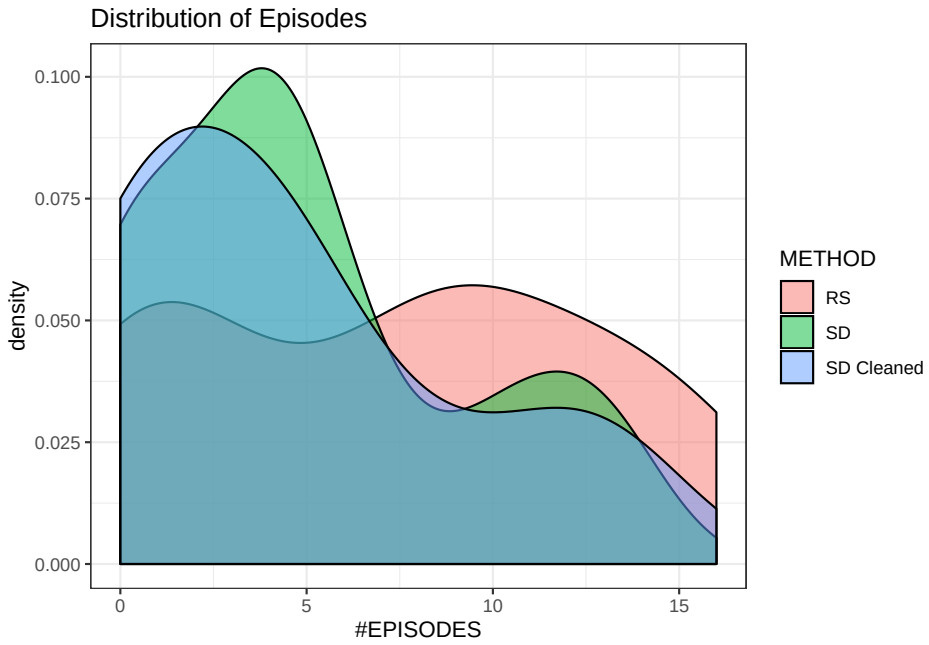
```
##       METHOD       PAAP.W      PAAP.V2
## 1         RS 0.9022845276 0.0330326496
## 2         SD 0.7772646861 0.0002251913
## 3 SD Cleaned 0.7865766146 0.0034218638
```

```
# NOT NORMAL!
```

7.7 Episodes and Conformance

```
plt = ggplot(data,aes(`#EPISODES`,fill=METHOD))+
  geom_density(alpha=.5)+
  ggtitle("Distribution of Episodes")+
  theme_bw()
```

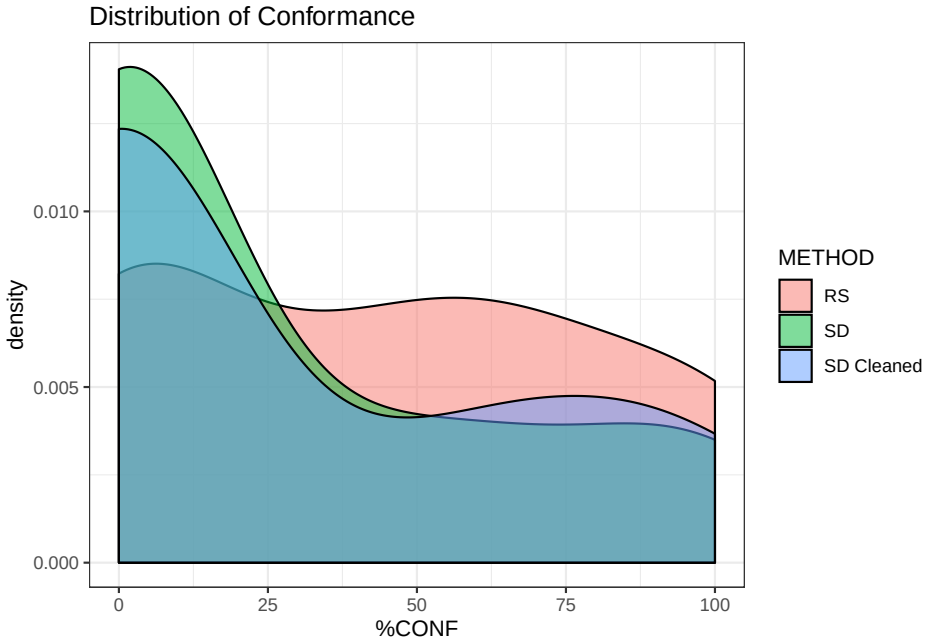
```
plt
```




```
#test EPISODES by method for normality
aggregate(formula = `#EPISODES` ~ METHOD,
          data = data,
          FUN = function(x) {y <- shapiro.test(x); c(y$statistic, y$p.value)})
```

```
##       METHOD #EPISODES.W #EPISODES.V2
## 1         RS  0.91029839   0.04800951
## 2         SD  0.88793575   0.01718088
## 3 SD Cleaned  0.85854606   0.02907183
```

```
# NOT NORMAL

plt = ggplot(data,aes(`%CONF`,fill=METHOD))+
  geom_density(alpha=.5)+
  ggtitle("Distribution of Conformance")+
  theme_bw()
plt
```




```
#test %CONF by method for normality
aggregate(formula = `%CONF` ~ METHOD,
          data = data,
          FUN = function(x) {y <- shapiro.test(x); c(y$statistic, y$p.value)})
```

```
##        METHOD      %CONF.W     %CONF.V2
## 1          RS 8.712612e-01 8.263399e-03
## 2          SD 7.182923e-01 3.374988e-05
## 3 SD Cleaned 6.997035e-01 3.668629e-04
#NOT NORMAL
```

7.8 NOTHING IS NORMAL!!

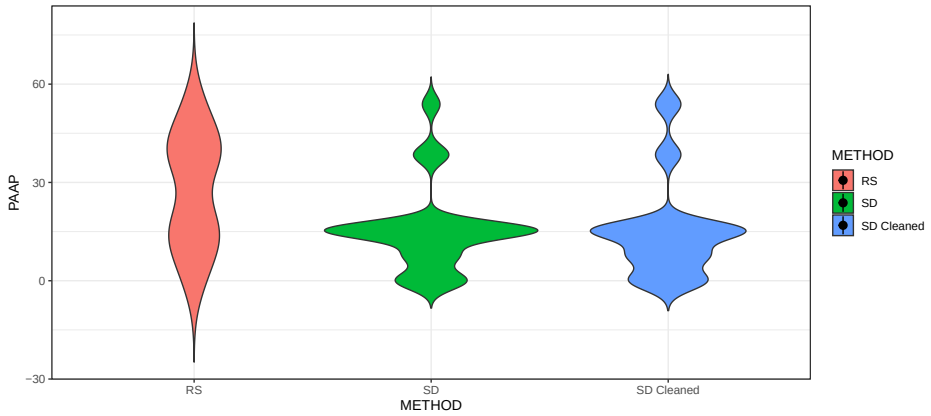
Never fear, we can use **nonparametric tests**.

7.9 Violin Plots

A violin plot is similar to a boxplot, showing the summary statistics like mean and interquartile ranges. A violin plot also demonstrates the distribution and density of the data, shedding light on where the data "falls" around the summary statistics. It is important to critically think about what a mean is expressing. If I have 20 participants who score `PAAP == 58` and 20 participants who score `PAAP == 0`, the mean will be `29`, with not a single participant scoring anywhere near the mean in reality. A violin plot would allow us to discover any misconceptions like that in our data. Below, we plot the different conditions by the different metrics of software quality (PAAP, EPISODES, and CONFORMANCE).

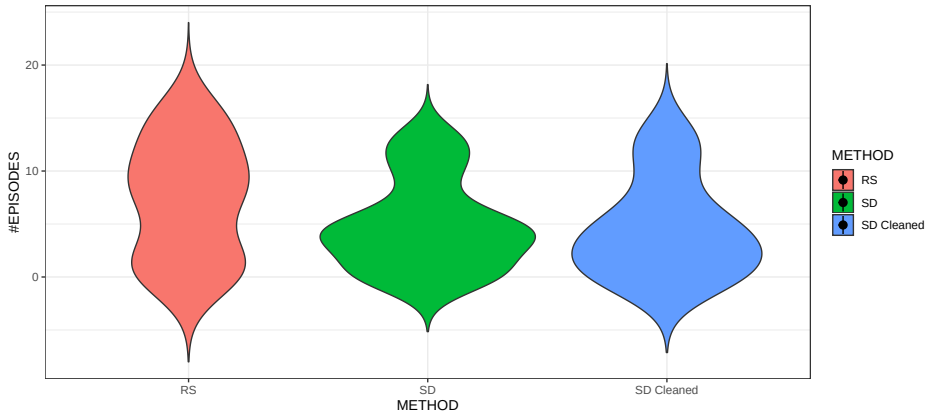
```
ggplot(data,aes(METHOD,PAAP,fill=METHOD))+
  geom_violin(trim=FALSE)+
  #stat_summary(fun.data="mean_sdl", mult=1,
                 #geom="crossbar", width=0.2 )+
  stat_summary(fun.data=mean_sdl,
                 geom="pointrange", color="black")+
  theme_bw()
```

```
## Warning: Computation failed in `stat_summary()`:
## Hmisc package required for this function
```



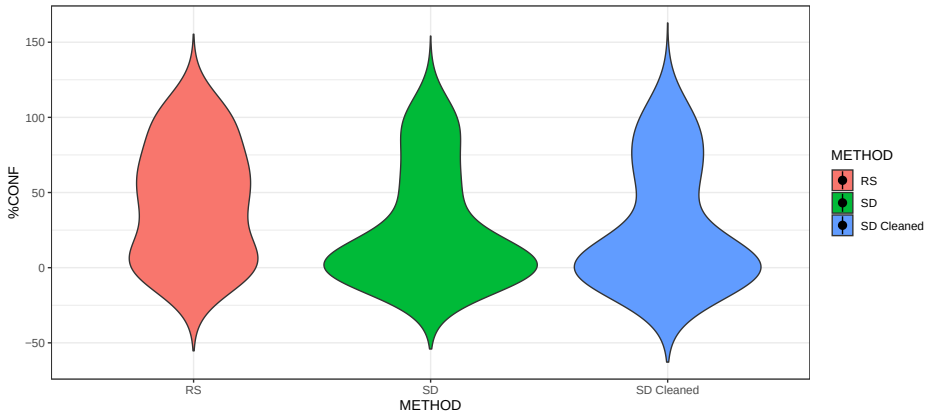

```
ggplot(data,aes(METHOD,`#EPISODES`,fill=METHOD))+
  geom_violin(trim=FALSE)+
  stat_summary(fun.data=mean_sdl,
                  geom="pointrange", color="black")+
  theme_bw()
```

```
## Warning: Computation failed in `stat_summary()`:
## Hmisc package required for this function
```




```
ggplot(data,aes(METHOD,`%CONF`,fill=METHOD))+
  geom_violin(trim=FALSE)+
  stat_summary(fun.data=mean_sdl,
                  geom="pointrange", color="black")+
  theme_bw()
```

```
## Warning: Computation failed in `stat_summary()`:
## Hmisc package required for this function
```



Remember, these are the **key comparisons** of the study. Most other tests are done to ensure that the study has *validity* (that it is testing what they say they are testing). After making sure that the populations are comparable, removing participants who did not adhere to the task, and testing data for normality, we can finally try to answer our question!

> "To what extent does sleep deprivation impact developers' performance?"

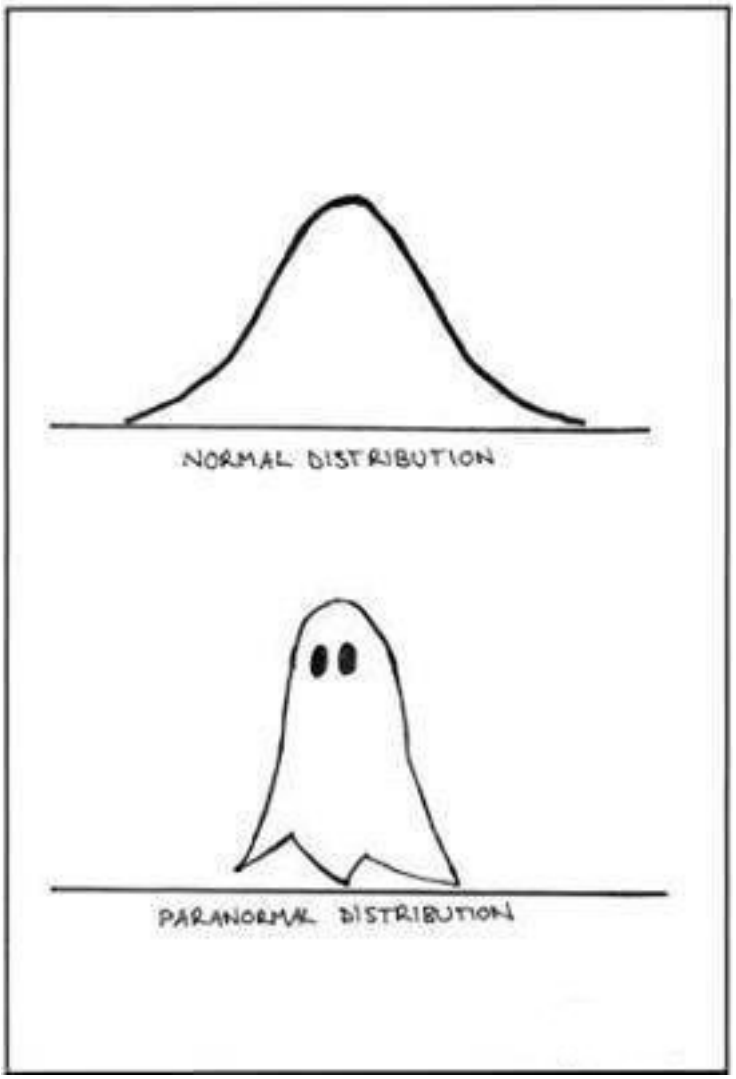
7.10 Effect Size: Cliff's Delta

We've discussed how a *p-value* does not actually tell you anything about how large the effect is. With hundreds of people, we could be absolutely positive that sleep deprivation *does* affect developer performance, but we wouldn't know anything about *how much* from a p-value. People tend to think their effect is larger if they get a smaller p-value (such as .05 vs. .00001). But this is only saying the likelihood that the groups being compared are *not different*. In one case, there is a 5% chance they are actually the same, whereas in the second case, there is only a .001% chance of the groups being actually the same. However confident we are that they are actually different, we still don't know by how much they differ. In practice, you may find that some software is faster than another with a p-value of .0001. You can be *sure* that the second software is faster. But what if it is only faster by a single milisecond? Is it worth re-doing the entire system? You be the judge.

So not only do we need to know the p-value, but we also need the effect size. There are several ways to measure effect size, but we *must* rely on a nonparametric test, due to the non-normality of our data. We use something called **Cliff's delta**, which relies on comparing the positive or negative signs between groups. The function below actually contains thresholds for if we can consider the effect to be large, medium, or negligable.

```
cliffs.d  <-  function(x,y) {

    r = mean(rowMeans(sign(outer(x, y, FUN="-"))))
    r = round(r,3)
    #cat("Cliff-Deelta val = ", r)

    size = "large"
    if ( 0.147 < abs(r) & abs(r) < 0.33){
        size= "small"
    }else{
        if ( 0.33 <= abs(r) & abs(r) < 0.474){
            size = "medium"
        }
    }

    if (abs(r) < 0.147)
        size = "negligible"
    return(c(cat("\n Cliff Delta ", size, " (", r, ")\n"), size, r))



}
```

7.11 Parametric vs. NonParametric Statistical Power



Similar to effect size, we care about **statistical power**. Imagine a scenario where there *is* a difference, it's a *big* difference, but it's *very unlikely* that we will get that result again. Then the confidence that we should reject the null hypothesis is *high*, the effect size is *high*, but the statistical power is *low*. Ideally, we would have high statistical power in addition to a large effect size. Below is a function that routes the data to different tests, depending on the normality of the data. We already know that our data requires nonparametric testing. You can see in the code for the nonparametric test that the p value is *simulated* and

averaged in 1000 runs.

```
StatisticalPowerParam <- function(distribution1, nameDistribution1, distribution2, nameDistributi

#
#  parametric analysis
#

sdd <- sd(c(distribution1,distribution2))
delta <- abs(mean(distribution1) - mean(distribution2))
n = max(length(distribution1), length(distribution2))
pow <- power.t.test(n, delta, sdd, sig.level=0.05, power=NULL, type="two.sample", alternative="tw
power = round(pow$power,3)
cat(" Statistical Power", power)
cat(" beta-val", 1- power, "\n")

return(power)
}

StatisticalPowerNonParam <- function(distribution1, nameDistribution1, distribution2, nameDistrib

#  non-parametric analysis
 M1 <- mean(distribution1)
 M2 <- mean(distribution2)
 sd1 <- sd(distribution1)
 sd2 <- sd(distribution2)
 n1 <- length(distribution1) ### sample size
 n2 <- length(distribution2) ### sample size
 n <- n1 + n2
 pval <- replicate(1000, wilcox.test(rnorm(n1,M1,sd1), rnorm(n2,M2,sd2), paired=FALSE)$p.value)
 power = round(sum(pval < 0.05)/1000,3)
 cat(" Statistical Power", power, "\n")
 cat(" beta-val", 1- power)
 return(power)
}
```

