Name	NYIT ID	Project Title	Source
Mehreet Singh Bajaj	1274698	(3Projects) Predict Fuel Flow Rate of Airplanes TV, Halftime Shows, and the Big Game Clustering Bustabit Gambling Behavior	https://www.crowdanalytix.com/contests/predict-fuel-flow-rate-of-airplanes-during-different-phases-of-a-flight https://www.kaggle.com/achyutbabu/tv-halftime-shows-and-the-big-game https://github.com/mplza/DataCamp-R/tree/master/Clustering%20Bustabit%20Gambling%20Behavior/datasets
Mohammed Safwan Aslam Kazi	1270795	Disney Movies and Box Office Success The Android App Market on Google Play	https://data.world/kgarrett/disney-character-success-00-16/disney-characters.csv https://www.kaggle.com/lava18/google-play-store-apps/googleplaystore.csv https://www.kaggle.com/lava18/google-play-store-apps/googleplaystore_user_reviews.csv
Maitry Amishkumar Jariwala	1275288	Level Difficulty in Candy Crush Saga Predict Taxi Fares with Random Forests	**https://github.com/ryanschaub/Level-Difficulty-in-Candy-Crush-Saga/blob/master/candy_crush.csv** **https://github.com/noahweber1/datacamp-project---PREDICT-TAXI-FARES-WITH-RANDOM-FORESTS**
Rishabh Badjatya	1270662	A Visual History of Nobel Prize Winners Analyze Your Runkeeper Fitness Data	https://www.kaggle.com/nobelfoundation/nobel-laureates https://github.com/AilingLiu/Analyze-Your-Runkeeper-Fitness-Data/tree/master/datasets

PREDICTING FUEL FLOW RATES OF AIRPLANES
And

TV, HALFTIME SHOWS AND THE BIG GAME

AND

CLUSTERING BUSTABIT GAMBLING BEHAVIOUR

PROJECTS

Submitted in partial fulfilment of the requirement for the award of the degree of

MASTER'S IN DATA SCIENCE

In course

DATA MINING (CSCI 415)

by

MEHREET SINGH BAJAJ

1274698

New York Institute of Technology

DECLARATION

I hereby declare that the projects entitled " **PREDICTING FUEL FLOW RATES OF AIRPLANES", "TV, HALFTIME SHOWS AND THE BIG GAME** " and "**CLUSTERING BUSTABIT GAMBLING BEHAVIOUR"** submitted as part of the partial course requirements for the course ***DATA MINING (project part – 1)*** , for the award of the degree of Masters in Data Science at New York Institute of Technology during the 2nd semester, has been carried out by me. I declare that the project has not formed the basis for the award of any degree, associate ship, fellowship or any other similar titles elsewhere.

Further, I declare that I will not share, re-submit or publish the code, idea, framework and/or any publication that may arise out of this work for academic or profit purposes without obtaining the prior written consent of the Course Faculty Mentor and Course Instructor.

Signature of the Student:

Place: New York Institute of Technology

Date: 10.10.2020

CERTIFICATE

This is to certify that the projects entitled " **PREDICTING FUEL FLOW RATES OF AIRPLANES" , " TV, HALFTIME SHOWS AND THE BIG GAME** " and "**CLUSTERING BUSTABIT GAMBLING BEHAVIOUR"** is a record of bonafide work carried out as part of the course ***DATA MINING (project part 1)***, under my guidance by ***Mehreet Singh Bajaj (1274698)*** , during the academic semester ***2nd***, in partial fulfilment of the requirements for the award of the Degree of Masters in Data Science , at New York Institute of Technology during academic year 2020.

HUANYING (HELEN) GU
Project guide
New York Institute of Technology

ACKNOWLEDGEMENT

I wish to express my deep sense of gratitude to my guide HUANYING (HELEN) GU, Professor , Computer and Data Science department , New York Institute of Technology for her guidance and useful suggestions, for encouraging me to work on this project, using the techniques taught in the class and implementing them for real case scenarios, without which it would've been impossible to complete this project , that too in time. I'm really thankful to her for believing in me and my project idea and helping me in shaping it up even better than I thought I could.

I feel blessed to have parents Dr Satinderjit Singh Bajaj and Dr Jagminder Kaur Bajaj . It was due to the intellectual discussion that they had with me all the time that always motivated me and inspired me to do something, and I could come up with the willingness to do this project.

I would also like to thank my cousin Preet Kanwal Singh who would keep me updated with the project requirements and help me out whenever I was stuck in a problem.

I would also like to mention my friend Hellen Vials and younger brother Prabh Arjun Singh Bajaj's little inputs about suggesting good modifications to the documents to make it look more appealing.

MEHREET SINGH BAJAJ (1274698)

Master's in data science

DATA MINING (2nd semester)

ABSTRACT

For part 1 of project assignment I have done 3 projects as follows:

PROJECT 1: PREDICTING FUEL FLOW RATES OF AIRPLANE

Fuel is of utmost importance as it is one of the non renewable resource and using it in a efficient and smart way is highly required in today's date. This is the most common problem faced by the airline industry these days .Fuel constitutes around 30% of the operating cost of airlines due to which we have higher cost of tickets . Developing cost saving strategies especially on fuel is of prime importance to airlines and reducing the emission of staggering amounts of greenhouse gases along with reducing fuel intake can have a significant positive impact on the environment .Given an idea of the utility of the resources used by my work so that no extra resources are used and may go in vain
Now the problem statement is that
The ability to predict the Fuel Flow (FF) rate of airplanes during different phases of a flight (Taxi, Takeoff, Climb, Cruise, Approach, and Rollout) will help understand

- The significant drivers of FF rate for each of these phases and also help understand the factors that make the airplanes perform at higher levels of fuel efficiency during the different phases of a flight.
- Insights from the exercise can help derive the best practices, which make flights more fuel efficient under different conditions.

After the project it was observed that climb, cruise and approach are most important phases for optimizing fuel and the consumption. The most important features consisted of rate of change of altitude, longitudinal acceleration and ground speed. The clearest visible trend between predictors and fuel flow rate is in the climb phase. In other phases, some of the predictors have a weakly visible trend, but since the root mean squared error is small, it is assumed that the features have strong nonlinear interactions which are not clearly visible in simple plots.
All of the work was done in Jupyter Notebook using Python and with the concepts of supervised Machine Learning with various data cleaning techniques from tidying up the data to applying different models like Random Forests and eXtreme Gradient Boosting to get the best suitable results and then validation and checking performance using Root Mean Square error (rmse) method.

PROJECT 2: TV, HALFTIME SHOWS AND THE BIG GAME

Whether or not you like football, the Super Bowl is a spectacle. There's drama in the form of blowouts, comebacks, and controversy in the games themselves. There are the ridiculously expensive ads, some hilarious, others gut-wrenching, thought-provoking, and weird. The half-time shows with the biggest musicians in the world, sometimes riding giant mechanical tigers or leaping from the roof of the stadium and in this project, we will find out how some of the elements of this show interact with each other. We will answer questions like:

- What are the most extreme game outcomes?
- How does the game affect television viewership?
- How have viewership, TV ratings, and ad cost evolved over time?
- Who are the most prolific musicians in terms of halftime show performances?

PROJECT 3: CLUSTERING BUSTABIT GAMBLING BEHAVIOUR

Have we ever wondered if you could quantify the behavior of gamblers at the casino? Some seem to win the most, some can be reckless and risky with their bets, and others are casual about the whole experience. While collecting this data from the casino might be a challenge, there is an online platform called Bustabit in which gamblers can bet Bitcoin. We've collected data on thousands of Bustabit gambling sessions and tracked the user, the amount bet, the amount won, and various properties of the particular game itself. Using this data, we will perform a cluster analysis from start to finish in an attempt to group gamblers based on their gambling behavior.

TABLE OF CONTENTS

Page No.

REQUIREMENT ANALYSIS

2.1 Dataset

PROJECT 1: PREDICTING FUEL FLOW RATES OF AIRPLANES

Our **dataset** consists of uncompressed data of about 14.4 GB of Flight record data (FDR) from 1005 different flight instances which represents readings of numerous sensors on aircraft and other engineering parameters. Every second measurements were done each of which contained 227 parameters. Flight data consisted of measurements during all the **8 phases** (unknown, preflight, taxi, takeoff, climb, cruise, descent and roll out) .