7.12 Wilcoxon (aka Mann-Whitney) Test

The big concept to follow here is that because our data is **not normal** it warrants a nonparametric comparison test. When you have two groups that are not normally distributed, you can use the `wilcox.test` function in R. Each nonparametric test has a parametric counterpart, which tends to be more powerful because the data is normally distributed and can be more sensitively represented using parameters. The Wilcoxon test's parametric counterpart is the **t-test**. We can see that if our data *did* conform to a normal distribution, the below function

would route the data to the parametric test, which is a t-test. The Wilcoxon test and the t-test will give us *confidence* that we can reject the null hypothesis, but the function also outputs the effect size, statistical power, and percentage improvement between groups.

```
Control_vs_Treatment <- function(nameTreatment, treatment, nameControl, control, direct

  sTreatment <- shapiro.test(treatment)
  sControl <- shapiro.test(control)

  if (sTreatment$p.value > 0.05 && sControl$p.value > 0.05) {
    cat(" Parametric analyses allowed \n", nameTreatment, "is normal ", round(sTreatmer
        nameControl, " is normal ", round(sControl$p.value,4), "\n")

    #     #### The distributions are both normal parametric analyses can be computed.
    #       # t-test
    a <- t.test(treatment, control, alternative = direction, paired = FALSE, exact = FA
    a = round(a$p.value,3)
    #       # ANOVA
    #     a <-anova(lm(dataExp$X.TUS ~ dataExp$Method))
    #     a = round(as.numeric(a$`Pr(>F)`[1]),3)
    effectSize <- dCohen(treatment, control, "independent")
    s <- StatisticalPowerParam(treatment, nameTreatment, control, nameControl, directio

  }else {
    cat(" Non-parametric analyses \n", nameTreatment, ": ", round(sTreatment$p.value,4)
        ": ", round(sControl$p.value,4))

    a <- wilcox.test(treatment, control, alternative = direction, paired = FALSE, exact
                     correct = TRUE)
    #a = a$p.value
    a = round(as.numeric(a[3]),3)
    effectSize <- cliffs.d(treatment,control)
    s = StatisticalPowerNonParam(treatment, nameTreatment, control, nameControl)
  }

  #descriptive stats for treatment
  t = summary(treatment)

  #descriptive stats for control
  c = summary(control)
  impr = ((t[4]-c[4])/c[4])*100
  impr = round (impr,3)
  cat("\n Mean improvement", impr,"\\%")

  cat("\n Statistical test p-value (i.e., Hnx) ", a, "\n")
```

```
  if (a < 0.05){
    cat("***  STATISTICALLY SIGNIFICANT DIFFERENCE ***\n")
  }
  cat("&", a, "&", effectSize, "&", impr, "&", s )

}
```

7.13 Results

```
Control_vs_Treatment('SD PAAP',SD_Cleaned$PAAP,'RS PAAP',NOSD_Cleaned$PAAP) #large and significar
```

```
##  Non-parametric analyses
##  SD PAAP :  0.0034
##  RS PAAP :  0.0344
##  Cliff Delta  large  ( -0.476 )
##  Statistical Power 0.664
##  beta-val 0.336
##  Mean improvement -48.077 \%
##  Statistical test p-value (i.e., Hnx)  0.016
## ***  STATISTICALLY SIGNIFICANT DIFFERENCE ***
## & 0.016 & large -0.476 & -48.077 & 0.664
```

```
Control_vs_Treatment('SD #EPISODES',SD_Cleaned$`#EPISODES`,'RS # EPISODES',NOSD_Cleaned$`#EPISODE
```

```
##  Non-parametric analyses
##  SD #EPISODES :  0.0291
##  RS # EPISODES :  0.0871
##  Cliff Delta  small  ( -0.32 )
##  Statistical Power 0.338
##  beta-val 0.662
##  Mean improvement -39.623 \%
##  Statistical test p-value (i.e., Hnx)  0.116
## & 0.116 & small -0.32 & -39.623 & 0.338
```

```
Control_vs_Treatment('SD %CONF',SD_Cleaned$`%CONF`,'RS %CONF',NOSD_Cleaned$`%CONF`) #small
```

```
##  Non-parametric analyses
##  SD %CONF :  4e-04
##  RS %CONF :  0.0164
##  Cliff Delta  small  ( -0.293 )
##  Statistical Power 0.26
##  beta-val 0.74
##  Mean improvement -40.409 \%
##  Statistical test p-value (i.e., Hnx)  0.135
## & 0.135 & small -0.293 & -40.409 & 0.26
```

We successfully replicate the findings in the *Need For Sleep* paper, demonstrating that there was a significant difference for Percentage of Acceptance Asserts Passed: those who were sleep deprived performed 48% *worse* on the programming task than those who had regular sleep.

Chapter 8

Does It Really Matter to Test First or Test After?

In my experience, the world of software is an *opinionated* one. It seems like everyone has special tools, techniques, practices, and packages that they swear by. In this lesson, we will investigate one of these classic claims: Which is better? Test-first or test-last development? We will also explore what it looks like when people misattribute benefits to a specific practice, when in reality the cause is entirely different. The study we are using is: *A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?*(Fucci et al., 2016) Below is an excellent quote from the paper, highlighting how statistics can separate the relevant findings from other distracting factors:

> This information can liberate developers and organizations who are interested in adopting TDD, and trainers who teach it, from process dogma based on pure intuition, allowing them to focus on aspects that matter most in terms of bottom line.

8.1 Test-Driven Development Process

- what is it?
- why do people care?

8.2 The Study

- beyond their very complex design
- granularity, uniformity, sequencing, and refactoring
- test-first, test-last

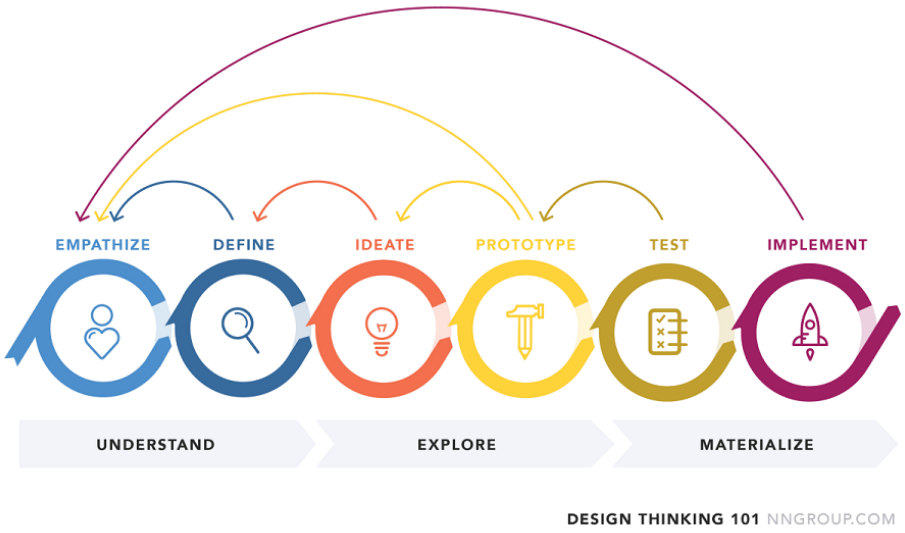


Figure 8.1: "Design Thinking" model emphasizes the iterative dynamic of creating something useful

8.3 Modeling Recipe

Here is a little recipe for **modeling**:

1. Define the **outcome variable(s)** that you care about
2. Define the **factors** that might affect (1); be willing to iterate!
3. Determine which kind of model can explain the data we see: `lm`,`glm`,`chisq`,`anova` etc.
4. Keep it as simple as possible, and try to include as few **features** and **interactions** as you can. *simpler is better, without* ***underfitting***
5. Determine an appropriate metric to **evaluate the model**: `rmse`, `R^2`, `AIC`, `BIC`, etc.
6. Compare models to find the model with a nice balance between simplicity and **goodness of fit**.
7. Continue to ask yourself; does this model actually make sense in the world? Even if it mathematically **fits** really well, is it plausible?
8. If so, then you have a very reasonable model! If you can, test it on new data, or perform an experiment that confirms your **hypothesis**.

9. Without experimentation, you cannot know anything about **causality**. Instead, your model provides a representation of the phenomenon and the factors involved.

8.4 Outcomes We Care About

> *Which subset of the factors best explain the variability in external quality?*

So basically this question is asking "what combo of variables contributes to making stuff good?"

With any measure, we have to **operationalize** it in some way. "Code Quality" is defined as:

$$QLTY = \frac{\sum_{i=1}^{\#TUS} QLTY_i}{\#TUS} \times 100,$$

with $QLTYi$ defined as:

$$QLTY_i = \frac{\#ASSERT_i(PASS)}{\#ASSERT_i(ALL)}.$$

Don't panic. I've actually wanted to do a research study where I use eyetracking technology to watch how learners react to equations in an academic paper. If you're anything like me, your eyes *glaze right over it* and then you curse yourself for not immediately understanding it by just absorbing it into your mind without reading it. When you dig into these equations, you see that it's a fancy way of counting stuff. `TUS` is the number of "Tackled User Stories", or "how many problems attempted". So, for each of those stories, count up the proportion of passing assert statements, and take the average over all the stories attempted. This way, we are simply measuring functional correctedness, with no accounting for things like style or readability.

> *Which subset of the factors best explain the variability in developer productivity?*

And this question is asking "what combo of variables contributes to developers producing more?" (though you may remember from Analyze This! that this may be unwise to measure!)

Productivity is defined as:

$$PROD = \frac{OUTPUT}{TIME}$$

with OUTPUT as the total passing assert statements. TIME measures from the

time the task is opened until closed.

8.5 Factors We Measure

The following is taken directly from Table 2.

- **Granularity**: A fine-grained development process is characterized by a cycle duration typically between 5 and 10 minutes. A small value indicates a granular process. A large value indicates a coarse process.
- **Uniformity**: A uniform development process is characterized by cycles having approximately the same duration. A value close to zero indicates a uniform process. A large value indicates a heterogeneous, or unsteady, process.
- **Sequencing**: Indicates the prevalence of test-first sequencing during the development process. A value close to 100 indicates the use of a predominantly test-first dynamic. A value close to zero indicates a persistent violation of thre test-first dynamic.
- **Refactoring**: Indicates the prevalence of the refactoring activity in the development process. A value close to zero indicates nearly no detectable refactoring activity (negligable refactoring effort). A value close to 100 indicates a process dominated by refactoring activity (high refactoring effort).

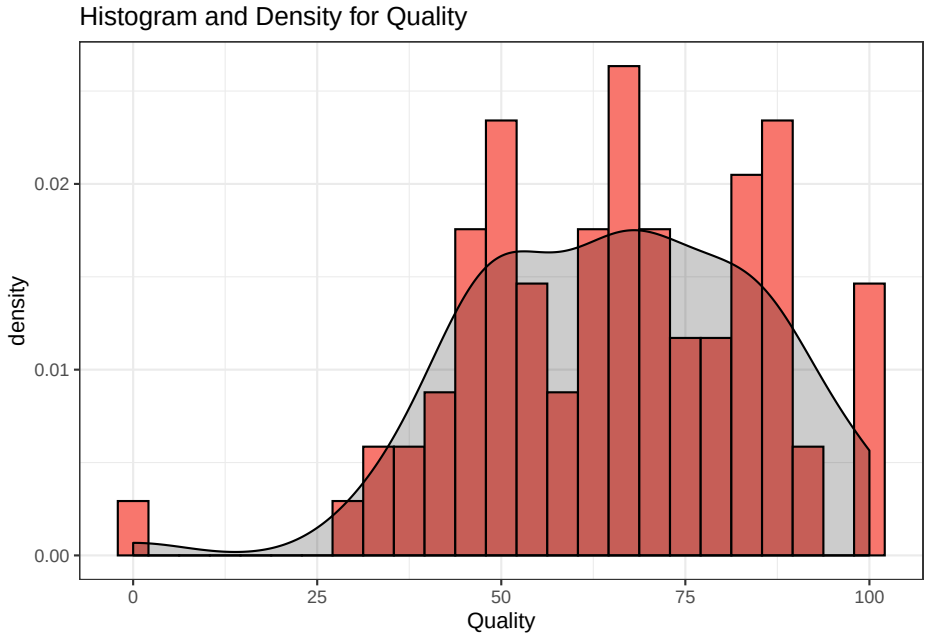
```
data <- read.csv("data/dissectionTDD/dataset.csv",sep=";")
head(data)
```

```
##   ID  QLTY PROD   GRA  UNI   SEQ   REF
## 1  1 77.27 0.07  0.87 0.29  0.00 45.45
## 2  2 58.71 0.21 31.07 3.26 50.00 50.00
## 3  3 85.42 0.32  1.66 1.78 28.12  6.25
## 4  4 84.52 0.34  1.05 0.91  8.62 50.00
## 5  5 45.91 0.17  7.30 6.40 20.00 26.66
## 6  6 77.95 0.37  8.02 8.36  0.00 44.44
```

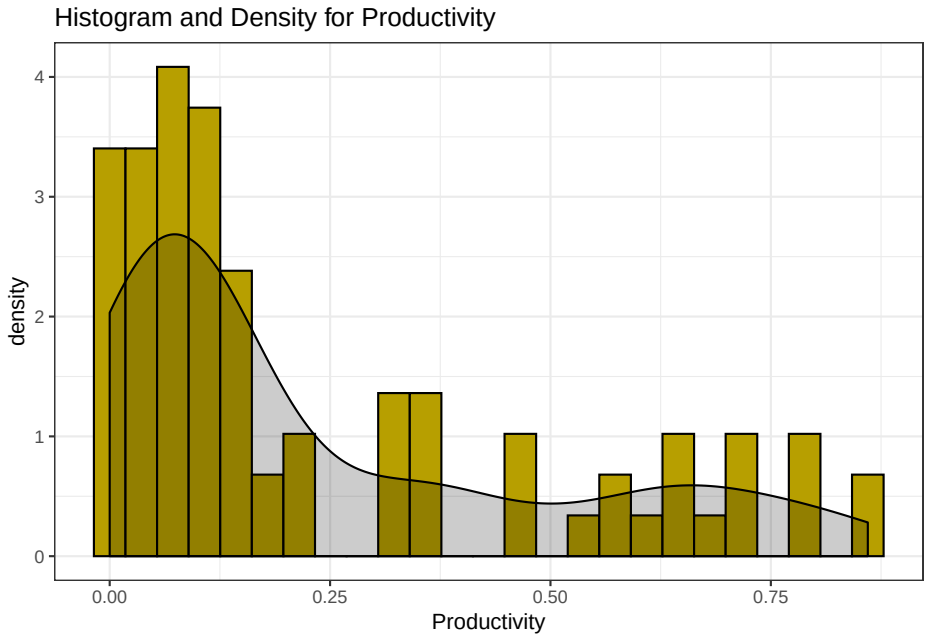
8.6 Descriptive Statistics

There is a difference between **descriptive statistics** and **inferential statistics**. Descriptive statistics describe properties of the data; such as **means**, **ranges**, and **normality** of the variables of interest. Inferential statistics will draw the actual *conclusions* about the data; reporting on **correlations**, **hypothesis tests**, and [**estimation of parameters]**](glossary.html#parameter). Inferential statistics helps to generalize about a larger **population**, that can go beyond the descriptive statistics of an immediate **sample**.

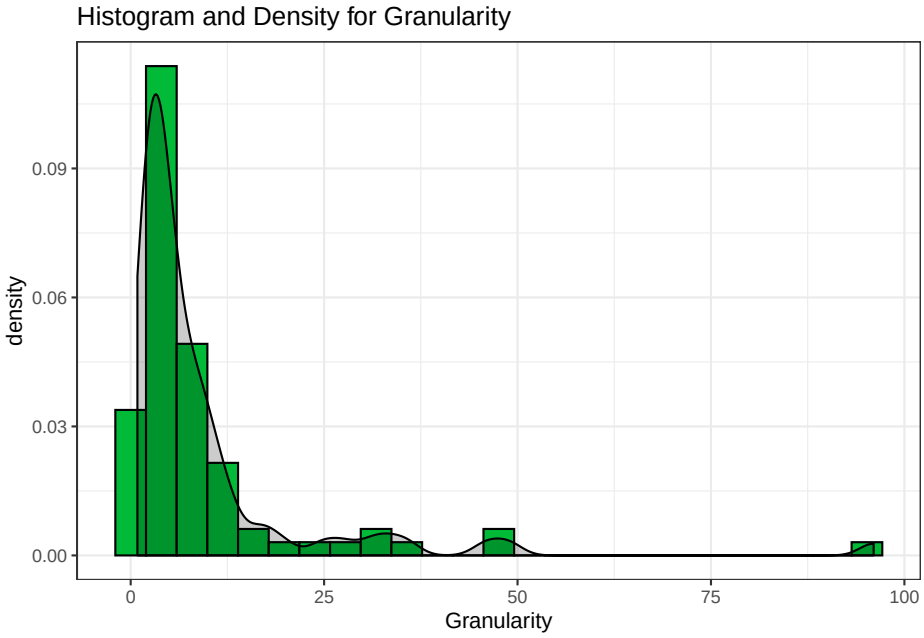
```r
library(ggplot2)

#small function to generate colors for ggplot
gg_color_hue <- function(n) {
  hues = seq(15, 375, length = n + 1)
  hcl(h = hues, l = 65, c = 100)[1:n]
}

titles <- c("Quality","Productivity","Granularity","Uniformity","Sequencing","Refactoring")
#get some colors for each
cols = gg_color_hue(length(titles))

# loop to create 6 density plots to look at spread for each variable
loop <- 2:7
for( i in loop){
  x <- data[[i]]
  plt <- ggplot(data,aes(x)) +
    ggtitle(paste("Histogram and Density for",titles[i-1]))+
  geom_histogram(aes(y = ..density..), bins=25,color="black",fill=cols[i-1])+
  geom_density(aes(y = ..density..),color="black",fill="black", alpha=.2,stat = 'density')+
    xlab(titles[i-1])+
  theme_bw()

  print(plt)
  print(shapiro.test(x))
}
```



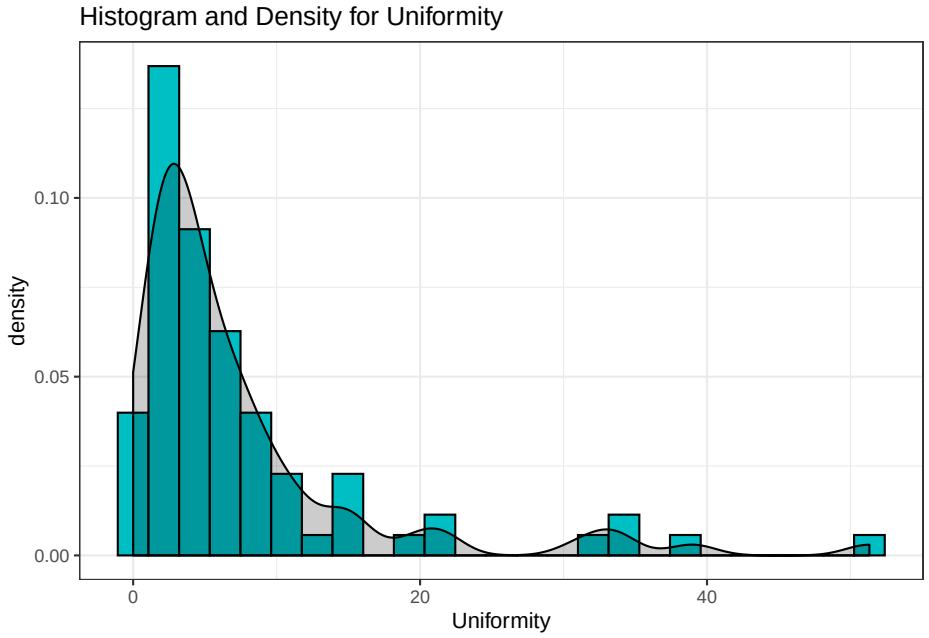

```
##
##  Shapiro-Wilk normality test
##
## data:  x
## W = 0.97198, p-value = 0.0695
```



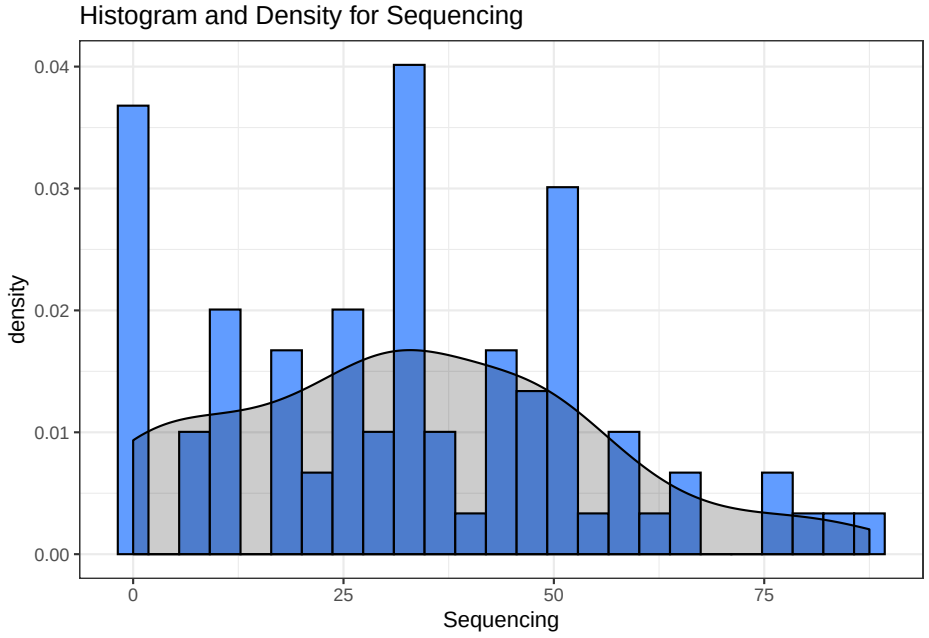
```
##
##  Shapiro-Wilk normality test
##
## data:  x
## W = 0.80645, p-value = 5.326e-09
```



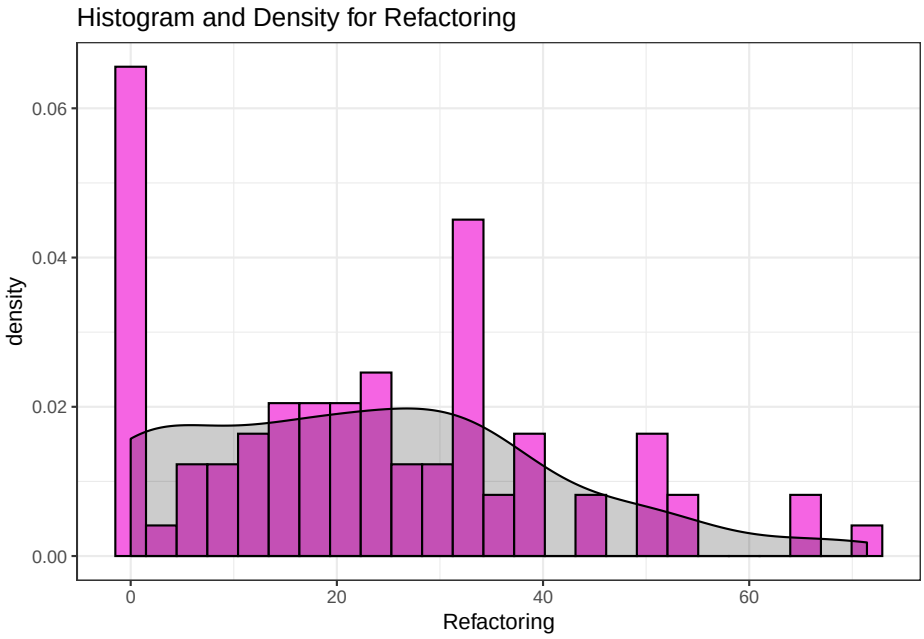

```
##
##  Shapiro-Wilk normality test
##
## data:  x
## W = 0.54482, p-value = 1.524e-14
```




```
##
##  Shapiro-Wilk normality test
##
## data:  x
## W = 0.67565, p-value = 3.623e-12
```



```
##
##  Shapiro-Wilk normality test
##
## data:  x
## W = 0.96182, p-value = 0.01545
```




```
##
##  Shapiro-Wilk normality test
##
## data:  x
## W = 0.94084, p-value = 0.0009059
```

8.7 Let's Talk About Models

Remember that we are looking for *which features affect quality and productivity.* Like many model setups, we have collected data and are looking for which features carry the most **weight**. That means, there are a bunch of factors affecting an outcome (like Productivity) and we want to know which ones matter the most. You might also simply be looking to record the weights on each of the features, not caring which ones "matter" but just trying to get an accurate representation of the phenomenon (say, what is affecting climate change, or what factors lead to higher rates of cancer?) When creating a model, we are looking to represent the various dynamics in the world that make something happen; *we represent it so that we can understand it and predict it.* In our scenario, we have quantitative **measures** for the following: **Granularity, Uniformity,**

Sequencing, Refactoring, Quality, and **Productivity**. We want to know how `GRA`, `UNI`, `SEQ`, and `REF` are affecting our outcome variables: `PRO` and `QLTY`. Ideally, our objective would be to provide a conclusion like *"The more refactoring you do, the better your code quality, and test-first is better for productivity"* (not an actual conclusion of this paper). This paper keeps a certain tone throughout, emphasizing how we can use actual metrics to stop petty debates within the software world.

If you jump ahead to the **Discussion**, we know that the eventual findings are:

> We conclude that granularity, uniformity and refactoring effort together constitute the best explanatory factors for external quality [and] productivity.

So what would that conclusion look like in model form? First off, the conclusion implies that `SEQ` does not matter, and is not even included in the best performing model. This is interesting, because so many people evangelize the Test-Driven Development technique! In fact, it's not the sequencing of when you write the tests, but the level to which you iterate rapidly when writing code that affects quality and productivity. It could be the case that people attributed their success to the wrong factor: *test-first* or *test-last*, when in fact, iteration was key to success.

8.8 Regression Analysis vs. Hypothesis Testing

- explain how because theyre all continuous variables, we dont do hypothesis testing. we use regression TODO

```
m <- glm(QLTY ~ GRA + UNI + REF + SEQ +GRA:UNI+SEQ:REF, data=data)
m
```

```
##
## Call:  glm(formula = QLTY ~ GRA + UNI + REF + SEQ + GRA:UNI + SEQ:REF,
##     data = data)
##
## Coefficients:
## (Intercept)          GRA          UNI          REF          SEQ
##   78.880246    -0.380340    -0.605808    -0.298070     0.012607
##     GRA:UNI      REF:SEQ
##    0.005605     0.001325
##
## Degrees of Freedom: 81 Total (i.e. Null);  75 Residual
## Null Deviance:       30130
## Residual Deviance: 24850     AIC: 717.2
```

```
summary(m)
```

```
##
```

```
## Call:
## glm(formula = QLTY ~ GRA + UNI + REF + SEQ + GRA:UNI + SEQ:REF,
##     data = data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -60.146  -12.889    0.459   12.631   32.976
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 78.880246   7.132852  11.059   <2e-16 ***
## GRA         -0.380340   0.193825  -1.962   0.0534 .
## UNI         -0.605808   0.489124  -1.239   0.2194
## REF         -0.298070   0.190446  -1.565   0.1218
## SEQ          0.012607   0.137549   0.092   0.9272
## GRA:UNI      0.005605   0.013912   0.403   0.6882
## REF:SEQ      0.001325   0.005571   0.238   0.8126
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 331.2695)
##
##     Null deviance: 30129  on 81  degrees of freedom
## Residual deviance: 24845  on 75  degrees of freedom
## AIC: 717.23
##
## Number of Fisher Scoring iterations: 2
```

```
AIC(m)
```

```
## [1] 717.2294
```

```
summary(lm(QLTY ~ GRA + UNI + REF + SEQ +GRA:UNI+SEQ:REF, data=data))$r.squared
```

```
## [1] 0.1753826
```

```
m <- glm(QLTY ~ GRA + UNI +REF, data=data)
#m
#summary(m)
AIC(m)
```

```
## [1] 711.5502
```

```
summary(lm(QLTY ~ GRA + UNI + REF + SEQ, data=data))$r.squared
```

```
## [1] 0.1731051
```

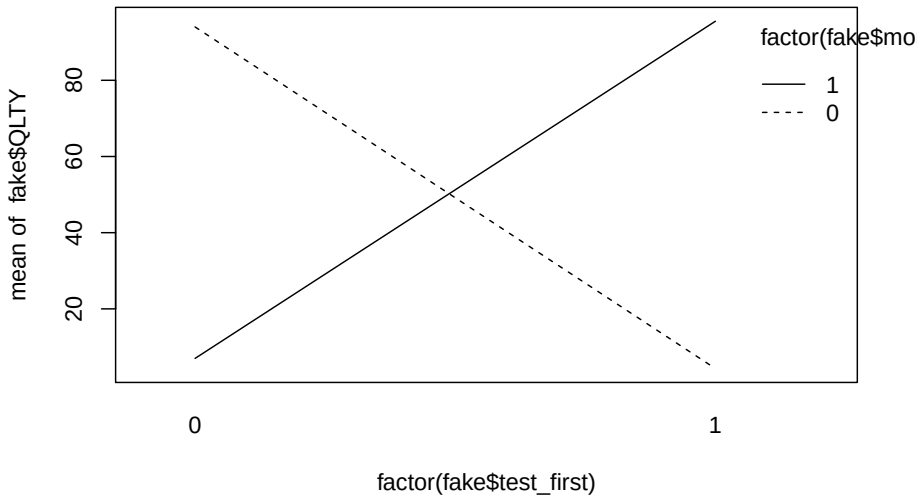
8.9 Let's Talk About Interactions

This paper mentions something called **interactions**. We will walk through a more basic example of what an interaction is before exploring what it means in this context. You can imagine a scenario where there is a **categorical variable**: `test_first`, which can take on either 1 or 0 if the developer tested first or last, respectively. Now imagine we also record whether developers are coding in the morning or the afternoon. *It just so happens in our fake scenario*, that if you test-first in the mornings, your quality skyrockets. If you test-first any other time, you don't get the same effect. Therefore, there is an interaction between testing first and the time of day. This is **problematic** because if we don't look into this interaction, we simply report a mean quality of somewhere in the middle; not accounting for how the test sequencing and time of day is affecting that quality. **If there is an interaction, the lines in the categorical interaction plot will cross**.

```
fake <- read.csv("data/fake_interaction.csv")
head(fake,10)
```

```
##    ID test_first QLTY morning
## 1   1          0    9       1
## 2   2          0    8       1
## 3   3          0    4       1
## 4   4          0   97       0
## 5   5          0   91       0
## 6   6          1    0       0
## 7   7          1    4       0
## 8   8          1    9       0
## 9   9          1   93       1
## 10 10          1   98       1
```

```
fake$morning <- as.factor(fake$morning)
fake$test_first <- as.factor(fake$test_first)
interaction.plot(factor(fake$test_first),factor(fake$morning),fake$QLTY)
```



Take a look at the means when we *do* take into account the interaction, looking at both `test_first` and `morning` as contributing to `QLTY`. If we *only* look at one or the other, our means are misleading, because clearly there is another factor affecting the code quality! Sure, we could say that on average the code quality is somewhere in the middle; but it's disingenuous because the truth is that your code quality will suffer if you're using test_first in the afternoon. (In this **FAKE DATA**, of course). In order to truly represent the phenomenon, we have to include results about the interaction.

```
fake %>%
  group_by(test_first,morning) %>%
  summarise(mean=mean(QLTY))
```

```
## # A tibble: 4 x 3
## # Groups:   test_first [2]
##   test_first morning  mean
##   <fct>      <fct>   <dbl>
## 1 0          0        94
## 2 0          1         7
## 3 1          0         4.33
## 4 1          1        95.5
```

```
fake %>%
  group_by(test_first) %>%
  summarise(mean=mean(QLTY))
```

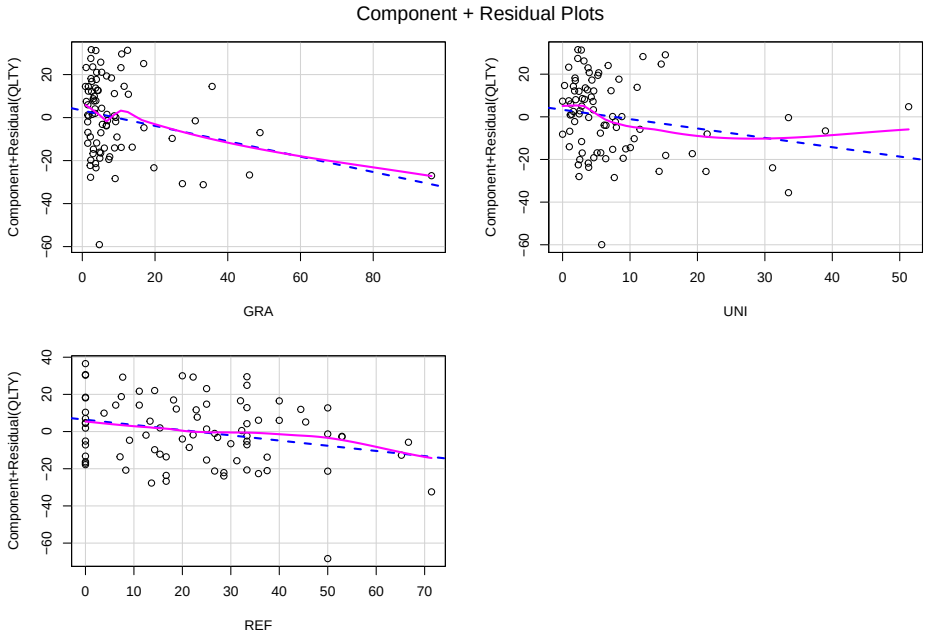
```
## # A tibble: 2 x 2
##   test_first  mean
##   <fct>      <dbl>
## 1 0           41.8
## 2 1           40.8
```

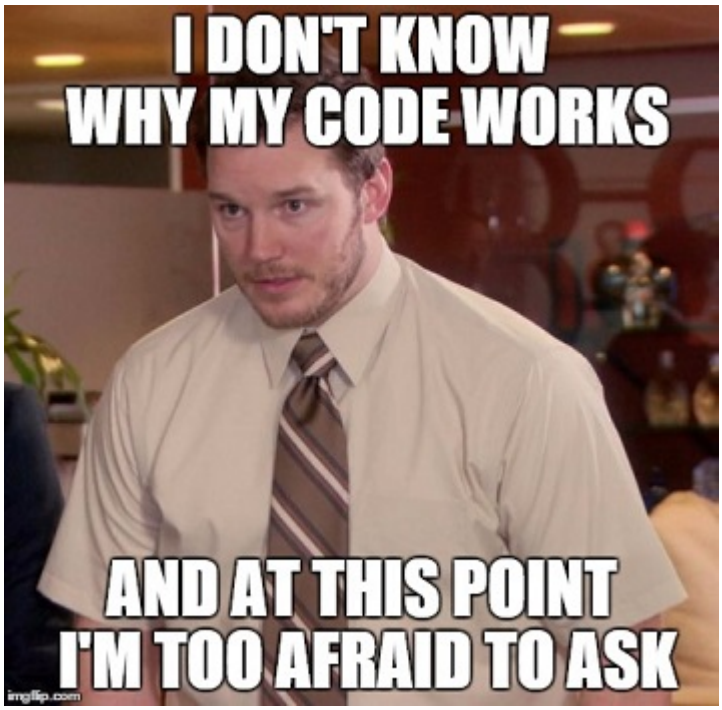
```
fake %>%
  group_by(morning) %>%
  summarise(mean=mean(QLTY))
```

```
## # A tibble: 2 x 2
##   morning  mean
##   <fct>   <dbl>
## 1 0        40.2
## 2 1        42.4
```

8.10 Component + Residual Plots

```
library(car)

m<-glm(QLTY~GRA+UNI+REF,data=data)
crPlots(m)
```





8.11 Interpreting Regression Output

Here I've used a **general linear model** to map out the factors affecting code quality. In the paper, they systematically included or excluded different factors and their interactions, in order to find the best fitting model. "Finding the best fitting model" means finding which combination of factors, and in what weights, best represents the relationship between those factors and the outcome we care about (quality or productivity). Remember, you could always add more and more factors to a model; the code will still output an answer. But it doesn't mean it's a *correct* model, or a *useful* one.