The dataset has been divided into training and testing set .The data distribution across training and test data sets is as follows:

- **Training dataset: 60%**
- **Test set: 40%**

There are 5 training data files each containing 200 csv files with record data for training and 1 testing file for validation of our model.

PROJECT 2: TV, HALFTIME SHOWS AND THE BIG GAME

Our **dataset** consists of uncompressed data made up of three CSV files, one with game data, one with TV data, and one with halftime musician data for all 52 Super Bowls through 2018.

Project 2

PROJECT 3 : CLUSTERING BUSTABIT GAMBLING BEHAVIOUR

The dataset used includes 10,000 games of Bustabit. Each game tracks the particular gambler, the Busted at value of the game, and the multiplier at which the gambler cashed out.

2.2 A Look at Data

PROJECT 1: PREDICTING FUEL FLOW RATES OF AIRPLANES

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	ACID	Flight_ins	Year	Month	Day	Hour	Minute	Second	ABRK	ELEV_1	ELEV_2	EVNT	FADF	FADS	FGC3	FIRE_1	FIRE_2	FIRE_3	FIRE_4	FLAP	FQTY_1	FQTY_2
2	676	6.76E+14	2004	5	11	15	18	18	119.9836	20.51736	60.29174	1	15	15	120	0	0	0	0	94	6536	
3	676	6.76E+14	2004	5	11	15	18	19	119.9836	20.51736	60.33266	1	15	15	120	0	0	0	0	95	6536	
4	676	6.76E+14	2004	5	11	15	18	20	119.9836	20.51736	60.29174	1	15	15	120	0	0	0	0	95	6536	
5	676	6.76E+14	2004	5	11	15	18	21	119.9836	20.51736	60.33266	1	15	15	120	0	0	0	0	94	6536	
6	676	6.76E+14	2004	5	11	15	18	22	119.9836	20.51736	60.33266	1	15	15	120	0	0	0	0	94	6536	
7	676	6.76E+14	2004	5	11	15	18	23	119.9836	20.53782	60.3122	1	15	15	120	0	0	0	0	94	6536	
8	676	6.76E+14	2004	5	11	15	18	24	119.9836	20.51736	60.3122	1	15	15	120	0	0	0	0	94	6536	
9	676	6.76E+14	2004	5	11	15	18	25	119.9836	20.51736	60.3122	1	15	15	120	0	0	0	0	94	6536	
10	676	6.76E+14	2004	5	11	15	18	26	119.9836	20.51736	60.3122	1	15	15	120	0	0	0	0	95	6536	
11	676	6.76E+14	2004	5	11	15	18	27	119.9836	20.51736	60.3122	1	15	15	120	0	0	0	0	94	6536	
12	676	6.76E+14	2004	5	11	15	18	28	119.9836	20.51736	60.3122	1	15	15	120	0	0	0	0	94	6536	
13	676	6.76E+14	2004	5	11	15	18	29	119.9836	20.4969	60.3122	1	15	15	120	0	0	0	0	94	6536	
14	676	6.76E+14	2004	5	11	15	18	30	119.9836	20.51736	60.33266	1	15	15	120	0	0	0	0	94	6536	
15	676	6.76E+14	2004	5	11	15	18	31	119.9836	20.51736	60.33266	1	15	15	120	0	0	0	0	94	6536	
16	676	6.76E+14	2004	5	11	15	18	32	119.9836	20.47644	60.3122	1	15	15	120	0	0	0	0	94	6536	
17	676	6.76E+14	2004	5	11	15	18	33	119.9836	20.53782	60.3122	1	15	15	120	0	0	0	0	94	6536	
18	676	6.76E+14	2004	5	11	15	18	34	119.9836	20.55827	60.33266	1	15	15	120	0	0	0	0	94	6536	
19	676	6.76E+14	2004	5	11	15	18	35	119.9836	20.41507	60.47586	1	15	15	120	0	0	0	0	94	6536	
20	676	6.76E+14	2004	5	11	15	18	36	119.9836	20.47644	60.43495	1	15	15	120	0	0	0	0	95	6528	
21	676	6.76E+14	2004	5	11	15	18	37	119.9836	20.4969	60.35312	1	15	15	120	0	0	0	0	95	6528	
22	676	6.76E+14	2004	5	11	15	18	38	119.9836	20.53782	60.37357	1	15	15	120	0	0	0	0	94	6528	
23	676	6.76E+14	2004	5	11	15	18	39	119.9836	20.53782	60.3122	1	15	15	120	0	0	0	0	94	6536	
24	676	6.76E+14	2004	5	11	15	18	40	119.9836	20.55827	60.3122	1	15	15	120	0	0	0	0	94	6536	
25	676	6.76E+14	2004	5	11	15	18	41	119.9836	20.61964	60.3122	1	15	15	120	0	0	0	0	95	6536	
26	676	6.76E+14	2004	5	11	15	18	42	119.9836	20.57873	60.33266	1	15	15	120	0	0	0	0	94	6536	
27	676	6.76E+14	2004	5	11	15	18	43	119.9836	20.61964	60.37357	1	15	15	120	0	0	0	0	94	6536	
28	676	6.76E+14	2004	5	11	15	18	44	119.9836	20.68102	60.29174	1	15	15	120	0	0	0	0	94	6536	
29	676	6.76E+14	2004	5	11	15	18	45	119.9836	20.94469	60.16899	1	15	15	120	0	0	0	0	94	6526	