Looking at the final model used in the paper, we see that the factors kept were **Granularity, Uniformity, and Refactoring** in order to model **Code Quality**. This is particularly interesting because **Sequencing** is *not* a factor in the model. The big question from software engineers was about *test-first* vs. *test-last*, but it turns out that it really doesn't matter. The other factors are more important, and have a *negative relationship* with code quality. That negative relationship is because of how the measures were formulated; the closer to 0, the more granular and uniform the cycles were. So, the smaller the values for `GRA`, `UNI` and `REF`, the higher the value on `QLTY`. By looking at the output of the summary of the model, we see the Estimates as negative, and significant (or at least marginally significant). You can see significance by noting the number

of stars (***) by each factor. (.) indicates a marginal trend. The t value is also indicative of how large the effect is, with the absolute value of t indicating significance. All of those significance factors are basically saying "hey! these factors are contributing to this outcome in a meaningful way!"

```
summary(m)
```

```
##
## Call:
## glm(formula = QLTY ~ GRA + UNI + REF, data = data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -60.710  -13.641    0.358   11.733   32.454
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  78.9278     4.0694  19.395   <2e-16 ***
## GRA          -0.3576     0.1690  -2.115   0.0376 *
## UNI          -0.4391     0.2502  -1.754   0.0833 .
## REF          -0.2799     0.1159  -2.416   0.0180 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 319.7768)
##
##     Null deviance: 30129  on 81  degrees of freedom
## Residual deviance: 24943  on 78  degrees of freedom
## AIC: 711.55
##
## Number of Fisher Scoring iterations: 2
```

8.12 Evaluating How Good a Model Is (AIC)

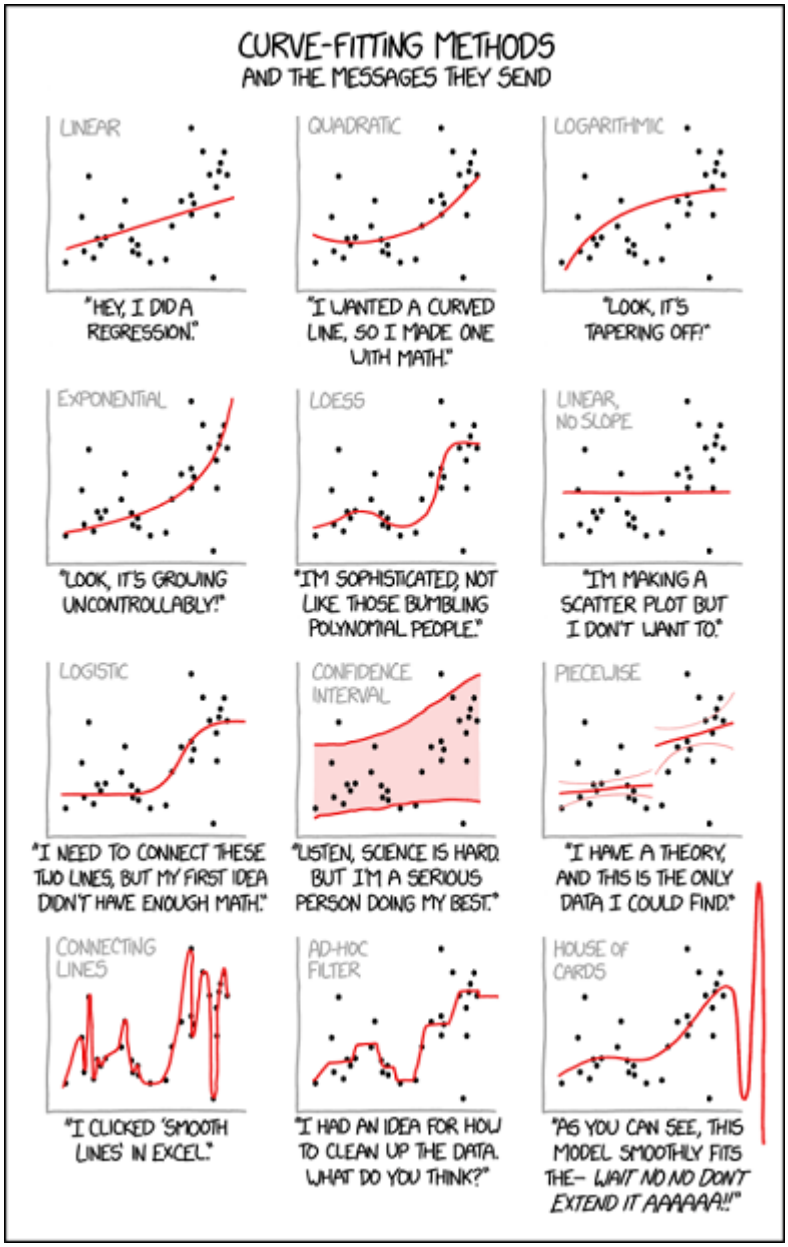
How do we determine which model is the "right" one? Well, there's a few ways to do that. Sometimes, we simply look at how good the model is at predicting test data. In this case, we are using something called AIC (Akaike Information Criterion). This is a very specific way of evaluating a model that deals with *preserving information*. It's interesting and cool, and strikes a nice balance between **goodness of fit** and *simplicity* (not throwing too many parameters into the model). It's okay if that doesn't make sense; just know that a lower AIC indicates a "better" model. Notice how if we include `SEQ` in the model, the AIC is higher.

```
print(AIC(m))
```

```
## [1] 711.5502
```

```
new_m <- glm(QLTY~GRA+UNI+REF+SEQ,data=data)
print(AIC(new_m))
```

```
## [1] 713.4556
```



8.12.1 You could try any combination of factors and their interactions, until you achieved the best fitting model for the phenomenon. Davide Fucci carefully walks through model choice, feature selection, investigation of interactions, and more. Follow the paper step by step to replicate their results on this data!

Chapter 9

Developer Performance and Designing a New Grading System for Teams

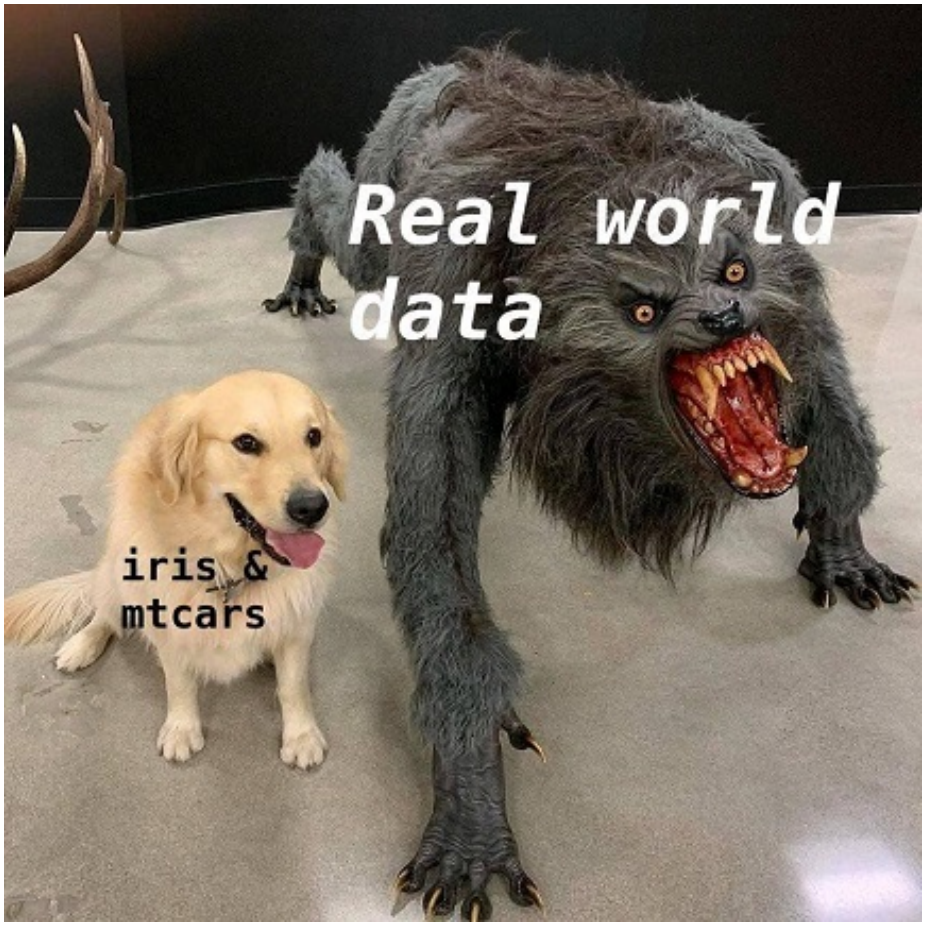
Someone wise once asked me:

> *"How do you choose a heart surgeon?"*
>
> "I don't know, how?"
>
> *"You look for the surgeon with the highest failure rate who is still accepting patients"*
>
> "And why on earth would you want that?"
>
> "*Because it means that they're getting* **all the hardest cases** *and people still trust them to do the surgery.*"

This little exchange is surprisingly relevant to how we measure developer performance. Bear with me. If I were to tell you that the best developer at my company fixed the least number of bugs, what would you think? *Perhaps they are fixing the really tough ones.* It could also be that they simply don't make mistakes but that is just untrue. We all make mistakes in our code and spend a lot of time debugging (novices more than experts, but also the problems get harder so there's a lot at play!). Imagine that at the end of the workweek, everyone has to report how many bugs they fixed. One person reports fixing 78 and another person reports fixing 2. Who is the better developer?

Maybe you have some theories, or some immediate reactions. Perhaps you have some explanations of why that's not enough information. This is great practice for figuring out how to measure a concept like performance.

One way to measure performance is to look at "number of **faults** found" in software testing. Finding and fixing faults may be a reasonable way to determine if developers can identify a problem and implement solutions to help software behave as expected. Reasonable enough, right? Hopefully you're learning that when we measure *anything*, we have to**operationalize** it in a reasonable way, taking into account prior research and interpreting results with our nuanced operationaliztion in mind. "Performance" is not necessarily captured entirely by "number of faults found" but it is totally reasonable to start somewhere. We take a look at data provided from Iivonen's *Identifying and Characterizing Highly Performing Testers–A Case Study in Three Software Product Companies*



http://www.knosof.co.uk/ESEUR/ESEUR-draft.pdf pg 52

9.1 How do developers differ in their measurable output performance?

```
library(knitr)
data <- read.csv("data/performance_dev.csv")

#just adding in a check on the percentages, we see some off-by-ones but that's probably due to ro
data$Totals <- data$Fixed + data$No.Fix +data$Duplicate + data$Cannot.reproduce

kable(data)%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

Tester	Defects	Extra.Hot	Hot	Normal	Status.Open	Fixed	No.Fix	Duplicate	Cannot.reproduce
A	74	4	1	95	12	62	26	12	0
B	73	0	56	44	15	87	6	2	5
C	70	0	29	71	36	71	24	0	4
D	51	0	27	73	33	85	6	0	9
E	50	2	16	82	30	89	9	0	3
F	18	0	22	78	22	64	14	0	21
G	17	18	18	65	18	71	14	0	14
H	17	53	18	29	6	94	0	0	6
I	12	8	17	75	42	100	0	0	0
J	2	0	0	100	0	50	50	0	0
K	80	21	59	20	13	90	7	0	3
L	55	0	29	71	27	80	15	0	5
M	48	13	21	67	19	97	3	0	0
N	48	17	38	42	8	89	7	0	5

Note that everything except the first column is in percentages

9.1.1 The Measure Matters

Here we have the developer with the *least* **defects** and the developer with the *most* defects. Let's play with **personas** a little bit:

Jesse Jordan (Tester J) found 2 defects, each self-reported as "Normal" faults. They fixed one of them, but couldn't fix the other. The paper points out that Jesse actually resigned during the data collection period, and this is only a partial time series. We don't know what actually happened to Jesse, or why they resigned.

```
data[data$Defects==min(data$Defects),]
```

```
##    Tester Defects Extra.Hot Hot Normal Status.Open Fixed No.Fix Duplicate
## 10      J       2         0   0    100           0    50     50         0
```

```
##    Cannot.reproduce Totals
## 10                0    100
```

Kendall Kennedy (Tester K) found 80 defects. They're still working on 13% of the faults, but for the finished ones they fixed 90%. They weren't able to fix 7% of those, though. They labeled the faults as primarily "Hot", with a few "Normal" and a few "Extra Hot". 3% were just not reproducible at all.

```
data[data$Defects==max(data$Defects),]
```

```
##    Tester Defects Extra.Hot Hot Normal Status.Open Fixed No.Fix Duplicate
## 11      K      80        21  59     20          13    90      7         0
##    Cannot.reproduce Totals
## 11                3    100
```

9.1.2 Do you think we can tell who is performing better from that measure? Surely there is more to the story than just number of defects found.

These lessons are here not only to provide some statistical skills but also to get you thinking about *how to **measure** things*. Towards the end of this lesson, we will also try to devise a system for **grading group work**. Keep that in mind as we operationalize what it means to measure performance.

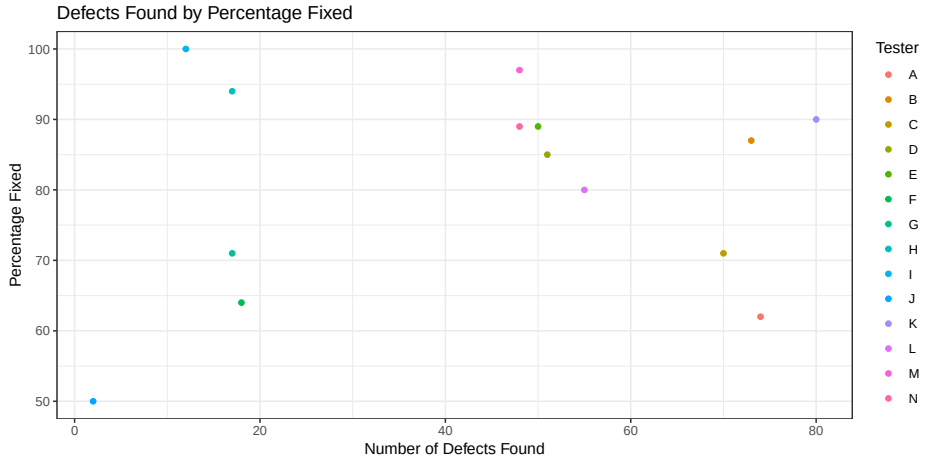
9.2 Weighted Features

Each of the columns represents a *feature of interest*. If you've heard the term "feature engineering" when reading about Big Data or Machine Learning, it's talking about choosing which **factors** to pay attention to. Sometimes we get so nervous about the buzzwords like "Machine Learning" and don't ever get the chance to break down what it all really means. Well, this problem is an introduction to what it might mean to select features and combine them to predict an outcome. Whether it's determining what series you will watch on Netflix, or how well you are performing at your job, this kind of problem persists across all disciplines. It might occur to you that *it's simply not fair* to use such a measure to determine someone's productivity. Most of the time, you'd probably be correct. Our data sources, the people designing the measures, and the people interpreting the results are ill-equipped to account for the tons of diversity in the world. Hopefully now that you're armed with more statistical know-how, you'd be able to better advocate for yourself against harmful models.

While it may not be ideal to use a blanket measure for someone's productivity, it might still be useful to implement different support systems for workers in a company; if someone is struggling to feel productive and accomplished, perhaps we could catch it with some metrics and intervene to help them be the most fulfilled version of themselves.

We discussed that "number of faults found" isn't the best measure of performance in isolation, and we talked about how you need to include other features in combination. But should every feature be **weighted** equally? Or do some things matter more than others? In the performance data we have, we even have the developer ratings of how *serious* the faults were that they found. Let's look at how Jesse and Kendall rated everything.

```
min_and_max <- data[data$Defects==max(data$Defects) | data$Defects==min(data$Defects) ,] #an ove

ggplot(data,aes(Defects,Fixed,color=Tester))+
  geom_point()+
  xlab("Number of Defects Found")+
  ylab("Percentage Fixed")+
  ggtitle("Defects Found by Percentage Fixed")+
  theme_bw()
```

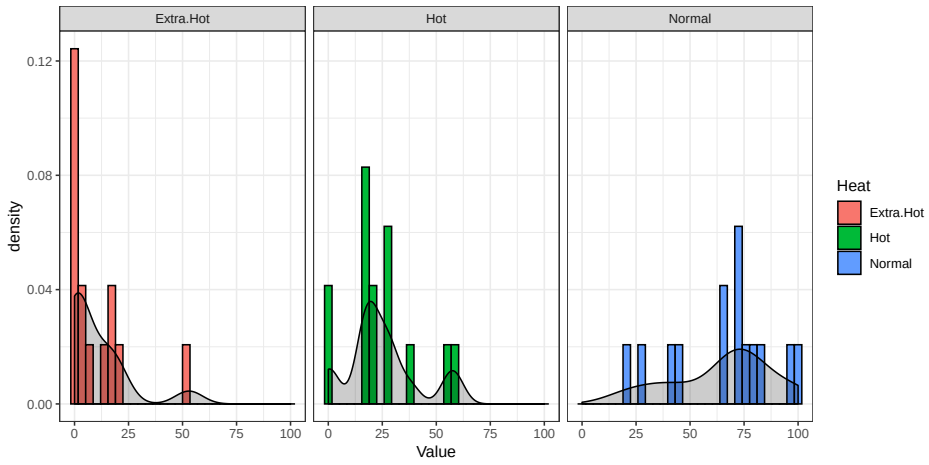



```
cor(data$Defects,data$Fixed) #nada
```

```
## [1] 0.1754979
```

```
#I want the distribution of Extra Hot, Hot, Normal for each tester
library(tidyr)
hot <- gather(data, Heat, Value, c("Extra.Hot","Hot","Normal"), factor_key=TRUE)

ggplot(hot,aes(Value,fill=Heat))+
  facet_wrap(~Heat)+
  geom_histogram(aes(y = ..density..), color="black")+
  geom_density(aes(y = ..density..),color="black",fill="black", alpha=.2,stat = 'density')+
  theme_bw()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



We can see that "Extra Hot" faults occur with a lower frequency (the majority being close to 0), and "Normal" faults are…well, normal. We don't actually have access to how many of the **Fixed** faults were which Heat Level, but you might imagine that fixing an "Extra Hot" should count for more than fixing a "Normal" fault. Keep in mind that these are self-reported Heat levels, as well. One person's "Normal" might be someone else's "Extra Hot". But is there a way to give someone more credit for fixing an "Extra Hot" fault than for fixing a "Normal" one? Or would that simply incentivize developers to label all of their work as "Extra Hot"? Maybe other developers could rate how difficult the faults were, so that people wouldn't game the system. But that would also be really unfair, as someone might have worked *really* hard on something that would take someone else less time. Or another developer may not have the expertise to actually rate the severity of a fault for another developer.

Weighted Features are a foundation in machine learning models. Algorithms can actually systematically infer the proper combination of weights on each feature that most accurately predicts an outcome variable. But this is a complex interaction between algorithm and experimental design. An algorithm can only work with what was **measured**; and sometimes our **operationalization** of what should be measured isn't valid. This is a delicate balance that hopefully comes with experience, but be reminded that you always have the power to question even the most confident statistical result!

TODO: explain **anova**

```
#you can play around with different combos that you think might affect the percentage
#Note: No.Fix will predict Fixed, because they're related by their sum. It's not meani
model <- glm(data$Fixed~data$Defects+data$Status.Open+data$Extra.Hot)
Anova(model) #significant effect of Status.Open and Extra.Hot on percentage Fixed, can
```

```
## Analysis of Deviance Table (Type II tests)
##
```

```
## Response: data$Fixed
##                   LR Chisq Df Pr(>Chisq)
## data$Defects        1.6298  1   0.201735
## data$Status.Open    6.5808  1   0.010308 *
## data$Extra.Hot      9.5049  1   0.002049 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
model <- glm(data$Fixed~data$Defects)
Anova(model) #no significance
```

```
## Analysis of Deviance Table (Type II tests)
##
## Response: data$Fixed
##               LR Chisq Df Pr(>Chisq)
## data$Defects  0.38134  1     0.5369
```

9.3 Investigating the Grant-Sackman Legend

9.3.1 How long do different programmers take to solve the same task? It's not what you think!

As I delve more into the world of software engineering, I'm uncovering countless myths, folklore, and dogma. It seems that software engineers are particularly religious when it comes to their opinions and cited "facts", choosing multiple hills to die on, over and over. This might be part of the engineering culture, where people must consistently argue to prove their value, without a ton of access and experience with academic literature. This is a product of how we value employees, productivity, and "sounding smart" and we are all victim to it. So let's strip back some of the layers of the things we thought we knew, and try to approach these things with an evidence-based mindset. And hey, it might be even more fun to argue when you've got proper statistical skills and citations in your back pocket!

Apparently, the software world loves to cite the "fact" that the difference between the worst programmer and best programmer in a group is about *10-fold*. It's one of those things that is either commonly stated or debated, but nevertheless it persists. Personally, I find it to be yet another excuse to try to rank yourself against others; seeing where you "fit" in the hierarchy of productivity and performance. Unfortunately, no one really wants to be called out as the "worst" programmer, and if you really are the "best" programmer, you've got a million opportunities already! Realistically, we all probably have a distribution of skills that are each useful for different problems. Here's a cheesy, mathematical, motivational comic I drew when I was in my first research job learning Bayesian inference and constantly feeling behind:

Okay so let's investigate the original culprit of this 10-fold measure.

9.3.2 The Study

12 developers participated in 2 programming tasks, each with a `coding` portion and a `debugging` portion. One task was to do some algebra operations, and the other task was to find the proper path through a 20x20 maze. Half of the developers were in an `online` group, and the other half in an `offline` group. They were timed at everything they did. This is the study that is the origin of the famous claim that there's such a big discrepancy between programmers in how fast they can do any given task. This original study came up with a 28:1 ratio between the best to worst programmer times.

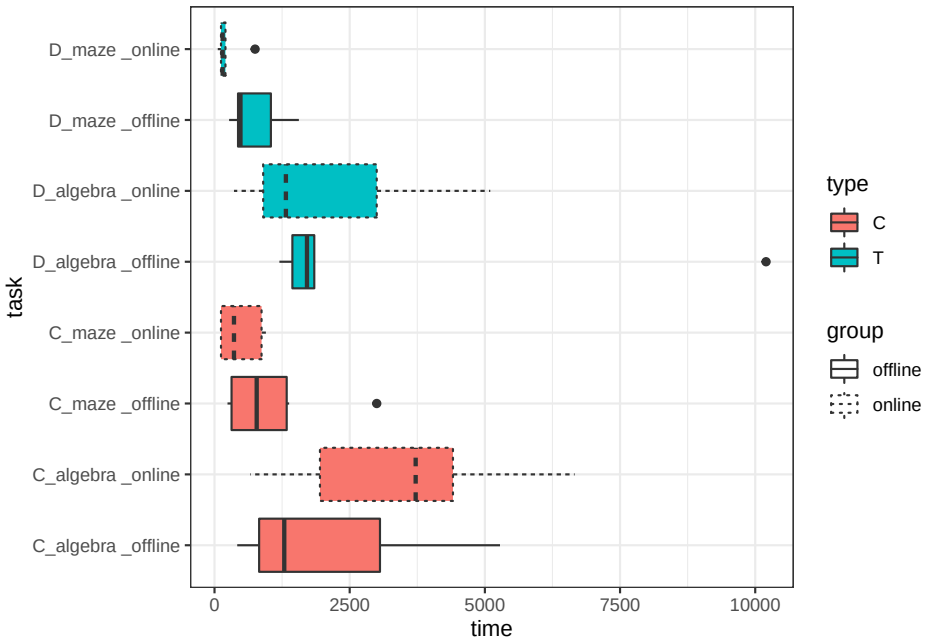
There are three problems with this number:

1. It is wrong.
2. Comparing the best with the worst is inappropriate.
3. One should use more data to make such a claim.

So, those dingbats back in the 70s tried to say that for any given project, theres a 28:1 ratio of time-taken/ability between the worst developer and the best developer, meaning theres tons of variation for any developer and its a crapshoot. They literally did their comparisons incorrectly, comparing groups that could not logically be compared. They also allowed developers to code in different langugaes, with some of the programmers choosing Assembly; no one knows why. The measure, *if anything* should be about 9:1, but the study is riddled with issues. We walk through a few of these issues, and look at that **original data** below:

```
#
# GS-perm-diff.R, 29 Aug 16
# Data from:
# The 28:1 {Grant}/{Sackman} legend is misleading, or: {How} large is interpersonal va
# Lutz Prechelt
#
# Example from:
# Evidence-based Software Engineering: based on the publicly available data
# Derek M. Jones
#
# TAG experiment developer performance


source("data/ESEUR_config.r") # FIXME



gs=read.csv(paste0(ESEUR_dir, "data/grant-sackman.csv.xz"), as.is=TRUE)
gs2 <-gs
gs2$task[gs$group=="online"] <- paste(gs$task[gs$group=="online"],"_online")
```

```
gs2$task[gs$group=="offline"] <- paste(gs$task[gs$group=="offline"],"_offline")

plt = ggplot(gs2,aes(task,time,fill=type,linetype=group))+
  geom_boxplot()+
  coord_flip()+
  theme_bw()
plt
```



9.4 Order Effects

It's important to explain order effects, because they have the potential to seriously mess with experimental results. **Order effects** can sometimes be found in a **within-subjects** experimental design. **Within-subjects** means that each participant does more than one condition. It makes sense to mix around the order that participants see each condition, because of **learning effects**, **priming**, or **fatigue**. Imagine the following scenario:

> First Task: programmers must implement a Tetris game
>
> Second Task: programmers choose between creating a Chess game or making a data visualization

Is it possible that the first task affects the choice in the second task? If they were reversed, would something else happen? Maybe programmers get tired, or get used to creating some graphics and want to continue thinking about

creating games. Or maybe they're so sick of making games that they always choose to make a data visualization next. Order can affect things. It makes the most sense to randomly present the conditions to your participants, to try to "wash out" any effects like that in the final results. If, regardless of Order, participants are always choosing to make a Chess game, perhaps it's just the more interesting option. But sometimes we can have accidental effects of Order that are difficult to anticipate. What if we mix around Order, but for the people who made Tetris first, they *always* choose Chess? But for people who didn't make Tetris first, it's more of a 50-50 toss up? That is an **Order Effect**.

You can test for Order effects by modeling Order as an actual feature in your model. Normally, we simply mix up the Order to avoid any confounding factors. But what if Order is the confound, itself? We want to look at the *weight* that Order is having on our **outcome variable**:

```
#is there an order effect?

model = lm(time ~seq,
           data = gs)


Anova(model,
      type = "II")
```

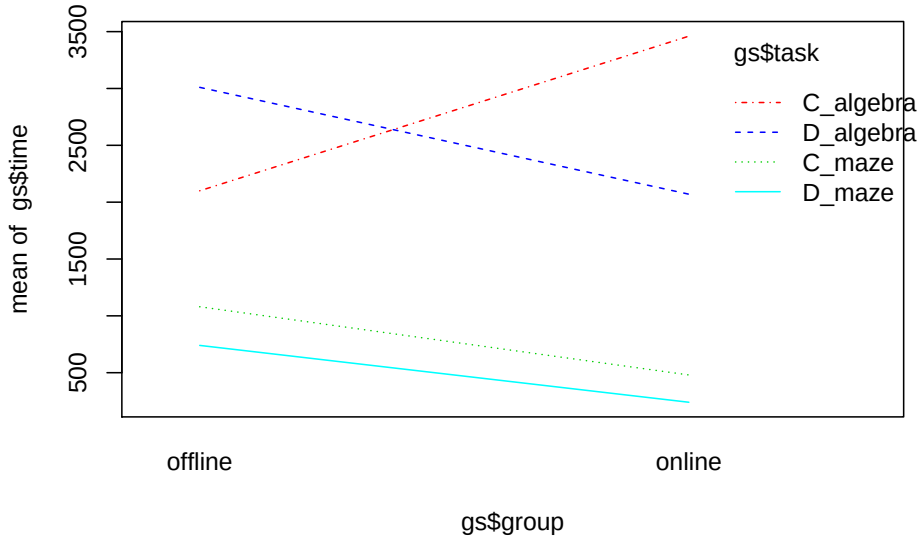
```
## Anova Table (Type II tests)
##
## Response: time
##              Sum Sq Df F value    Pr(>F)
## seq        44686140  1  14.198 0.0004669 ***
## Residuals 144774360 46
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

It turns out that there *is* an effect of Order, which is yet another problem with the commonly parroted Grant-Sackman study.

9.5 Interaction Plots

Turns out we might have another Boogeyman affecting our results. It's called an **interaction**. This is when you have two **categorical factors** that are affecting the outcome variable in contradiction. **Stats Cheatcode: if you plot the variables and any of them cross, you've got an interaction effect**. These aren't necessarily *bad* but they need to be taken into account. This is because taking the means without considering the interacting factor will result in some value somewhere in the middle, when in fact, it's systematically dependent on some other factor you'd be missing.

```
interaction.plot(gs$group,gs$task,gs$time,col=2:5)
```



Here is a little bit tricky but let's try to unpack it. We do see two lines crossing, so we know we have an interaction somewhere. It looks like the crossing pair is between `C_algebra` and `D_algebra`. What does that mean? So, first off, we are comparing **Coding** the algebra task, and **Debugging** the algebra task. There is an interaction across `online` and `offline` (groups of interest). Take a deep breath, don't worry I'm kind of confused too. Okay.

OFFLINE:

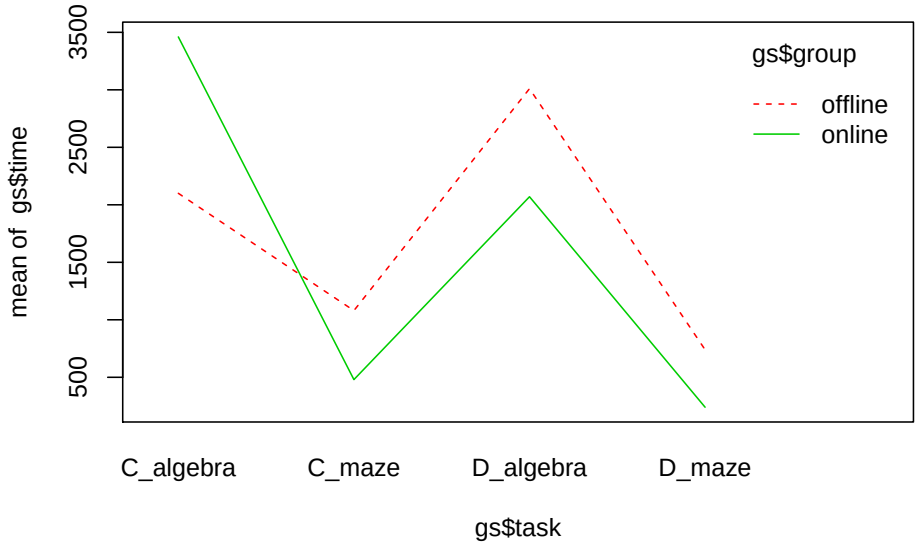
- D_algebra: `mean = 3010`
- C_algebra: `mean = 2100`

ONLINE:

- D_algebra: `mean = 2070`
- C_algebra: `mean = 3460`

So that means that if you were in the `offline` condition, you **debug slower than you code**. But if you were in the `online` condition, you **code slower than you debug**. That's not a meaningless result! It was a little bit tricky to get to, but that's actually something to pay attention to. It's also weird that it *didn't* happen for the `maze` task. So something funky is going on, and these are the kind of hidden disaster-storms you can find in your data and *need to account for*.

```
interaction.plot(gs$task,gs$group,gs$time,col=2:3)
```



Here, we have another crossing in the interaction plot. Let's break it down like we did in the last one.

CODING ALGEBRA:

- offline: `mean = 2100`
- online: `mean = 3460`

CODING MAZE:

- offline: `mean = 1080`
- online: `mean = 480`

So this means that for coding the maze task, `offline` coding was slower than `online` coding, but for the algebra task, `offline` coding was faster than `online` coding. You can try to come up with what that means about the tasks themselves (also noting that the maze task took less time overall), but what I'm trying to point out is that these are the things that a responsible statitician *must* look into when they have data they want to make claims about. Certain interactions can actually invalidate the entire experiment. And if you take nothing else away, just remember: *parallel = move along, crossed = something's wrong!* I just made that up, but it's probably good.

```
summary <- gs %>%
  group_by(task,group) %>%
  summarise(mean=mean(time),min=min(time),max=max(time),ratio=max(time)/min(time))
kable(summary)%>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

task	group	mean	min	max	ratio
C_algebra	offline	2100	420	5280	12.571429
C_algebra	online	3460	660	6660	10.090909
C_maze	offline	1080	240	3000	12.500000
C_maze	online	480	120	960	8.000000
D_algebra	offline	3010	1200	10200	8.500000
D_algebra	online	2070	360	5100	14.166667
D_maze	offline	740	270	1560	5.777778
D_maze	online	240	60	750	12.500000

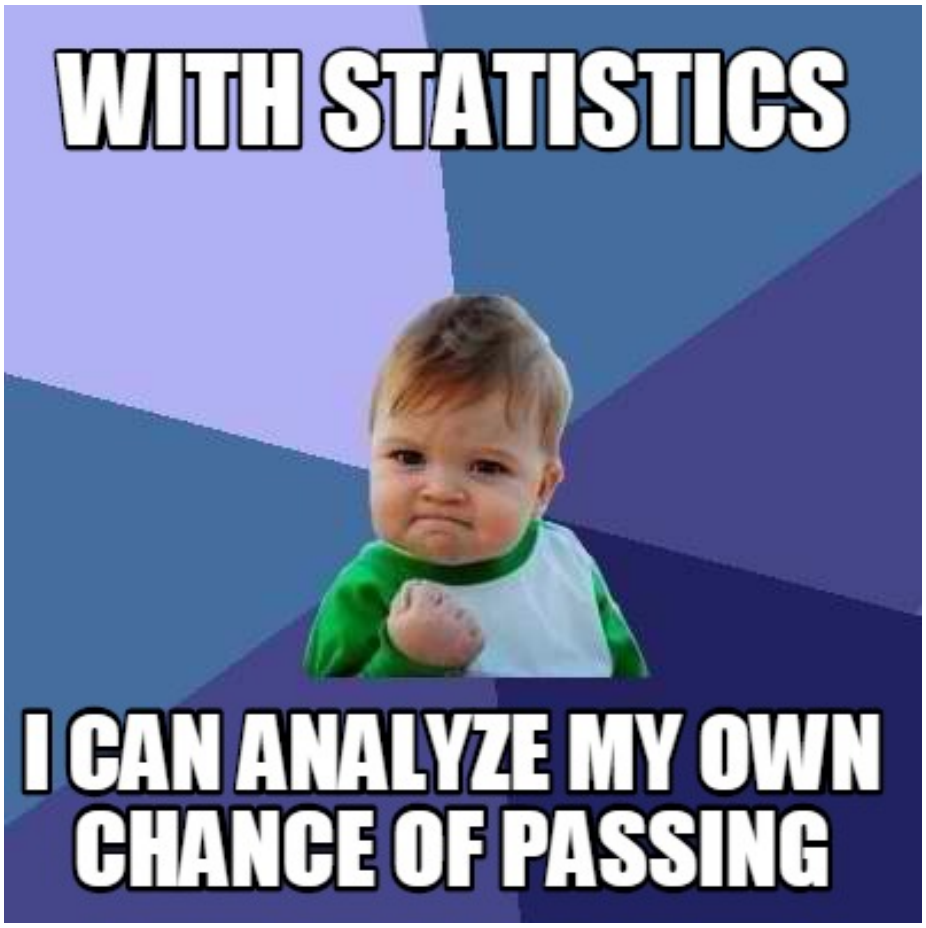
9.6 Back to Basics: How to Measure Performance at all

https://catenary.wordpress.com/2011/01/12/the-thorny-and-the-obvious/

The problem is not necessarily in how rigorously we are measuring performance; though having more participants and less confounding variables is a must! The real problem lies more in the idea that we are measuring performance at all; without an agreed-upon definition of what that means and how it helps. Several companies use performance metrics to monitor employees, give bonuses, or terminate employment. But given all of the statistical caveats we have looked at in this lesson, how can we really be sure that those measures aren't more harmful than helpful? Your next task will be to design a system for measuring group performance for a project. Make sure to provide argumentation for why your measure is valid, reasonable, and helpful.