676200405111519

PROJECT 2: TV, HALFTIME SHOWS AND THE BIG GAME

date	super_bowl	venue	city	state	attendance	team_winner	winning_pts	qb_winner_1	qb_winner_2	coach_winner	team_loser	losing_pts	qb_loser_1	qb_loser_2	coach_loser	combined_pts	difference_pts
2018-02-04	52	U.S. Bank Stadium	Minneapolis	Minnesota	67612	Philadelphia Eagles	41	Nick Foles		Doug Pederson	New England Patriots	33	Tom Brady		Bill Belichick	74	8
2017-02-05	51	NRG Stadium	Houston	Texas	70807	New England Patriots	34	Tom Brady		Bill Belichick	Atlanta Falcons	28	Matt Ryan		Dan Quinn	62	6
2016-02-07	50	Levi's Stadium	Santa Clara	California	71088	Denver Broncos	24	Peyton Manning		Gary Kubiak	Carolina Panthers	10	Cam Newton		Ron Rivera	34	14
2015-02-01	49	University of Phoenix Stadium	Glendale	Arizona	70288	New England Patriots	28	Tom Brady		Bill Belichick	Seattle Seahawks	24	Russell Wilson		Pete Carroll	52	4
2014-02-02	48	MetLife Stadium	East Rutherford	New Jersey	82529	Seattle Seahawks	43	Russell Wilson		Pete Carroll	Denver Broncos	8	Peyton Manning		John Fox	51	35
2013-02-03	47	Mercedes-Benz Superdome	New Orleans	Louisiana	71024	Baltimore Ravens	34	Joe Flacco		John Harbaugh	San Francisco 49ers	31	Colin Kaepernick		Jim Harbaugh	65	3
2012-02-05	46	Lucas Oil Stadium	Indianapolis	Indiana	68658	New York Giants	21	Eli Manning		Tom Coughlin	New England Patriots	17	Tom Brady		Bill Belichick	38	4
2011-02-06	45	Cowboys Stadium	Arlington	Texas	103219	Green Bay Packers	31	Aaron Rodgers		Mike McCarthy	Pittsburgh Steelers	25	Ben Roethlisberger		Mike Tomlin	56	6
2010-02-07	44	Sun Life Stadium	Miami Gardens	Florida	74059	New Orleans Saints	31	Drew Brees		Sean Payton	Indianapolis Colts	17	Peyton Manning		Jim Caldwell	48	14
2009-02-01	43	Raymond James Stadium	Tampa	Florida	70774	Pittsburgh Steelers	27	Ben Roethlisberger		Mike Tomlin	Arizona Cardinals	23	Kurt Warner		Ken Whisenhunt	50	4
2008-02-03	42	University of Phoenix Stadium	Glendale	Arizona	71101	New York Giants	17	Eli Manning		Tom Coughlin	New England Patriots	14	Tom Brady		Bill Belichick	31	3
2007-02-04	41	Dolphin Stadium	Miami Gardens	Florida	74512	Indianapolis Colts	29	Peyton Manning		Tony Dungy	Chicago Bears	17	Rex Grossman		Lovie Smith	46	12
2006-02-05	40	Ford Field	Detroit	Michigan	68206	Pittsburgh Steelers	21	Ben Roethlisberger		Bill Cowher	Seattle Seahawks	10	Matt Hasselbeck		Mike Holmgren	31	11
2005-02-06	39	Alltel Stadium	Jacksonville	Florida	78125	New England Patriots	24	Tom Brady		Bill Belichick	Philadelphia Eagles	21	Donovan McNabb		Andy Reid	45	3
2004-02-01	38	Reliant Stadium	Houston	Texas	71525	New England Patriots	32	Tom Brady		Bill Belichick	Carolina Panthers	29	Jake Delhomme		John Fox	61	3
2003-01-26	37	Qualcomm Stadium	San Diego	California	67603	Tampa Bay Buccaneers	48	Brad Johnson		Jon Gruden	Oakland Raiders	21	Rich Gannon		Bill Callahan	69	27
2002-02-03	36	Louisiana Superdome	New Orleans	Louisiana	72922	New England Patriots	20	Tom Brady		Bill Belichick	St. Louis Rams	17	Kurt Warner		Mike Martz	37	3
2001-01-28	35	Raymond James Stadium	Tampa	Florida	71921	Baltimore Ravens	34	Trent Dilfer		Brian Billick	New York Giants	7	Kerry Collins		Jim Fassel	41	27
2000-01-30	34	Georgia Dome	Atlanta	Georgia	72625	St. Louis Rams	23	Kurt Warner		Dick Vermeil	Tennessee Titans	16	Steve McNair		Jeff Fisher	39	7
1999-01-31	33	Pro Player Stadium	Miami Gardens	Florida	74803	Denver Broncos	34	John Elway		Mike Shanahan	Atlanta Falcons	19	Chris Chandler		Dan Reeves	53	15
1998-01-25	32	Qualcomm Stadium	San Diego	California	68912	Denver Broncos	31	John Elway		Mike Shanahan	Green Bay Packers	24	Brett Favre		Mike Holmgren	55	7
1997-01-26	31	Louisiana Superdome	New Orleans	Louisiana	72301	Green Bay Packers	35	Brett Favre		Mike Holmgren	New England Patriots	21	Drew Bledsoe		Bill Parcells	56	14
1996-01-28	30	Sun Devil Stadium	Tempe	Arizona	76347	Dallas Cowboys	27	Troy Aikman		Barry Switzer	Pittsburgh Steelers	17	Neil O'Donnell		Bill Cowher	44	10
1995-01-29	29	Joe Robbie Stadium	Miami Gardens	Florida	74107	San Francisco 49ers	49	Steve Young		George Seifert	San Diego Chargers	26	Stan Humphreys		Bobby Ross	75	23
1994-01-30	28	Georgia Dome	Atlanta	Georgia	72817	Dallas Cowboys	30	Troy Aikman		Jimmy Johnson	Buffalo Bills	13	Jim Kelly		Marv Levy	43	17
1993-01-31	27	Rose Bowl	Pasadena	California	98374	Dallas Cowboys	52	Troy Aikman		Jimmy Johnson	Buffalo Bills	17	Jim Kelly	Frank Reich	Marv Levy	69	35
1992-01-26	26	Metrodome	Minneapolis	Minnesota	63130	Washington Redskins	37	Mark Rypien		Joe Gibbs	Buffalo Bills	24	Jim Kelly		Marv Levy	61	13
1991-01-27	25	Tampa Stadium	Tampa	Florida	73813	New York Giants	20	Jeff Hostetler		Bill Parcells	Buffalo Bills	19	Jim Kelly		Marv Levy	39	1
1990-01-28	24	Louisiana Superdome	New Orleans	Louisiana	72919	San Francisco 49ers	55	Joe Montana		George Seifert	Denver Broncos	10	John Elway		Dan Reeves	65	45
1989-01-22	23	Joe Robbie Stadium	Miami Gardens	Florida	75129	San Francisco 49ers	20	Joe Montana		Bill Walsh	Cincinnati Bengals	16	Boomer Esiason		Sam Wyche	36	4
1988-01-31	22	Jack Murphy Stadium	San Diego	California	73302	Washington Redskins	42	Doug Williams		Joe Gibbs	Denver Broncos	10	John Elway		Dan Reeves	52	32
1987-01-25	21	Rose Bowl	Pasadena	California	101063	New York Giants	39	Phil Simms		Bill Parcells	Denver Broncos	20	John Elway		Dan Reeves	59	19
1986-01-26	20	Louisiana Superdome	New Orleans	Louisiana	73818	Chicago Bears	46	Jim McMahon		Mike Ditka	New England Patriots	10	Tony Eason	Steve Grogan	Raymond Berry	56	36
1985-01-20	19	Stanford Stadium	Palo Alto	California	84059	San Francisco 49ers	38	Joe Montano		Bill Walsh	Miami Dolphins	16	Dan Marino		Don Shula	54	22
1984-01-22	18	Tampa Stadium	Tampa	Florida	72920	Los Angeles Raiders	38	Jim Plunkett		Tom Flores	Washington Redskins	9	Joe Theismann		Joe Gibbs	47	29
1983-01-30	17	Rose Bowl	Pasadena	California	103667	Washington Redskins	27	Joe Theismann		Joe Gibbs	Miami Dolphins	17	David Woodley		Don Shula	44	10
1982-01-24	16	Pontiac Silverdome	Pontiac	Michigan	81270	San Francisco 49ers	26	Joe Montana		Bill Walsh	Cincinnati Bengals	21	Ken Anderson		Forrest Gregg	47	5
1981-01-25	15	Louisiana Superdome	New Orleans	Louisiana	76135	Oakland Raiders	27	Jim Plunkett		Tom Flores	Philadelphia Eagles	10	Ron Jaworski		Dick Vermeil	37	17
1980-01-20	14	Rose Bowl	Pasadena	California	103985	Pittsburgh Steelers	31	Terry Bradshaw		Chuck Noll	Los Angeles Rams	19	Vince Ferragamo		Ray Malavasi	50	12
1979-01-21	13	Orange Bowl	Miami	Florida	79484	Pittsburgh Steelers	35	Terry Bradshaw		Chuck Noll	Dallas Cowboys	31	Roger Staubach		Tom Landry	66	4
1978-01-15	12	Superdome	New Orleans	Louisiana	76400	Dallas Cowboys	27	Roger Staubach		Tom Landry	Denver Broncos	10	Craig Morton		Red Miller	37	17
1977-01-09	11	Rose Bowl	Pasadena	California	103438	Oakland Raiders	32	Kenny Stabler		John Madden	Minnesota Vikings	14	Fran Tarkenton		Bud Grant	46	18
1976-01-18	10	Orange Bowl	Miami	Florida	80187	Pittsburgh Steelers	21	Terry Bradshaw		Chuck Noll	Dallas Cowboys	17	Roger Staubach		Tom Landry	38	4
1975-01-12	9	Tulane Stadium	New Orleans	Louisiana	80997	Pittsburgh Steelers	16	Terry Bradshaw		Chuck Noll	Minnesota Vikings	6	Fran Tarkenton		Bud Grant	22	10
1974-01-13	8	Rice Stadium	Houston	Texas	71882	Miami Dolphins	24	Bob Griese		Don Shula	Minnesota Vikings	7	Fran Tarkenton		Bud Grant	31	17
1973-01-14	7	Memorial Coliseum	Los Angeles	California	90182	Miami Dolphins	14	Bob Griese		Don Shula	Washington Redskins	7	Bill Kilmer		George Allen	21	7
1972-01-16	6	Tulane Stadium	New Orleans	Louisiana	81023	Dallas Cowboys	24	Roger Staubach		Tom Landry	Miami Dolphins	3	Bob Griese		Don Shula	27	21
1971-01-17	5	Orange Bowl	Miami	Florida	79204	Baltimore Colts	16	Earl Morrall	Johnny Unitas	Don McCafferty	Dallas Cowboys	13	Craig Morton		Tom Landry	29	3
1970-01-11	4	Tulane Stadium	New Orleans	Louisiana	80562	Kansas City Chiefs	23	Len Dawson	Mike Livingston	Hank Stram	Minnesota Vikings	7	Joe Kapp		Bud Grant	30	16
1969-01-12	3	Orange Bowl	Miami	Florida	75389	New York Jets	16	Joe Namath		Weeb Ewbank	Baltimore Colts	7	Earl Morrall	Johnny Unitas	Don Shula	23	9
1968-01-14	2	Orange Bowl	Miami	Florida	75546	Green Bay Packers	33	Bart Starr		Vince Lombardi	Oakland Raiders	14	Daryle Lamonica		John Rauch	47	19
1967-01-15	1	Memorial Coliseum	Los Angeles	California	61946	Green Bay Packers	35	Bart Starr		Vince Lombardi	Kansas City Chiefs	10	Len Dawson		Hank Stram	45	25