9.7 In Your Ideal World...

9.7.1 How should group projects be graded fairly?



- define your outcome variables
- define factors affecting the outcome variables
- define the weighting system
- define how the information will be combined to give a final "score"
- justify your design

Chapter 10

Function Modification

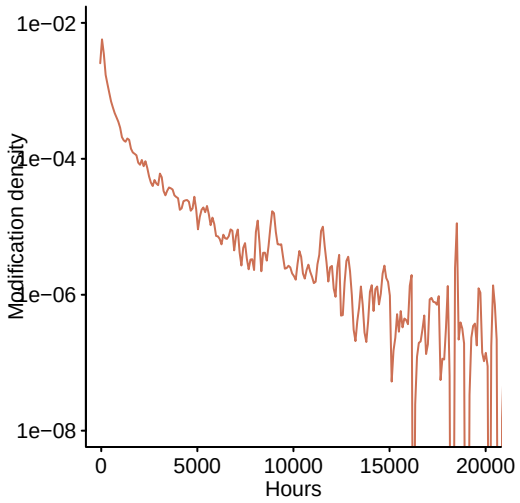
Modification and Developer Metrics at the Function Level: Metrics for the Study of the Evolution of a Software Project

https://github.com/Derek-Jones/ESEUR-code-data/blob/master/evolution/functions/num-func-mods.R

```
#
# num-func-mods.R, 22 Dec 15
#
# Data from:
# Modification and developer metrics at the function level: Metrics for the study of the evolutio
#
# Example from:
# Empirical Software Engineering using R
# Derek M. Jones

source("data/ESEUR_config.r") # FIXME

funcs=read.csv(paste0(ESEUR_dir, "data/ev_funcmod.tsv.xz"), as.is=TRUE, sep="\t")

revdate=read.csv(paste0(ESEUR_dir, "data/ev_rev_date.csv.xz"), as.is=TRUE)
#revdate=rbind(revdate, c(NA, NA))

revdate$date_time=as.POSIXct(revdate$date_time, format="%Y-%m-%d %H:%M:%S")

# Assign date-time for a given revid
funcs$revdate=revdate$date_time[funcs$revid]
funcs$prevdate=revdate$date_time[funcs$revprev]

time_between=funcs$revdate-funcs$prevdate
```

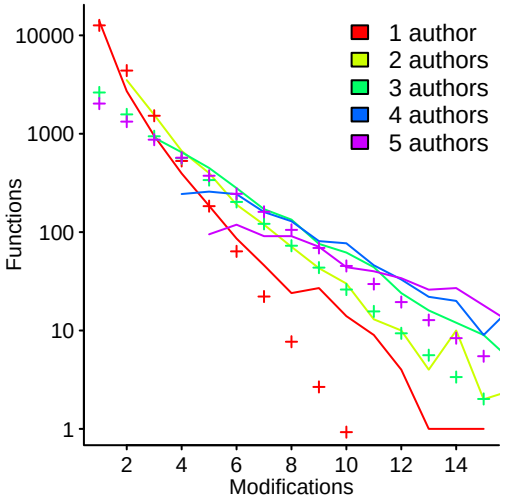
```
q=density(as.numeric(time_between)/(60*60), adjust=0.5, na.rm=TRUE)
plot(q, log="y", col=point_col,
    main="",
    xlab="Hours", ylab="Modification density\n",
    xlim=c(2, 20000), ylim=c(1e-8,1e-2))
```

```
## Warning in xy.coords(x, y, xlabel, ylabel, log): 69 y values <= 0 omitted
## from logarithmic plot
```




```
#revdate=read.csv(paste0(ESEUR_dir, "evolution/functions/ev_rev_date.csv.xz"), as.is=T

# Assign date-time for a given revid
#funcs$date=revdate$date_time[funcs$revid %in% revdate$revid]


count_mods=function(df)
{
# Only count additions and modifications
df=subset(df, typemod != "D")
num_mods=nrow(df)
num_authors=length(unique(df$author))

return(cbind(num_mods, num_authors))
}


mod_count=ddply(funcs, .(filename, func_name), count_mods)
total_mods=ddply(mod_count, .(num_mods, num_authors), nrow)
```

```
plot_mods=function(X)
{
t=subset(total_mods, num_authors == X)
lines(t$num_mods, t$V1, col=pal_col[X])
}


xbounds=c(1, 15)
ybounds=c(1, max(total_mods$V1))

plot(1, 1, type="n", log="y",
    xlab="Modifications", ylab="Functions\n",
    xlim=xbounds, ylim=ybounds)
pal_col=rainbow(5)
dummy=sapply(1:5, plot_mods)

legend(x="topright",legend=c("1 author", "2 authors", "3 authors", "4 authors", "5 authors"),
                bty="n", fill=pal_col, cex=1.3)


ma_mod=glm(V1 ~ I(log(num_authors)^0.4)*num_mods,
        data=total_mods, family=poisson)

pred1=predict(ma_mod, data.frame(num_authors=1, num_mods=1:15),
        type="response")
points(pred1, col=pal_col[1])
pred3=predict(ma_mod, data.frame(num_authors=3, num_mods=1:15),
        type="response")
points(pred3, col=pal_col[3])
pred5=predict(ma_mod, data.frame(num_authors=5, num_mods=1:15),
        type="response")
points(pred5, col=pal_col[5])
```



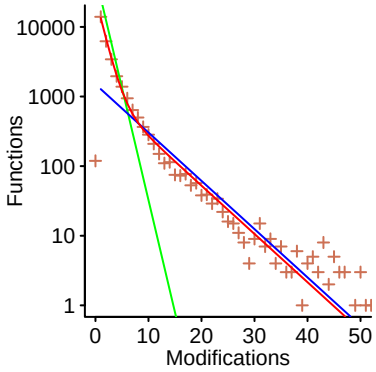

```
# a1=subset(total_mods, num_authors == 3)
# ma_mod=nls(V1 ~ a*exp(c*num_mods),
#        data=a1, trace=TRUE,
#        start=list(a=3000, c=-0.3))
```

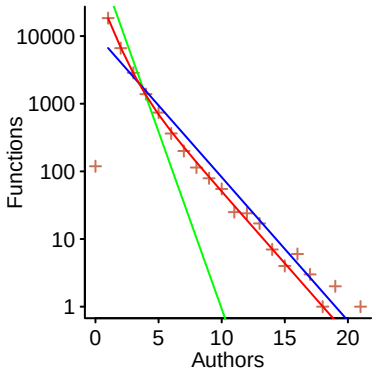
```
#
# author-mod-func.R, 25 Mar 18
#
# Data from:
# Modification and developer metrics at the function level: Metrics for the study of t
# Gregorio Robles and Israel Herraiz and Daniel M. Germ{\'a}n and Daniel Izquierdo-Cor
#
# Example from:
# Empirical Software Engineering using R
# Derek M. Jones



# The igraph package (which might be loaded when building the book)
# contains functions found in gnm.  The treemap package might also have
# been loaded, and its 'load' of igraph cannot be undone without first
# unloading treemap!
unloadNamespace("treemap")
unloadNamespace("igraph")

library("gnm")
library("plyr")

pal_col=rainbow(3)
plot_layout(2, 1)
```

```
# Investigate how many times files might be moved
merge_move=function(df)
{
deleted=subset(df, typemod == "D")
added=subset(df, typemod == "A")

return (cbind(nrow(deleted), nrow(added)))
}
# all_moves=ddply(funcs, .(func_name), merge_move)
# table(all_moves[, 2:3])
# Number of functions with the same name that are
# deleted (row) and added (column)
#       2
# 1        0    1    2    3    4    5    6    7
#   0      0 9152  286  111   31   12    1    2
#   1    107 9659 1075   74   17    9    5    1
#   2      3  474 1714  140   20   18    9    3
#   3      0    9   88  322   67   15    9    1
#   4      0    0   10   10   80   18    8    2
#   5      0    0    0   26    6   33    7    1
#
# 1 delete + 2 add could be a move: 1075 of them...


# Count number of changes and authors for a given function
count_mods=function(df)
{
# Only count additions and modifications
df=subset(df, typemod != "D")
num_mods=nrow(df)
num_authors=length(unique(df$author))

return(cbind(num_mods, num_authors))
}


# Two files may have the same name
mod_count=ddply(funcs, .(filename, func_name), count_mods)
total_mods=ddply(mod_count, .(num_mods, num_authors), nrow)


all_mods=ddply(total_mods, .(num_mods), function(df) sum(df$V1))
```

```
plot(all_mods$num_mods, all_mods$V1, log="y", col=point_col,
    xlim=c(0, 50),
    xlab="Modifications", ylab="Functions\n")

# a_mod=glm(V1 ~ num_mods+I(num_mods^2)+I(num_mods^3), data=all_mods[-1, ], family=poi
# a_mod=glm(V1 ~ num_mods, data=all_mods[-1, ], family=poisson)
# lines(1:50, exp(predict(a_mod, newdata=data.frame(num_mods=1:50), type="link")), col

a2_mod=gnm(V1 ~ instances(Mult(1, Exp(num_mods)), 2)-1,
               data=all_mods[-1,], verbose=FALSE,
               start=c(20000.0, -0.6, 300.0, -0.1),
               family=poisson(link="identity"))
exp_coef=as.numeric(coef(a2_mod))

lines(exp_coef[1]*exp(exp_coef[2]*all_mods$num_mods), col=pal_col[2])
lines(exp_coef[3]*exp(exp_coef[4]*all_mods$num_mods), col=pal_col[3])
t=predict(a2_mod)
lines(t, col=pal_col[1])

s1=exp(a2_mod$coef[2])
D_I_1=(1-s1)^2/s1

s2=exp(a2_mod$coef[4])
D_I_2=(1-s2)^2/s2

# c(s1, D_I_1, s2, D_I_2)


author_mods=ddply(total_mods, .(num_authors), function(df) sum(df$V1))

plot(author_mods$num_authors, author_mods$V1, log="y", col=point_col,
    xlab="Authors", ylab="Functions\n")

# a1_authors=glm(V1 ~ num_authors, data=author_mods[-1, ], family=poisson)
# lines(1:15, exp(predict(a1_authors, newdata=data.frame(num_authors=1:15), type="link
# a2_authors=glm(V1 ~ num_authors+I(num_authors^2), data=author_mods[-1, ], family=poi
# lines(1:20, exp(predict(a2_authors, newdata=data.frame(num_authors=1:20), type="link

a2_mod=gnm(V1 ~ instances(Mult(1, Exp(num_authors)), 2)-1,
               data=author_mods[-1,], verbose=FALSE,
               start=c(20000.0, -0.6, 300.0, -0.1),
               family=poisson(link="identity"))
exp_coef=as.numeric(coef(a2_mod))

lines(exp_coef[1]*exp(exp_coef[2]*author_mods$num_authors), col=pal_col[2])
```

```
lines(exp_coef[3]*exp(exp_coef[4]*author_mods$num_authors), col=pal_col[3])
t=predict(a2_mod)
lines(t, col=pal_col[1])
```






```
s1=exp(a2_mod$coef[2])
D_I_1=(1-s1)^2/s1

s2=exp(a2_mod$coef[4])
D_I_2=(1-s2)^2/s2

# c(s1, D_I_1, s2, D_I_2)
```

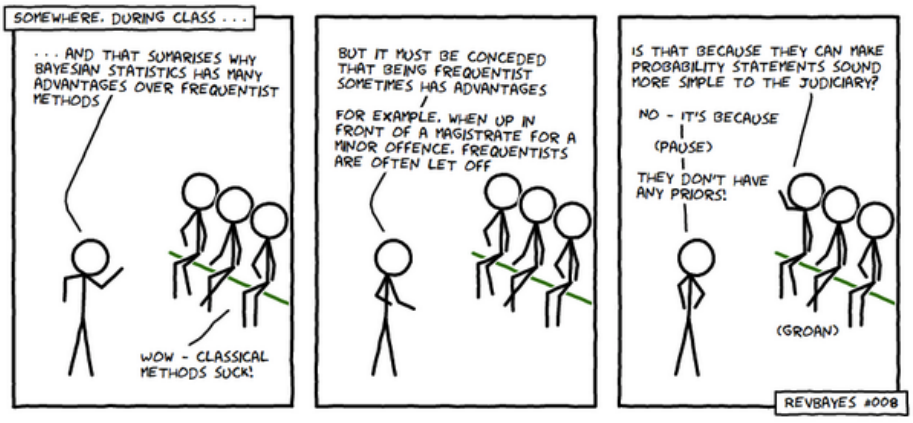
10.1 Figure 4.50: Number of functions

(in Evolution; the point at zero are incorrect counts) modified a given number of times (upper) or modified by a given number of different people (lower); red line is a bi-exponential fit, green/blue lines are the individual exponentials

Chapter 11

Bayesian vs Frequentist Statistics

So far, we have been using frequentist statistics; relying on hypothesis testing and *p-values*. Frequentist statistics were particularly popular when computational resources were scarce; but this is no longer the case. It is time to reexamine the benefits of applying Bayesian statistics to software engineering empirical work. If you've been lucky enough to never have been caught in the crossfire between a Bayesian and a Frequentist, then hopefully this lesson will go smoothly for you. But if you've firmly planted your flag on either side of the "debate", take a deep breath. This lesson merely demonstrates some of the benefits of using a Bayesian approach, while also pointing out that sometimes both approaches tend to result in functionally the same courses of action. Hopefully by the end, you'll feel much more prepared to chime in if a raging Frequentist and an exasperated Bayesian walk into a bar.



Bayesian Statistics in Software Engineering: Practical Guide and Case Studies

Researcher-Centered Design of Statistics: Why Bayesian Statistics Better Fit the Culture and Incentives of HCI

11.1 Frequentist Statistics

Let's try to concisely summarize what we have been doing so far, as we obtain *p-values* and make conclusions about *hypotheses*. Using frequentist methods, we have been comparing groups in our data to determine the probability that those groups were drawn from the same underlying data-driven process. In basic terms, we have been figuring out if the groups are truly different or not. Because we live in a messy world, even samples from the exact same source will differ a little bit. Imagine taking a sample of algae from a pond, or collecting everyone's feelings on a random Monday. If you were to do the same sample again, under the same circumstances, you'd still get slightly different results. The point of hypothesis testing is to make claims *in general* about the samples. A small p-value (less than .05) indicates that there is a low probability that the two groups are actually the same, whereas a larger p-value indicates a higher probability that the two groups are actually the same (no difference). We have seen this in several of our lessons so far, and you will come across these methods in most empirical software engineering papers.

11.2 Bayesian Statistics

11.3 Original Study

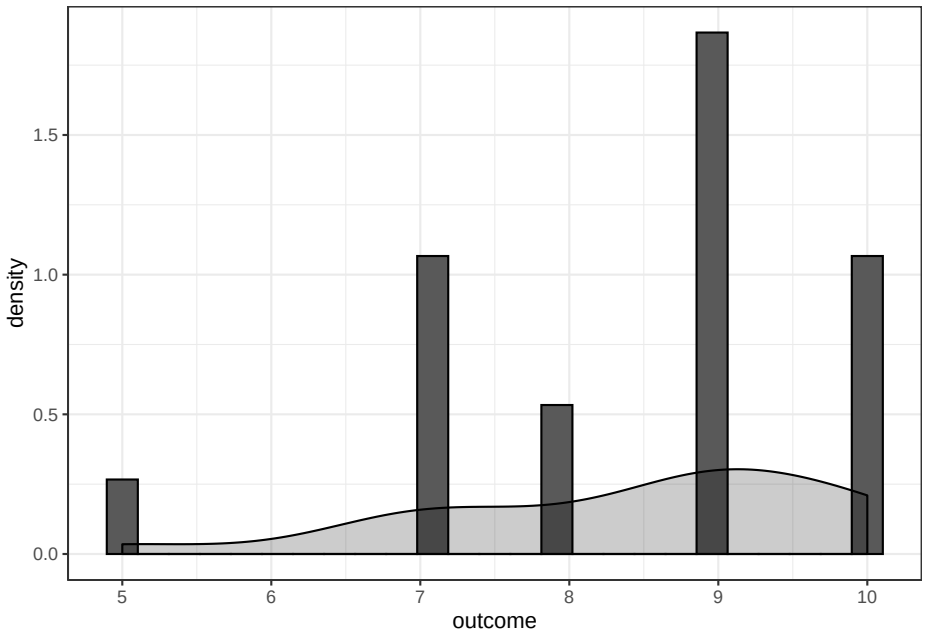
they use a U test

```
library(tidyverse)
data <- read.csv("data/agile/survey.csv")
head(data)
```

```
##         type group outcome stakeholders       time
## 1 Structured     0       9            9 Not at all
## 2      Agile     0       9            9   A little
## 3      Agile     0       7            8 Not at all
## 4      Agile     0      10           10 Not at all
## 5 Structured     0      10           10   A little
## 6 Structured     0       7            8   A little
```

```
plt <- ggplot(data[data$type=="Structured",],aes(outcome)) +
  geom_histogram(aes(y = ..density..), bins=25,color="black")+
  geom_density(aes(y = ..density..),color="black",fill="black", alpha=.2,stat = 'densit
  theme_bw()
```

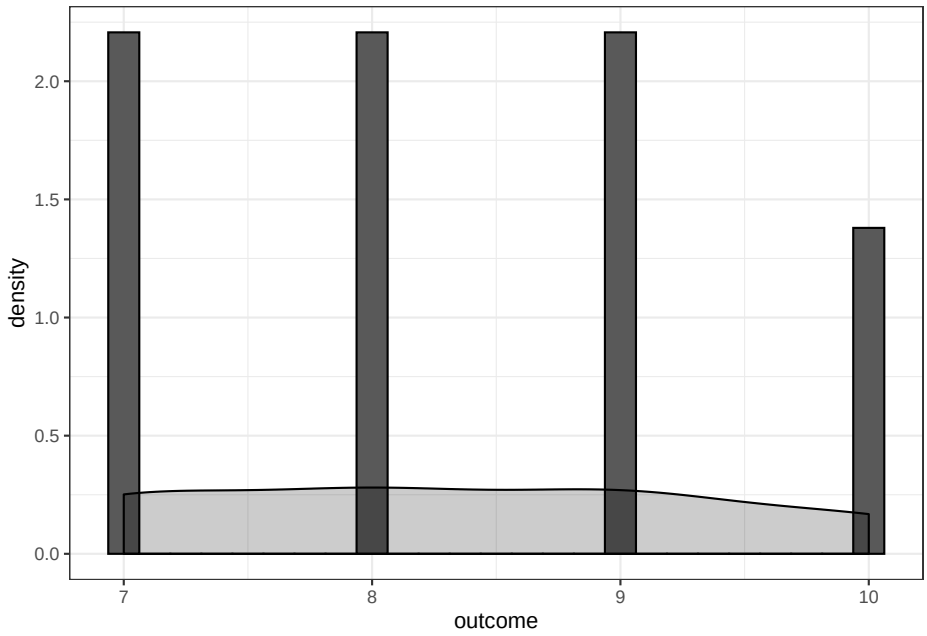
```
plt
```




```
shapiro.test(data$outcome[data$type=="Structured"])
```

```
##
##  Shapiro-Wilk normality test
##
## data:  data$outcome[data$type == "Structured"]
## W = 0.86996, p-value = 0.01777
```

```
plt <- ggplot(data[data$type=="Agile",],aes(outcome)) +
  geom_histogram(aes(y = ..density..), bins=25,color="black")+
  geom_density(aes(y = ..density..),color="black",fill="black", alpha=.2,stat = 'density')+
  theme_bw()

plt
```




```
shapiro.test(data$outcome[data$type=="Agile"])
```

```
##
##  Shapiro-Wilk normality test
##
## data:  data$outcome[data$type == "Agile"]
## W = 0.86798, p-value = 0.001824
```

```
#HERE IS THE U TEST THEY DID
wilcox.test(data$outcome[data$type=="Agile"],data$outcome[data$type=="Structured"])
```

```
## Warning in wilcox.test.default(data$outcome[data$type == "Agile"],
## data$outcome[data$type == : cannot compute exact p-value with ties
```

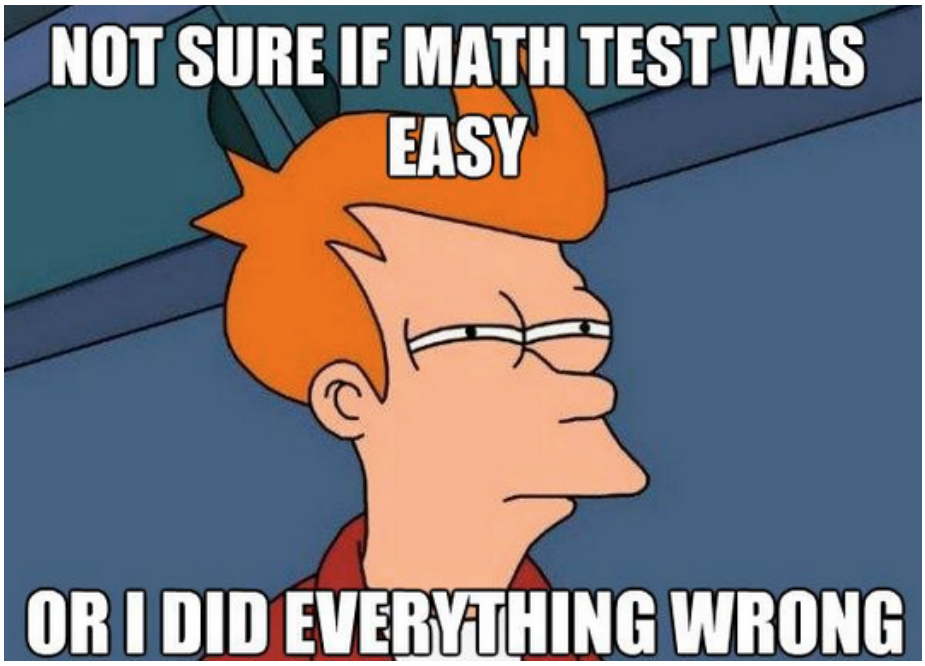
```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  data$outcome[data$type == "Agile"] and data$outcome[data$type == "Structured"
## W = 236, p-value = 0.5792
## alternative hypothesis: true location shift is not equal to 0
```

```
# larger scale study they ran, we will do the bayesian analysis here
itp <- read.csv("data/agile/itproj.csv")
head(itp)
```

```
##   type group percent    success  challenge    failure           time
## 1    H     0   1-10%       None     81-90%     11-20%        Neutral
## 2    H     0   1-10% Don't Know Don't Know Don't Know Not Applicable
```

```
## 3    H     0  31-40%     11-20%     81-90%       None    Ineffective
## 4    H     0  81-90%     71-80%     71-80%     21-30% Not Applicable
## 5    H     0  61-70% Don't Know Don't Know Don't Know        Neutral
## 6    H     0   1-10%      1-10%     81-90%      1-10%      Effective
##              ROI   stakeholders          quality
## 1        Neutral        Neutral        Effective
## 2 Not Applicable Not Applicable   Not Applicable
## 3        Neutral        Neutral Very Ineffective
## 4 Not Applicable Not Applicable   Not Applicable
## 5        Neutral Very Effective   Very Effective
## 6        Neutral    Ineffective Very Ineffective
```

11.4 Mathematical Peacocking, stop it!



While we work through this paper, it is important to remember that the entire point of academic research is scientific findings and communication; if the paper is riddled with equations and uninterpretable variable names, then *it is bad communication*. It has absolutely nothing to do with the reader's intelligence or ability to grasp that material. It is the author's responsibility to carefully communicate problems, methods, and results, while remembering that the purpose of the work is to be distributed. As we approach the problem of **Agile vs. Structured Development**, I will continuously translate the infuriating mathematical peacocking that is meant to obscure actual communication and build up an inflated ego of an entire research field who use LaTeX expressions

to make the reader feel inferior.

```
itp <- read.csv("data/agile/itproj.csv")
head(itp)
```

```
##   type group percent    success  challenge    failure           time
## 1    H     0   1-10%       None     81-90%     11-20%        Neutral
## 2    H     0   1-10% Don't Know Don't Know Don't Know Not Applicable
## 3    H     0  31-40%     11-20%     81-90%       None    Ineffective
## 4    H     0  81-90%     71-80%     71-80%     21-30% Not Applicable
## 5    H     0  61-70% Don't Know Don't Know Don't Know        Neutral
## 6    H     0   1-10%      1-10%     81-90%      1-10%      Effective
##              ROI   stakeholders          quality
## 1        Neutral        Neutral        Effective
## 2 Not Applicable Not Applicable   Not Applicable
## 3        Neutral        Neutral Very Ineffective
## 4 Not Applicable Not Applicable   Not Applicable
## 5        Neutral Very Effective   Very Effective
## 6        Neutral    Ineffective Very Ineffective
```

```
levels <- levels(itp$success)
replace <- c("",1,11,21,31,41,51,61,71,81,91,NA,NA)

for( i in 1:length(replace)){
  itp <- data.frame(lapply(itp, function(x) {
                  gsub(levels[i], replace[i], x)
              }))
}

itp$success <- as.numeric(as.character(itp$success))
itp$failure <- as.numeric(as.character(itp$failure))
itp$challenge <- as.numeric(as.character(itp$challenge))
summary <-data.frame()

for (i in levels(itp$type)){
  print(i)
  summary <-itp[itp$type==i,] %>%
  summarize(type=i,mean_fail = mean(na.omit(failure)),mean_chal = mean(na.omit(challen
 bind_rows(.,summary)
}
```

```
## [1] "A"
## [1] "H"
## [1] "I"
## [1] "L"
## [1] "T"
```

```
wilcox.test(itp$success[itp$type=="A"],itp$success[itp$type=="T"])
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  itp$success[itp$type == "A"] and itp$success[itp$type == "T"]
## W = 1908.5, p-value = 0.001021
## alternative hypothesis: true location shift is not equal to 0
```

```
wilcox.test(itp$failure[itp$type=="A"],itp$failure[itp$type=="T"])
```

```
## Warning in wilcox.test.default(itp$failure[itp$type == "A"],
## itp$failure[itp$type == : cannot compute exact p-value with ties
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  itp$failure[itp$type == "A"] and itp$failure[itp$type == "T"]
## W = 345, p-value = 0.01843
## alternative hypothesis: true location shift is not equal to 0
```

```
wilcox.test(itp$challenge[itp$type=="A"],itp$challenge[itp$type=="T"])
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  itp$challenge[itp$type == "A"] and itp$challenge[itp$type == "T"]
## W = 1196, p-value = 0.7509
## alternative hypothesis: true location shift is not equal to 0
```

Chapter 12

Code Quality and Lines of Code

12.0.1 Redundancy?

- Notes for me:
- Landman suggests not redundant enough to chuck the CC metric
- The correlation between aggregated CC for all subroutines and the total SLOC of a file is higher than the correlation between CC and SLOC of individual subroutines

12.1 How Does Complexity Relate to Quality?

FIXME

12.2 Cycolmatic Complexity

12.3 "Big Data"! Lines of Code vs. Function Declarations

We saw in our last exercise that we could use ASTs to explore the components of a program. That was a helpful exercise for learning about how we employ tools to collect data from programs on GitHub. However, we only collected a few hundred programs; and if you recall, they were all implementations of data structures and programming concepts in isolation. We know that from our *sample*, our results may not generalize to production code. This means we are lacking something called **external validity**: results of a study can realiably generalize to other contexts/samples (within reason).

Now, you've probably heard this before when discussing concepts like Big O: it only really starts to matter for performance when n is big enough. I know I've certainly questioned why *O(log(n))* vs. *O(n)* even matters when my little sorting snippet runs just fine whatever the algorithm. But here is a big concept: **Complexity matters for big data, and the important findings may be undetectable on toy problems**.

Landman et al. also investigated this problem, using *millions* of Java and C programs.

FIXME: They're actually different problems. I have to figure out how to tie them together: our problem looks at functions and lines, theirs looks at number of lines per function, and also looks at CC.

```
# lang,cc,sloc

source("data/ESEUR_config.r") # FIXME

library("plyr")
cc_loc=read.csv(paste0(ESEUR_dir, "data/Landman_m_ccsloc.csv.xz"), as.is=TRUE)
cc_loc=subset(cc_loc, cc != 0)
cc_loc$logloc=log(cc_loc$sloc)

C_loc=subset(cc_loc, lang == "C")
Java_loc=subset(cc_loc, lang == "Java")
```

Chapter 13

Number of Lines versus Number of Functions in Different Languages

13.1 "Ugh, why do we have to write this in *that* language?"

Whether you're groaning over having to translate your code into some other **language**, or arguing profusely over the benefits of some obscure **package**, it's likely that you've considered the differences between programming languages before. Perhaps you're such a language switcher that your semicolons get mixed up with your whitespace and you search for way too long for the `switch` statement that doesn't exist in Python. Perhaps you're committed to your language and never touching anything else, because yours is *clearly* superior. But whatever the case, we are going to explore some differences between languages today.

13.2 What does it mean to be "different"?

One of the most common statistical tools is to compare **samples** in order to detect a **significant difference**. This might mean detecting a difference in their averages, or more formally, detecting that they are drawn from truly different generating processes in the world. More on that: whenever we draw a sample, we are trying to capture a sample of some phenomenon happening; there is some process (that we can't quite know) that dictates how that data turns out. In this example. it could be that there is something inherent about Python vs. JavaScript interpreters that make them actually different. Or perhaps it could be that Python *users* are different from JavaScript *users*, with one tending to be more verbose than the other. We are trying to draw samples in order to capture patterns in the world, and those patterns come from some **causal** relationship. With comparisons, we really *cannot* determine causality, but this is to get us thinking about the "why" when we do see a difference afterall. If we see a difference in average number of lines for Python vs. JavaScript, is there something inherent about writing Python code that lends nicely to one liners? Or is it the population of Python users? Or is it the specific Python code we grabbed? These are all questions we need to think about when we hear someone say "Oh please, use this language instead! It's way better at… blah blah blah." So let's get to the bottom of those "blah blah blah" claims, shall we?

13.3 How does the number of lines relate to the number of function definitions?

You're faced with our question: How does the number of lines of code compare with the number of **function definitions** between languages? Immediately, I think about the many times I've seen code that copy-and-pastes the same block of code over and over, without declaring it as a function. (*Remember your coding etiquette, people! If you copy and paste 3 times, it's time for a function.*) But I'm also reminded of the massive code libaries, containing a function for *everything*, with hundreds of lines of code for all of those functions. Try to reflect on your own intuition about if the number of lines has a relationship to the number of function declarations in a program. Maybe take a look at a program you've written: how many lines? how many function declarations? And did it change over the duration of your first programming course? (I personally went from copy-and-pasting a hard-coded mess into neat little helper functions as I had to face my own monsterous and unmaintainable code.) But wouldn't that *shorten* my program length? You can imagine that the larger the program, the more function definitions there are. But you can also imagine just the opposite.

13.4 How do I parse a programming language? Abstract Syntax Trees

We need to parse a program in order to count up the number of function definitions. The most straightforward way of doing this might be to look for the "def" keyword in a language like Python. Imagine you were writing this yourself; trying to count the number of function definitions in a piece of code. You might rely on good old fashioned string matching: `for line in program: if "def" in line: function_count +=1` That's a fine start but it gets very messy, *fast*. Luckily, we can rely on something called an **Abstract Syntax Tree**. In this example, we use the `ast` package for Python and `acorn.js` for JavaScript. *TODO: lobstr for R* These parsers give us easy access to the different components of a program, and help us to count them up easier. The example scripts are located here (Python) and here (JavaScript).