PROJECT 3: CLUSTERING BUSTABIT GAMBLING BEHAVIOUR

bustabit

Id	GameID	Username	Bet	CashedOut	Bonus	Profit	BustedAt	PlayDate
14196549	3366002	papai	5	1.2	0	1	8.24	2016-11-20T19:44:19Z
10676217	3343882	znay22	3	NA	NA	NA	1.4	2016-11-14T14:21:50Z
15577107	3374646	rrrrrrrr	4	1.33	3	1.44	3.15	2016-11-23T06:39:15Z
25732127	3429241	sanya1206	10	NA	NA	NA	1.63	2016-12-08T18:13:55Z
17995432	3389174	ADM	50	1.5	1.4	25.7	2.29	2016-11-27T08:14:48Z
14147823	3365723	afrod	2	NA	NA	NA	1.04	2016-11-20T17:50:55Z
10802671	3344556	charles150	1	1.05	4	0.09	1.05	2016-11-14T18:52:58Z
6155532	3323151	antaresgold	8	NA	NA	NA	1.13	2016-11-08T16:57:04Z
15421788	3373736	fluxxy	2000	1.2	0	400	9.14	2016-11-23T00:26:05Z
3387542	3311658	jim4beaver	1000	1.05	0	50	10.09	2016-11-05T10:32:34Z
20408669	3401755	CIP1	6	NA	NA	NA	5.66	2016-11-30T22:47:56Z
11963476	3351578	dim4659	1	NA	NA	NA	2.69	2016-11-16T18:11:17Z
21409140	3406479	vadim777	24	NA	NA	NA	1.11	2016-12-02T07:13:23Z
2000777	3305205	venum1ze	584	NA	NA	NA	1.39	2016-11-03T14:18:07Z
21711361	3407867	runitrightnow	15	15	1.53	210.23	52.22	2016-12-02T16:48:51Z
11811238	3350751	Cheepot10	15	1.05	0	0.75	2	2016-11-16T12:28:36Z
5025134	3318575	paulri	1	1.33	0	0.33	1.9	2016-11-07T09:24:24Z
16265972	3378712	IsItPossible	2500	1.15	1.72	417.92	4.76	2016-11-24T10:15:52Z
2558604	3307967	Nord	12	1.5	1.75	6.21	1.67	2016-11-04T09:11:19Z
11423213	3348289	fatihakif	10	NA	NA	NA	2.67	2016-11-15T19:50:04Z
6753767	3325637	elena98hp1	2	1.25	0	0.5	4.11	2016-11-09T10:01:54Z
2387037	3307142	SHOLLEXHFF	2	NA	NA	NA	1.24	2016-11-04T03:18:03Z
25671554	3428895	Marat7777	660	NA	NA	NA	1.36	2016-12-08T15:47:19Z
25646985	3428754	putakabusta	500	NA	NA	NA	1.43	2016-12-08T14:49:57Z
16598171	3380523	MommiesStrats	21	NA	NA	NA	4.07	2016-11-24T22:38:07Z
2336159	3306862	benjovis	13	1.04	0	0.52	3.03	2016-11-04T01:20:30Z
294085	3296313	Amiri220	4	NA	NA	NA	1.82	2016-11-01T01:45:30Z
14011839	3364936	1225	100	1.11	0	11	46.11	2016-11-20T12:37:06Z
18691202	3393331	Ackaron	6	NA	NA	NA	1.11	2016-11-28T12:26:17Z
1152592	3300872	Amiri220	5	2	1.2	5.06	2.28	2016-11-02T08:37:46Z
4588286	3316569	aexxi2009	100	1.5	0	50	2.32	2016-11-06T19:59:40Z
25649054	3428765	user74	1500	1.1	0	150	2.3	2016-12-08T14:53:29Z
22107375	3410047	assefmonkey	177	1.37	0	65.49	6.7	2016-12-03T07:44:19Z
4723318	3317154	Noventa	38	1.05	0	1.9	5.11	2016-11-07T00:05:46Z
25800064	3429618	zmarat	10	1.04	0	0.4	1.22	2016-12-08T20:53:36Z
1091219	3300539	NEONDOUGH	99675	1.01	2.48	3463.91	1.33	2016-11-02T06:15:33Z
26420365	3433463	ksenaSS_04	90	NA	NA	NA	4.84	2016-12-09T22:53:44Z

PROJECT 1: PREDICTING FUEL FLOW RATES OF AIRPLANE

2.1 METHODOLOGY

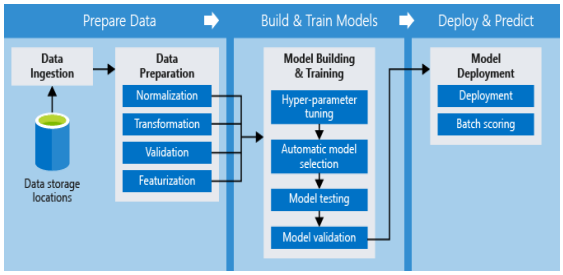


Fig 12 : The methodology diagram

This is the methodology which we will follow:

- **Data storage locations**: The recorded data or the raw data that is stored and will be used as the base for our whole project. More heavy the data is , the more trained our machine will be and more accurate the results will be.

- **Data ingestion**: Importing our set of data from the stored location to our main code which will further be processed. This includes steps like understanding our data, collecting it , describing the data, exploring and verifying the quality of it.

- **Data Preparation**: Consists of steps like selecting the data, cleaning it, constructing, integrating and formatting the data using technique like Normalization, Transformation, Validation and Featurization

- **Model Building & Training:** In this stage we will apply hyper parameter tuning (choose a set of optimal hyperparameters for a learning algorithm. Its value controls the learning process). Model selection (selecting various supervised algorithmic models which gives a better results to the given model), Model Testing (Testing the models and checking the errors with the rmse methods), scoring(applying those algorithmic models from historical data set to new dataset)
- **Model Deployment:** scoring (applying those algorithmic models from historical data set to new dataset in order to uncover the practical insights that help solving a problem) and finally deploying the model.

2.2 CONCEPTUAL OVERVIEW

Driving fuel efficiency involves developing strategies that touch upon various aspects of airplanes - broadly some of which are highlighted below:

- **Aspects related to Aircraft's actions on the ground** - e.g. include reducing taxiing times to reduce engine running times which translate into reduced fuel intake.
- **Aspects related to route planning** – e.g. taking shorter routes when inflight to destination taking into consideration any altitude restrictions that exist.
- **Aspects related to aircraft design** – e.g. improving aerodynamics, redesigning aircraft components to conserve fuel or reducing the weight on board like installation of lighter seats.

There are different phases in flight during a flight instance. All the readings from the sensors fit into the airplane are taken from these sensors every second and saved in flight record data.

There are mainly 7 phases of flight and one unknown phase which holds the data which was not recognised to be from any phase. Here is the description of the phases of the flight from 0 to 7 . Variable PH in our dataset refers to the phase of flight.

2.3 PH TABLE

PH	*PHASE NAME*	*DESCRIPTION*
0	**Unknown**	Consists of those readings in which phase cannot be determined
1	**Preflight**	Phase before the flight includes some checkups of airplane and ensure everything is working fine
2	**Taxi**	This phase refers to the estimated time an aircraft spends taxing between it's parking stand and runway or vice versa
3	**Takeoff**	Fuel required to takeoff the plane
4	**Climb**	phase pointing to increasing altitude of an airplane
5	**Cruise**	Phase when an aircraft levels after a climb to set altitude before it sets to descend
6	**Approach**	A final approach is the last leg in an aircraft is lined up with the runway and Descending for landing
7	**Rollout**	The phase of an aircraft's landing during which it travels along the runway while losing speed

2.4 HOW TO OVERCOME THE PROBLEM?

The Flight Record Data (FDR) can be used to

- Understand which features of dataset are correlated with Target (FF) we are trying to predict during different phases of a flight (Taxi, Take off, Climb, Cruise, Approach, and Rollout)
- Derive best practices that make flights fuel efficient under different conditions?
- We can later on do Exploratory Analysis on these top features

2.5 APPROACH

This is just a short overview of what strategies have been used to solve the problem. The detailed description will be in the code and explanation section.

2.5.1 Importing libraries

Firstly, we imported some of the basic important libraries used in python like NumPy, pandas, glob, os, sklearn, matplotlib and seaborn for enhanced visualization of our results using plots.

2.5.2 Loading into Pandas

Some pre-processing steps included downloading the data and loading it., pre-processed to make it smaller and faster to load. All the CSV (comma separated) files were loaded into panda's data frame and concatenated. After this the 64-bit datatypes (int64, float64) were converted to 32-bit datatypes (int32, float32). Finally, the combined Data Frame was saved as **pickle** files. The total size of train and test data pickle files on disk was about 6.5 GB.

2.5.3 Tidying the Data

Now , to check the the basic statistics of the data and separate the unwanted data from the important data DataFrame.describe() method was used and some variables showing no variance were observed and dumped using Variance Threshold method and all N/A if were present were also thrown out.

2.5.4 Visualization

We compared fuel flow across various flight phases (There are 7 phases) excluding Unknown Phase. As different phases have very different fuel flows.We also segregated why few flight instances are different from others in terms of Fuel Flow.
We have 600 flight instances. Data visualization was made clear using distplot and boxplot.
Then using FacetGrid function we obtained fuel flow plot for every phase on a separate axis.
Violin Plot and swarmplot showed the spread of fuel flow in the cruise phase was very wide .
Then compared and visualised Flight_instance with FF using visualization through boxplots. As the instances were very large in number it became difficult to visualise.

2.5.5 Splitting the data

Now the sklearn train_test_split , cross_validation K Fold strategy was used here to split data into pairs into groups .

2.5.6 Checking the Errors

Using the Root Mean Square Error method the more validation error that comes up the poorer the model is .

2.5.7 Models

Like data visualization techniques can be applied to present data, optimization techniques can be implemented to plan for actionable conclusions . We have used two models here for better results. Optimized Machine Learning Algorithms like ExtraTreesRegressor (using Decision tree regression strategies) and eXtreme Gradient Boosting have shown amazing results in many competitive coding contests and are known for their less error in rmse and fast computation. In the end one with less validation error was considered a better model .

2.5.8 Finding the Top Features

Now top features were found using both the models using the **plot_importance method** we created which was using. feature**_importances method**. Then the feature ranking was done, from where we come to know that PH(phase of flight) is one of the most important variable

2.5.9 Finding the correlation using Heatmaps

TO find the correlation between top features including the FF variable with each other we get the correlated features and remove highly correlated ones.

From this we will also get other top features which are important for minimizing the Fuel Flow rate.

Problem Statement

Reducing fuel consumption is extremely important for aviation industry as fuel constitutes ~ 30% of the operating cost of airlines. Reducing fuel intake can also have a significant positive impact on the environment. Hence, developing cost saving strategies especially on fuel is of prime importance to airlines. Driving fuel efficiency involves developing strategies that touch upon various aspects of airplanes - broadly some of which are highlighted below:

- Aspects related to Aircraft's actions on the ground - e.g. include reducing taxiing times to reduce engine running times which translate in to reduced fuel intake.
- Aspects related to route planning – e.g. taking shorter routes when inflight to destination taking in to consideration any altitude restrictions that exist.
- Aspects related to aircraft design – e.g. improving aerodynamics, redesigning aircraft components to conserve fuel or reducing the weight on board like installation of lighter seats.

In [1]:
```
import pandas as pd
import numpy as np
import glob
import os
import matplotlib.pyplot as plt
%matplotlib inline
from pylab import rcParams
rcParams['figure.figsize'] = 12, 10
import seaborn as sns
sns.set(style="white", color_codes=True)


from sklearn.feature_selection import VarianceThreshold

from sklearn.ensemble import ExtraTreesRegressor
from sklearn import metrics
```

Some preprocessing steps

In [3]:
```
#Method to load all train files; slighlty modified to record flight insta
def load_data(path):
    all_files = glob.glob(path + "/*.csv")
    list = []
    for i, file in enumerate(all_files[:200]):
        df = pd.read_csv(file, index_col = None, header = 0)
        df['flight_instance'] = i
        list.append(df)
    return pd.concat(list)
```

In [4]:
```
path = r"data"
train = load_data(path)
```

In [5]:
```
#Lets check basic statistics on dtata
pd.set_option("display.max_columns", 250)
train.describe()
```

Out[5]:

	ACID	Year	Month	Day	Hour	Minute	Se
count	1217028.0	1.217028e+06	1.217028e+06	1.217028e+06	1.217028e+06	1.217028e+06	1.217028
mean	676.0	2.003531e+03	6.326908e+00	1.458690e+01	1.236361e+01	2.955334e+01	2.950014
std	0.0	5.637601e-01	3.734886e+00	8.426349e+00	4.694716e+00	1.741971e+01	1.731905
min	676.0	2.002000e+03	1.000000e+00	1.000000e+00	0.000000e+00	0.000000e+00	0.000000
25%	676.0	2.003000e+03	3.000000e+00	7.000000e+00	8.000000e+00	1.400000e+01	1.400000
50%	676.0	2.004000e+03	6.000000e+00	1.300000e+01	1.200000e+01	3.000000e+01	3.000000
75%	676.0	2.004000e+03	1.000000e+01	2.200000e+01	1.600000e+01	4.500000e+01	4.500000
max	676.0	2.004000e+03	1.200000e+01	3.100000e+01	2.300000e+01	5.900000e+01	5.900000

Observations

- Few columns have same values throughout (no variance). Would be good idea to throw them away.
- None of the columns have NA's

In [6]:
```
#Re affirming that there are no NA's
train.isnull().sum().sum()
```

Out[6]: 0

In [7]:
```
#Lets throw away all the columns with less than 0 variance

def remove_low_varcols(df, threshold):
    var = VarianceThreshold(threshold=threshold)
    var.fit(df)
    all_cols = df.columns.values

    low_var_cols = all_cols[~var.get_support()]
    print('Columns with Varianceless than or equal to threshold are: ', lo

    final_cols = all_cols[var.get_support()]
    df_new = df.loc[:, final_cols]
    print("New shape ", df_new.shape)
    return df_new


train = remove_low_varcols(train, 0)
```

```
Columns with Varianceless than or equal to threshold are:  ['ACID' 'FIRE
_2' 'FIRE_3' 'FIRE_4' 'FQTY_3' 'POVT' 'SMOK' 'WAI_2'
 'APUF_Mean' 'APUF_Min' 'APUF_Max' 'TOCW_Min' 'CALT']
New shape  (1217028, 214)
```