13.5 Load Your Libraries

```
library(ggplot2)
library(dplyr)
```

13.6 Read in Your Data

```
py = read.csv("data/PythonFiles/Stats/python_stats.csv")
colnames(py) <- c("Lines","FunctionDefs")
length(py$Lines)
```

```
## [1] 238
```

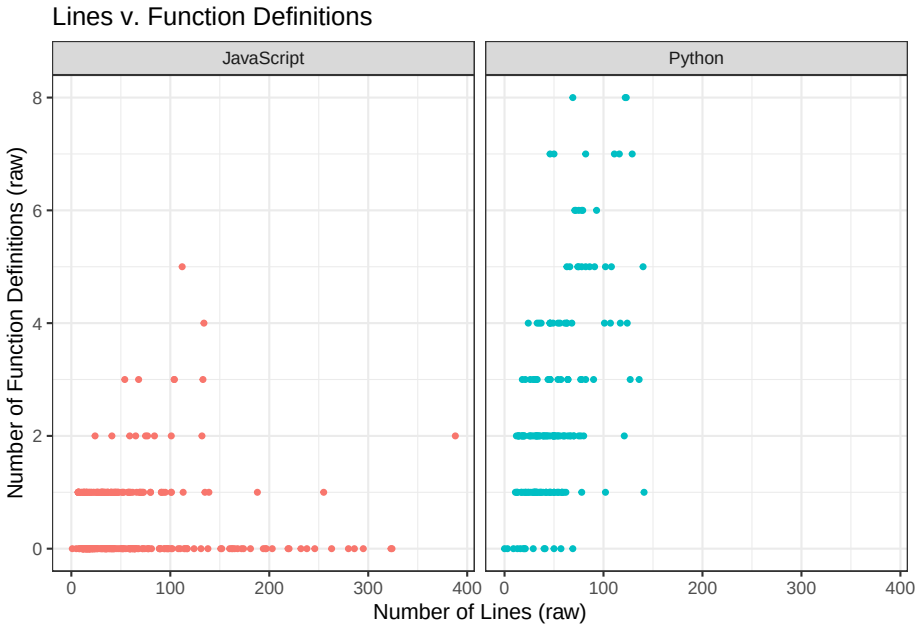
```
#240 python programs
py$Language = "Python"

# JS programs
js = read.csv("data/JSFiles/Stats/js_stats.csv")
colnames(js) <- c("Lines","FunctionDefs")
js$Language= "JavaScript"

data = rbind(py,js)
# some programs must not have been finished yet from the repository of programs?
#data = data[data$Lines>0,]
#data = data[data$FunctionDefs>0,]

# we have a few outliers in our data that skew things
out_js.lines <- boxplot(data$Lines[data$Language=="JavaScript"], plot=FALSE)$out
out_js.funcs <- boxplot(data$FunctionDefs[data$Language=="JavaScript"], plot=FALSE)$out
```

```
out_py.lines <- boxplot(data$Lines[data$Language=="Python"], plot=FALSE)$out
out_py.funcs <- boxplot(data$FunctionDefs[data$Language=="Python"], plot=FALSE)$out


data <- data[-which(data$Lines[data$Language=="JavaScript"] %in% out_js.lines),]
data <- data[-which(data$FunctionDefs[data$Language=="JavaScript"] %in% out_js.funcs),]
data <- data[-which(data$Lines[data$Language=="Python"] %in% out_py.lines),]
data <- data[-which(data$FunctionDefs[data$Language=="Python"] %in% out_py.funcs),]
```

13.7 Don't Compare Apples to Oranges

You may notice when we try to compare the languages, that it's very hard to see what's going on with the Python programs as opposed to the JavaScript programs. That's because we have entirely different scales; The JavaScript AST traces all of the callback functions, resulting in a mean of 55 functions per program, while Python has 3 functions per program. It could also be that our sample is simply not as comparable as we thought. The JavaScript files are *significantly* longer than the Python programs (*JS mean lines = 2646, Python mean lines = 69*). Not only are they difficult to visualize side-by-side, it is an actual statistical issue if we were to compare them! Our samples are just too different. But we are not entirely out of luck. We get to use **normalization** to make our apples (JS) and oranges (Python) more comparable.

```
plt = ggplot(data,aes(Lines,FunctionDefs))+
  facet_wrap(~Language)+
  geom_point(aes(colour=Language),size=1, show.legend=FALSE)+
  #geom_vline(xintercept=mean(data$Lines),colour="red",linetype="dashed")+
  #geom_hline(yintercept=mean(data$FunctionDefs),colour="red",linetype="dashed")+
  ggtitle("Lines v. Function Definitions")+
  xlab("Number of Lines (raw)")+
  ylab("Number of Function Definitions (raw)")+

  theme_bw()

plt
```




```
cor(data$Lines[data$Language=="JavaScript"],data$FunctionDefs[data$Language=="JavaScript"])
```

```
## [1] 0.0218202
```

```
cor(data$Lines[data$Language=="Python"],data$FunctionDefs[data$Language=="Python"])
```

```
## [1] 0.5789447
```

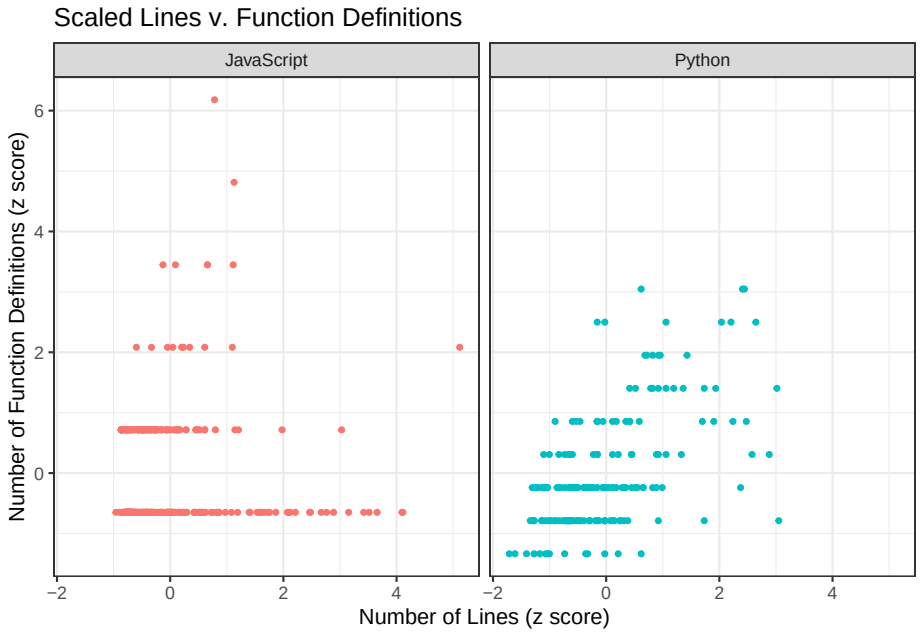
```
#take a look at the scales between the two languages
data %>%
  group_by(Language) %>%
  summarise(mean(Lines),mean(FunctionDefs), sd(Lines),sd(FunctionDefs))
```

```
## # A tibble: 2 x 5
##   Language   `mean(Lines)` `mean(FunctionDef~ `sd(Lines)` `sd(FunctionDefs~
##   <chr>              <dbl>              <dbl>       <dbl>             <dbl>
## 1 JavaScript          62.1              0.476        63.7             0.732
## 2 Python              50.7              2.44         29.6             1.83
```

```
#we need to normalize in order to more fairly compare
data = data %>%
  group_by(Language) %>%
  mutate(Lines.normed = scale(Lines)) %>%
  mutate(FunctionDefs.normed = scale(FunctionDefs))
```

```
plt = ggplot(data,aes(Lines.normed,FunctionDefs.normed))+
  facet_wrap(~Language)+
  geom_point(aes(colour=Language),size=1, show.legend=FALSE)+
```

```
  #geom_vline(xintercept=mean(data$Lines),colour="red",linetype="dashed")+
  #geom_hline(yintercept=mean(data$FunctionDefs),colour="red",linetype="dashed")+
  ggtitle("Scaled Lines v. Function Definitions")+
  xlab("Number of Lines (z score)")+
  ylab("Number of Function Definitions (z score)")+

  theme_bw()

plt
```



13.8 Sample Bias

You may have some questions about the sample we used. With any problem, we need to think about where our data came from (not just questions that we can ask *about* the data). Problems start in our sample, not in our analysis. This is an important concept to keep in mind when presented with *any* statistic. Is the sample representative? Where did the data come from? Is publicly available data different from what you could obtain from a private source? Is the sample from one group comparable with the sample from the other? In this case, I've tried to collect programs that are roughly the same content; introductory Computer Science programs and data structures. Whatever I find may not generalize to the languages as a whole; perhaps in writing out data structures the code is for educational purposes, and looks different from code in practice.

Whatever we conclude on this sample might not be representative on a different sample. Not to get too philosophical, but any statistics that we perform are really an exercise in:

"How do we know what we know?"

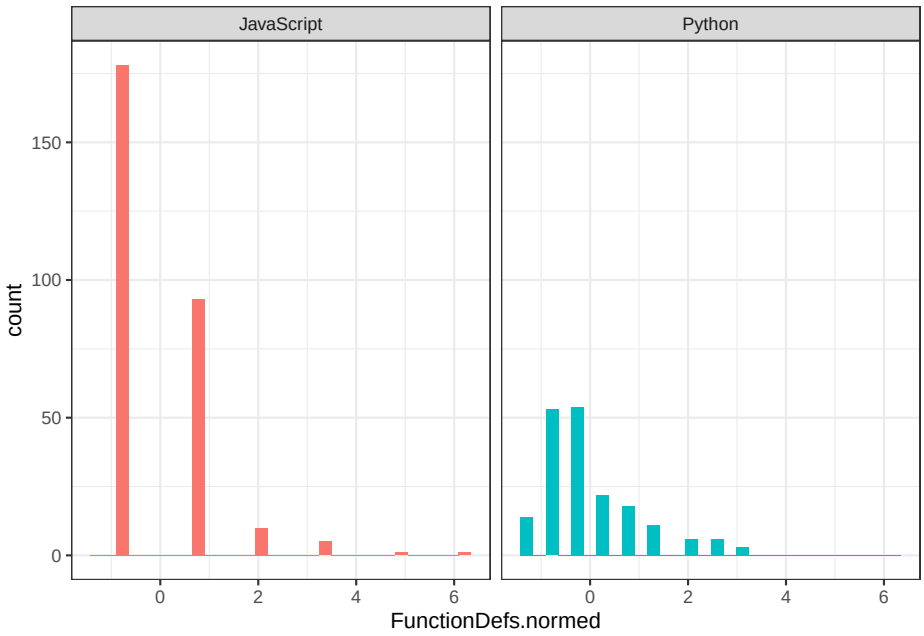
```
cor(data$FunctionDefs[data$Language=="JavaScript"],data$Lines[data$Language=="JavaScript"])
```

```
## [1] 0.0218202
```

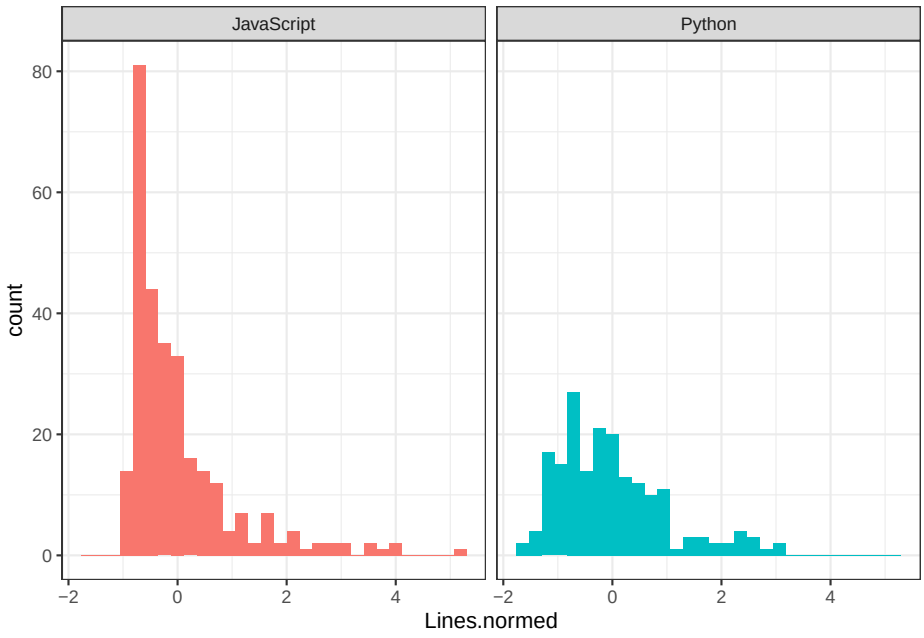
13.9 Normality Matters

We are all about finding differences, but comparing those differences is a delicate business. We learned that we can't compare Apples to Oranges, so we **normalized** our data. Sometimes, that's enough to be able to draw direct comparisons. Once we are on the same scale, we can go ahead and use statistical tests to compare groups. But there's something else that matters, and that's whether our data can be represented **parametrically** or not. Don't worry, it's not as scary as it sounds. **Parametric** means that the data distribution can be represented using *parameters*, like mean and standard deviation. We learned that the normal distribution can be expressed in terms of mean and standard deviation, and any data that falls **normally** is parametric. But lots of data will *not* fall under a normal distribution. Never fear, the weirdos can be compared too. We use something called **nonparametric testing**. Yep, you guessed it. These tests don't represent the data with parameters. They tend to use **ranks** instead. For now, just know that in order to compare two samples, we need to make sure we know which tests to use given the normality of those samples.

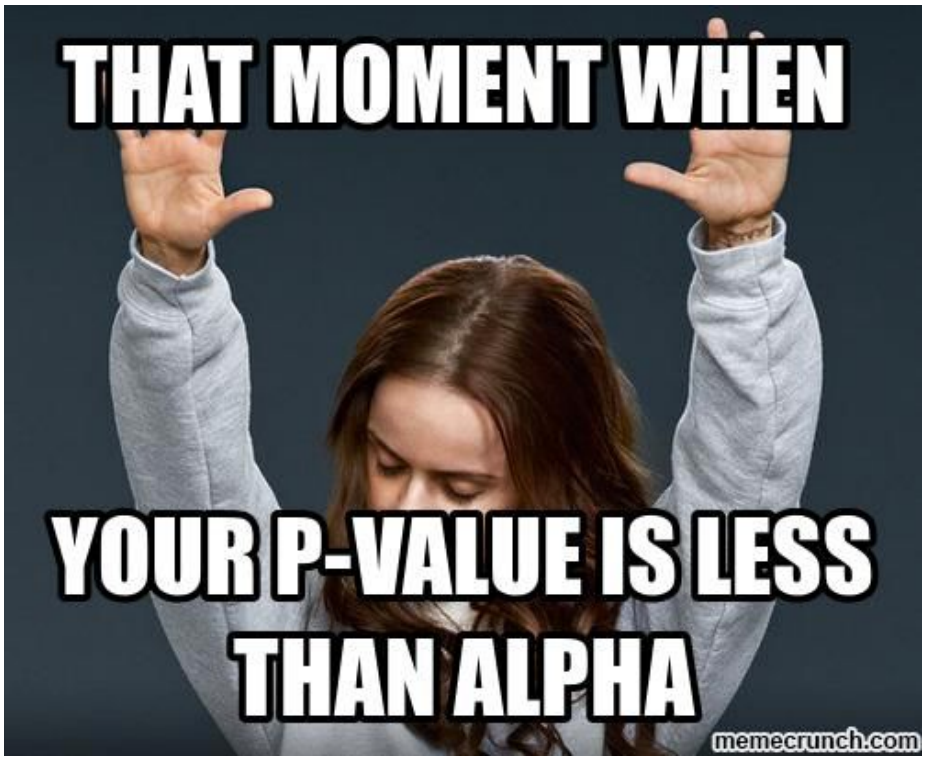
```
# here we can take a look at the distributions of our normalized (z-scores) data
plt = ggplot(data, aes(FunctionDefs.normed,fill=Language))+
  geom_histogram(show.legend=FALSE)+
  #geom_density()+
  facet_wrap(~Language)+
  theme_bw()
plt
```




```
plt = ggplot(data, aes(Lines.normed,fill=Language))+
  geom_histogram(show.legend=FALSE)+
  #geom_density()+
  facet_wrap(~Language)+
  theme_bw()
plt
```



it doesn't look normal, and that's pretty obvious. But for less obvious deviation from normality, we use a **Shapiro-Wilk Test**. The test is beginning with the hypothesis that our distribution and a normal distribution **are different**. The test will give back a W score and a **p-value**. When W is small, it is likely that our distribution is *different from normal*, with probability p. If the *p-value* is less than .05, we tend to conclude that our distribution is **not** normal.




```
# Python number of function definitions
shapiro.test(data$FunctionDefs[data$Language=="Python"])
```

```
##
##  Shapiro-Wilk normality test
##
## data:  data$FunctionDefs[data$Language == "Python"]
## W = 0.87633, p-value = 2.871e-11
```

```
# JavaScript number of function definitions
shapiro.test(data$FunctionDefs[data$Language=="JavaScript"])
```

```
##
##  Shapiro-Wilk normality test
##
## data:  data$FunctionDefs[data$Language == "JavaScript"]
## W = 0.64367, p-value < 2.2e-16
```

```
# Python number of lines
shapiro.test(data$Lines[data$Language=="Python"])
```

```
##
##  Shapiro-Wilk normality test
```

```
##
## data:  data$Lines[data$Language == "Python"]
## W = 0.92613, p-value = 3.962e-08
```

```
# JavaScript number of lines
shapiro.test(data$Lines[data$Language=="JavaScript"])
```

```
##
##  Shapiro-Wilk normality test
##
## data:  data$Lines[data$Language == "JavaScript"]
## W = 0.75725, p-value < 2.2e-16
```

13.10 Does Comparing Averages Make Sense?

13.10.1 NonParametric Tests

13.11 How to Compare Spreads

13.11.1 Clustering

13.12 What Have We Learned?

13.13 Next Steps and Other Questions

Appendix A

License

This is a human-readable summary of (and not a substitute for) the license. Please see this page for the full legal text.

This work is licensed under the Creative Commons Attribution 4.0 International license (CC-BY-4.0).

You are free to:

- **Share**—copy and redistribute the material in any medium or format
- **Remix**—remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution**—You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **No additional restrictions**—You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity,

privacy, or moral rights may limit how you use the material.

Appendix B

Code of Conduct

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

B.1 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- using welcoming and inclusive language,
- being respectful of differing viewpoints and experiences,
- gracefully accepting constructive criticism,
- focusing on what is best for the community, and
- showing empathy towards other community members.

Examples of unacceptable behavior by participants include:

- the use of sexualized language or imagery and unwelcome sexual attention or advances,
- trolling, insulting/derogatory comments, and personal or political attacks,
- public or private harassment,
- publishing others' private information, such as a physical or electronic address, without explicit permission, and
- other conduct which could reasonably be considered inappropriate in a professional setting

B.2 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

B.3 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

B.4 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by emailing the project team. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

B.5 Attribution

This Code of Conduct is adapted from the Contributor Covenant version 1.4.

Appendix C

Contributing

Contributions of all kinds are welcome, from errata and minor improvements to entirely new sections and chapters: please email us or submit an issue or pull request to our GitHub repository. Everyone whose work is incorporated will be acknowledged; please note that all contributors are required to abide by our Code of Conduct.

Please note that we use Simplified English rather than Traditional English (i.e., American rather than British spelling and grammar).

C.1 Contributors

- Yim Register
- Greg Wilson

Appendix D

Glossary

Sometimes the biggest barrier to learning is the jargon that people use. Here's a glossary of some terms from software engineering, statistics, and data science so that you can tackle interesting problems instead of being hung up on language. If something isn't here, it's not you: it's us. Please let us know what terms are missing or what definitions are confusing and we'll do our best to fix them.

Fucci, D. et al. 2016. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering*. 43:597–614.

Fucci, D. et al. 2018. Need for sleep: The impact of a night of sleep deprivation on novice developers' performance. *IEEE Transactions on Software Engineering*.

Gousios, G. 2013. The ghtorrent dataset and tool suite. *In* Proc. 10th working conference on mining software repositories (msr'13). IEEE Press, Piscataway, NJ, USA. 233–236.