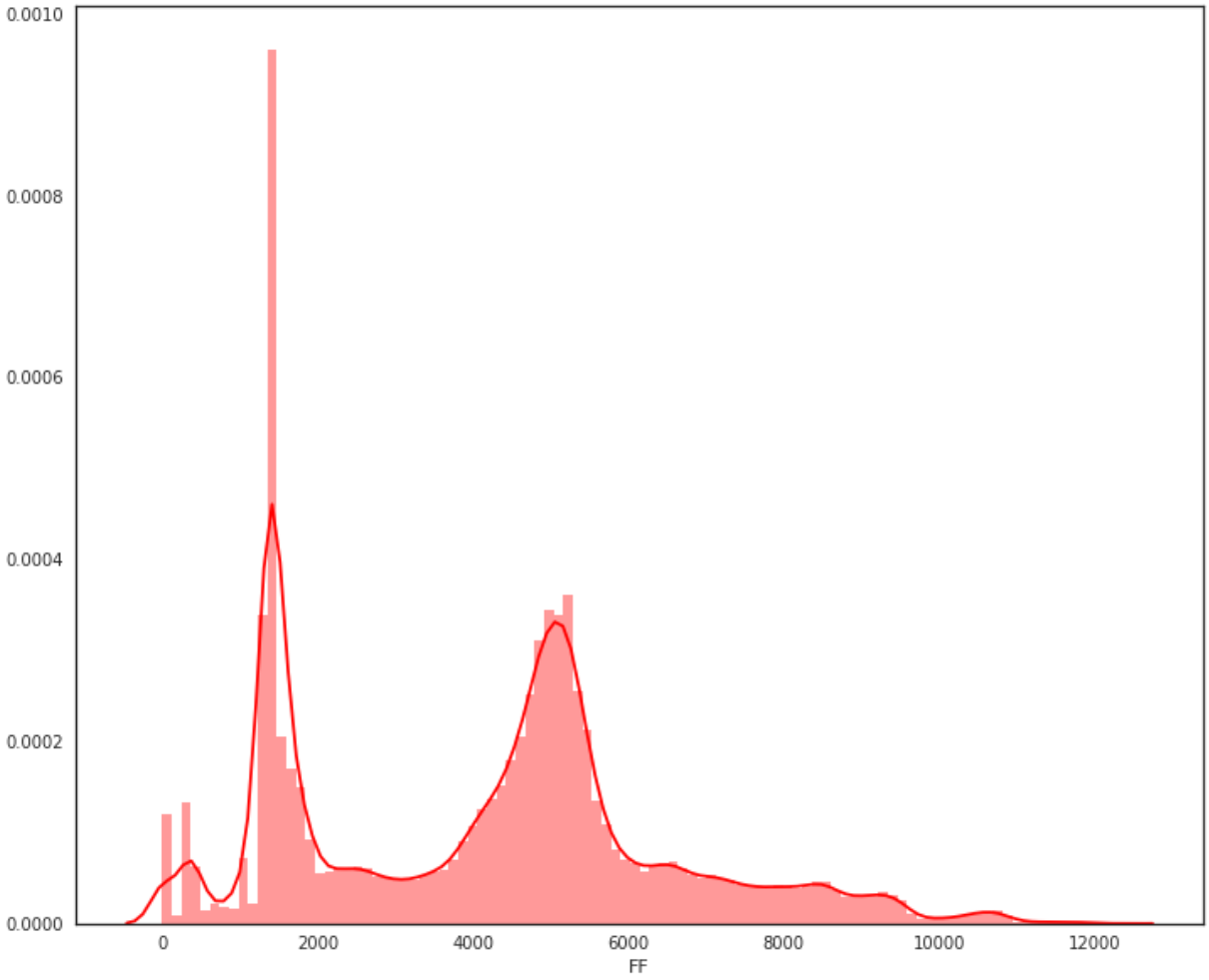
Visualizations

Few points to note here are:

- We are required to compare fuel flow across various flight phases (There are 7 phases). We are already aware that different phases have very different fuel flows.
- We also need to segregate why few flight instances are different from others in terms of Fuel flow
  - We have 600 flight instances

In [8]:
```
#Lets have a look at target distribution
plt.figure(figsize=(12,10))
sns.distplot(train['FF'], bins=100, color='red')
plt.show()
```

```
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

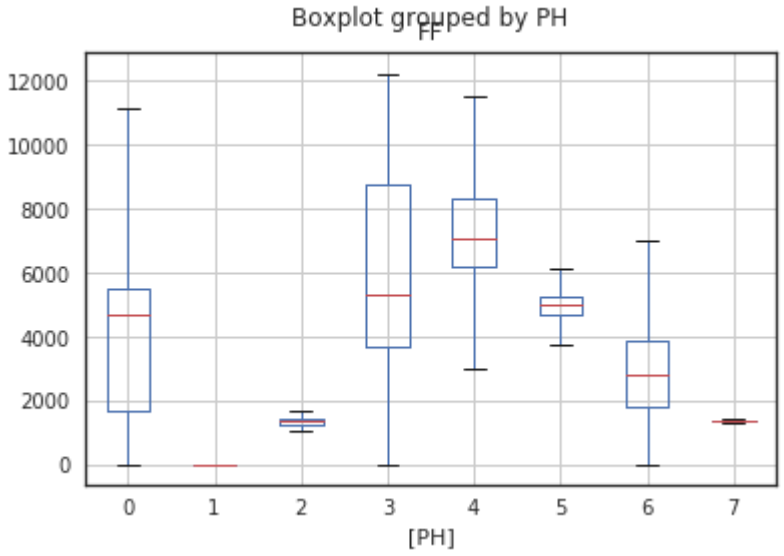


In [9]:
```
train.PH.value_counts()
```

Out[9]:
```
5    361086
2    242135
6    229122
4    216804
0    125960
3     27418
1     10447
7      4056
Name: PH, dtype: int64
```

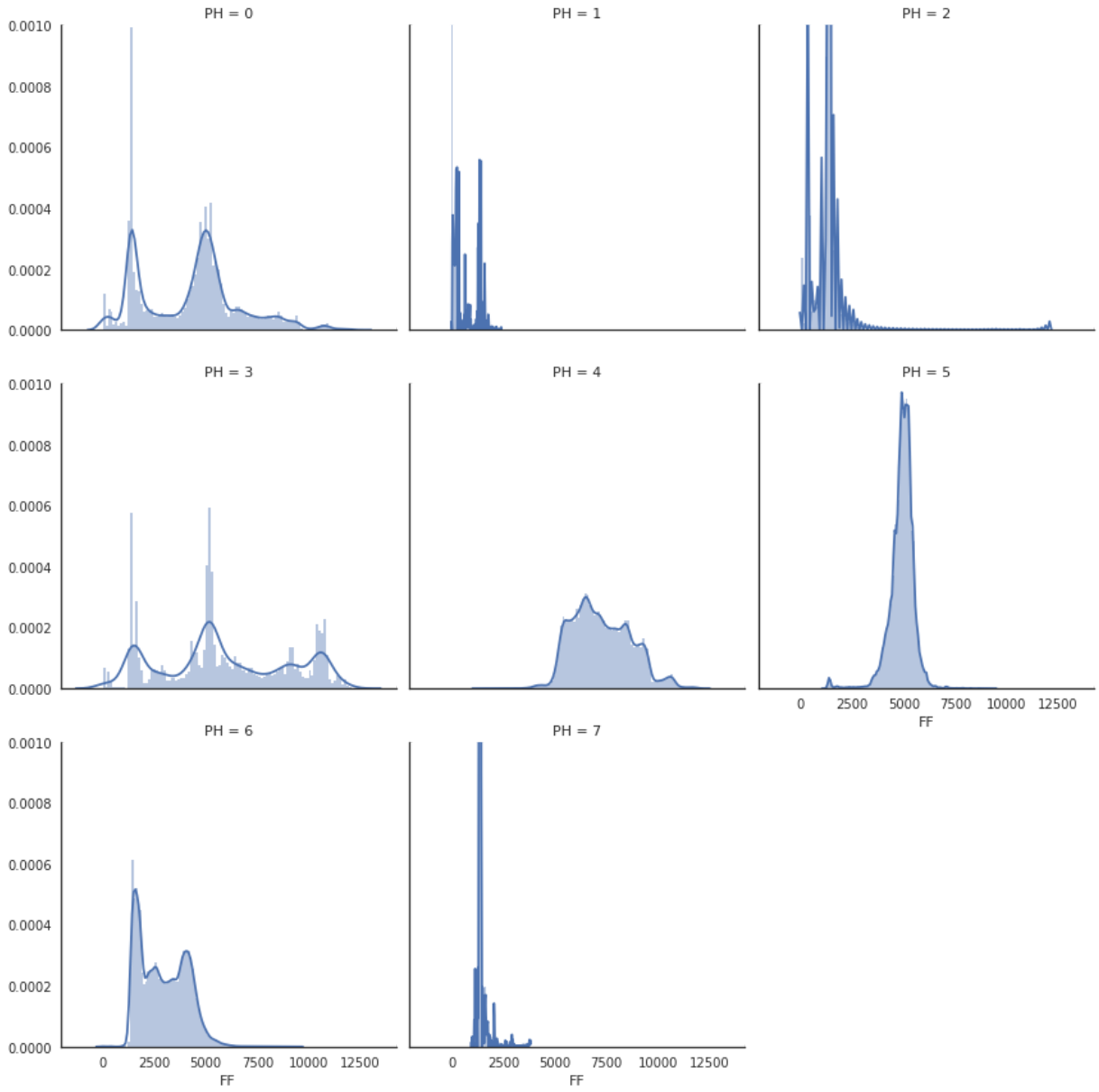
In [10]:
```
#Creating Box plot of Fuel Flow across different phases of flight
plt.figure()
train[["PH", "FF"]].boxplot( by="PH")
```

Out[10]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3fddd6dbe0>

<Figure size 432x288 with 0 Axes>
```



In [11]:
```
#Lets look at target vaiable for different flight phases
g = sns.FacetGrid(train, col="PH", col_wrap=3, size=4)
g = g.map(sns.distplot , 'FF', bins=100)
plt.ylim([0, 0.001])
plt.show()
```

```
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```



Looks like few phases 3, 4,5, 6 have most spread.

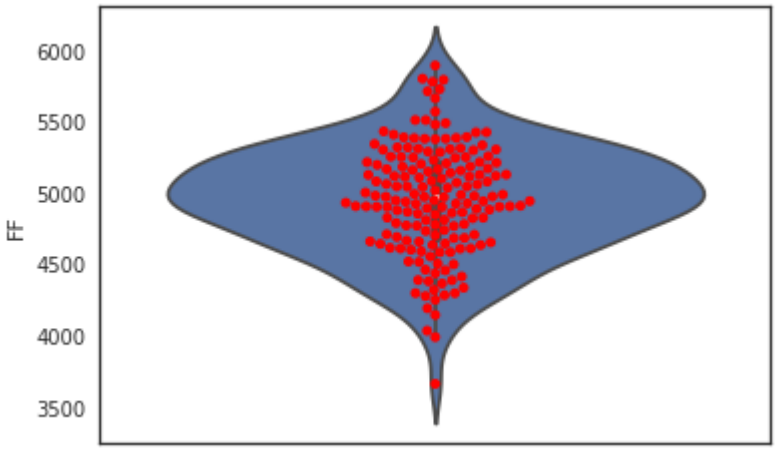
Phase 2 seems to have outliers

In [12]:
```
#Lets look at cruise phase (5) where plane spends most time.
train_ph5 = train.loc[train.PH == 5]

#
train_ph5_agg = train_ph5.groupby('flight_instance').agg('mean')
train_ph5_agg.head()
```

Out[12]:

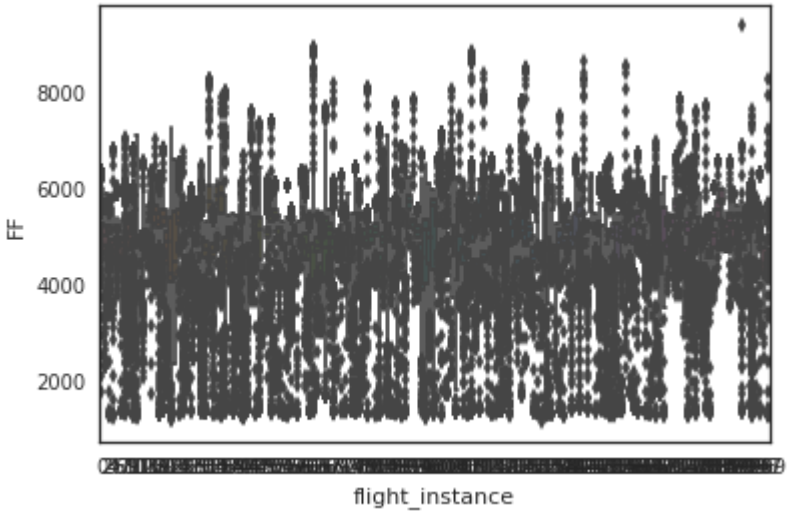
flight_instance	Year	Month	Day	Hour	Minute	Second	ABRK	ELEV_1	ELE
0	2004.0	1.0	18.0	10.158825	29.167289	29.451083	119.729707	-2.170143	79.573
2	2004.0	8.0	15.0	14.311631	37.683980	29.782736	119.939679	-7.867826	42.633
3	2004.0	2.0	28.0	11.000000	12.698593	29.582719	119.782676	-2.659749	71.942
4	2003.0	12.0	22.0	13.650624	26.062983	29.805704	119.983559	-2.700639	80.911
5	2004.0	4.0	28.0	17.549166	28.220802	29.485978	119.877092	-3.981836	60.074

In [13]:
```
sns.violinplot(y='FF', data=train_ph5_agg)
sns.swarmplot(y = 'FF', data= train_ph5_agg, color='red')
plt.show()
```



Quite a spread in mean fuel flow @cruise for different instances of flight

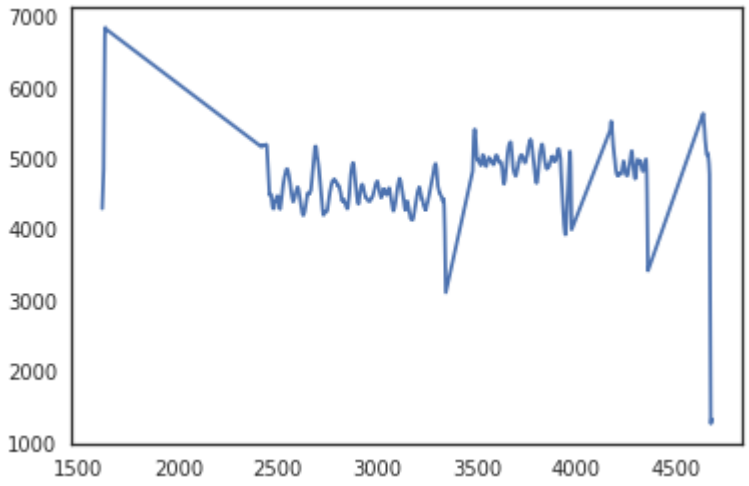
In [14]:
```
#How does distributions for different flight instances compare -
sns.boxplot(data=train_ph5, x='flight_instance', y='FF')
#sns.swarmplot(data=train_ph5, x='flight_instance', y='FF')
plt.show()
```



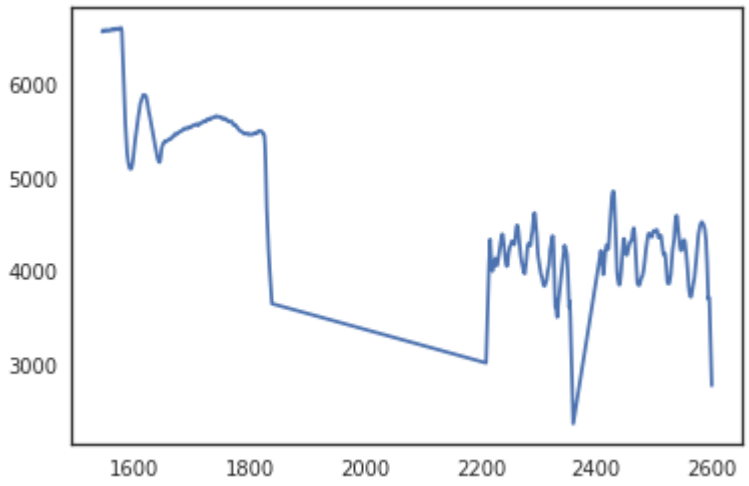
Looks interesting! Lot of values pretty far from median for all flight instances. Understanding them could be key here.

Lets pick few random flight instances and look if there is there are trends s w.r.t. duration of flight phase (We know that readings are sorted in time for a given flight instance)

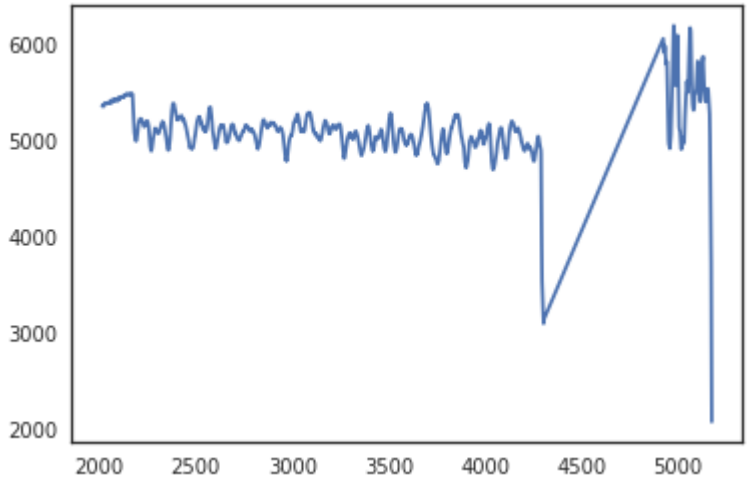
In [15]:
```
flight_instance = 4
plt.plot(train_ph5.loc[train_ph5.flight_instance == flight_instance, 'FF'
plt.show()
```



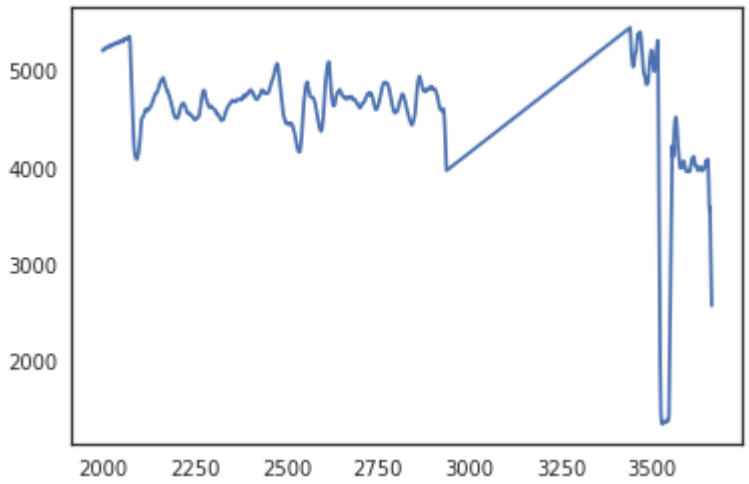
In [16]:
```
flight_instance = 23
plt.plot(train_ph5.loc[train_ph5.flight_instance == flight_instance, 'FF'
plt.show()
```



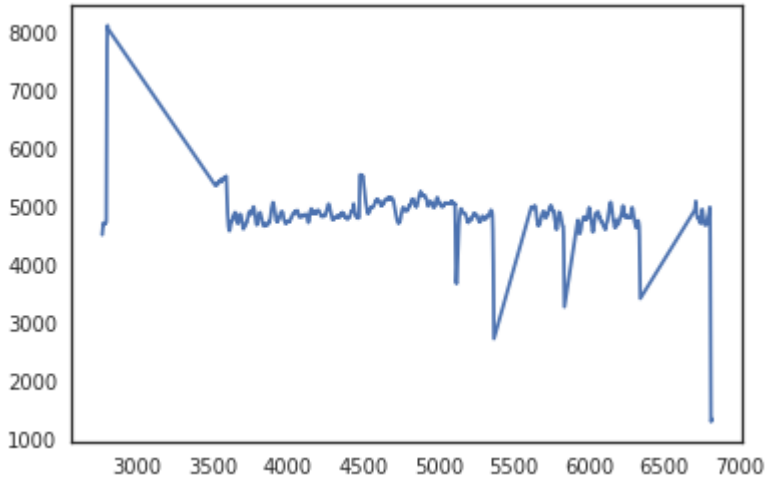
In [17]:
```
flight_instance = 49
plt.plot(train_ph5.loc[train_ph5.flight_instance == flight_instance, 'FF'
plt.show()
```



In [18]:
```
flight_instance = 61
plt.plot(train_ph5.loc[train_ph5.flight_instance == flight_instance, 'FF'
plt.show()
```



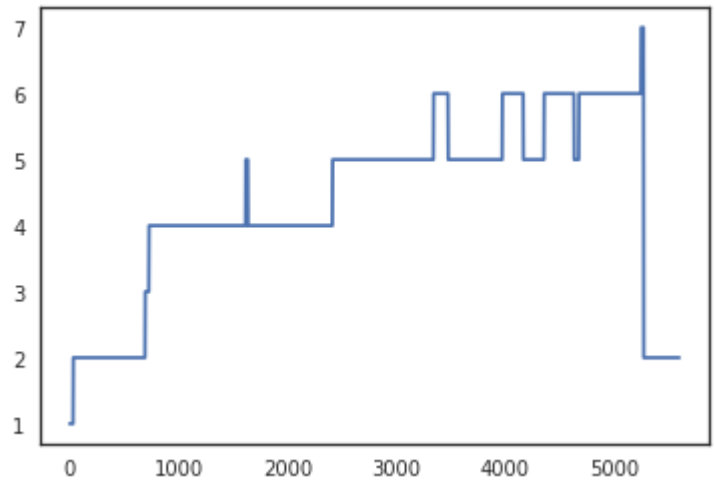
In [19]:
```
flight_instance = 79
plt.plot(train_ph5.loc[train_ph5.flight_instance == flight_instance, 'FF'
plt.show()
```



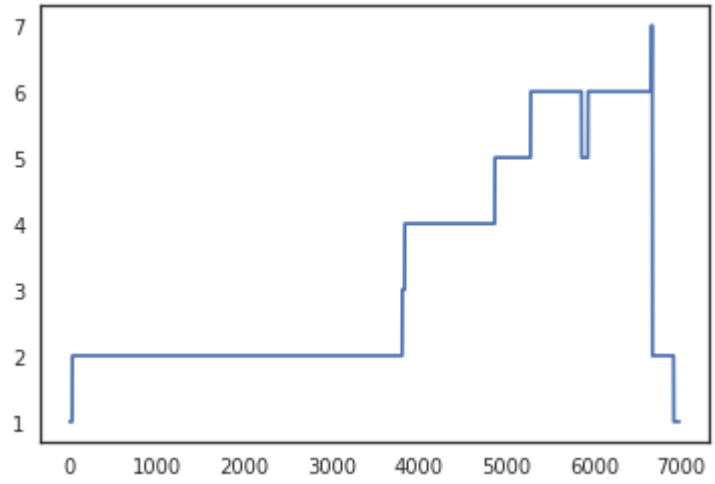
Observations:

- There patches of values where there are no readings in phase 5; probably phase changed intermittently (to be checked below)
- There are few values for each instance where sudden drop in fuel flow is observed (could be measurement/data processing error?)
- Last value and sometimes beginning values of a given phase are far off (could be due aggregation of values??)
- There is mean shift of fuel flow in some instances (could be change of cruise altitude etc.)

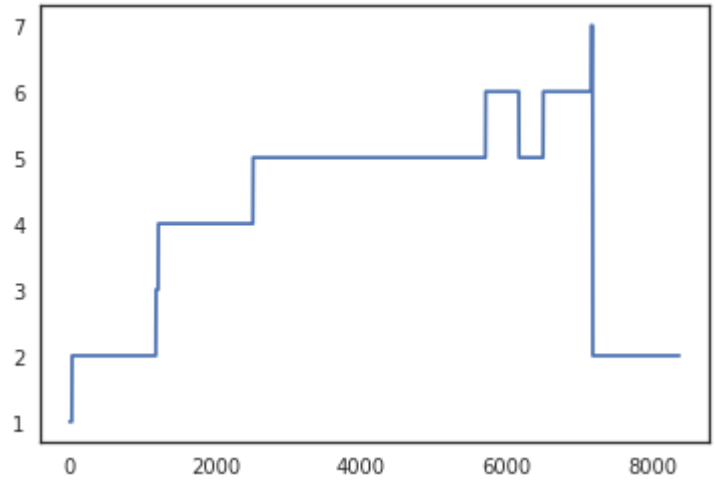
In [20]:
```
#To confirm faulty allocation of phases, lets plot instance 4.
plt.plot(train.loc[train.flight_instance == 4, 'PH'])
plt.show()
```



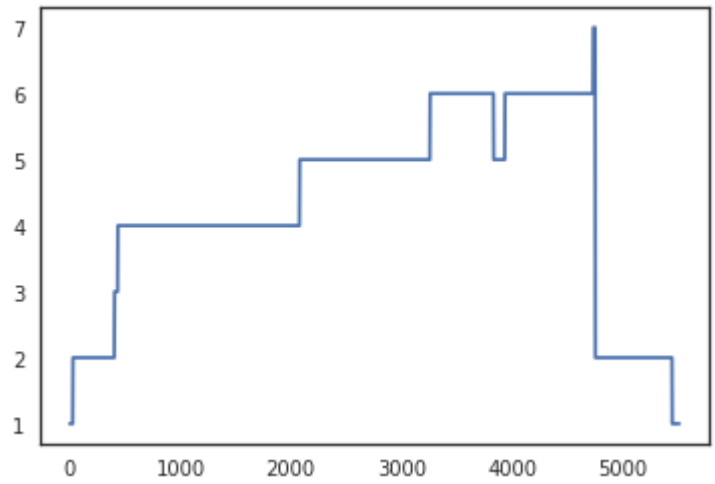
In [21]:
```
plt.plot(train.loc[train.flight_instance == 16, 'PH'])
plt.show()
```



In [22]:
```
plt.plot(train.loc[train.flight_instance == 56, 'PH'])
plt.show()
```



In [23]:
```
plt.plot(train.loc[train.flight_instance == 72, 'PH'])
plt.show()
```



SO, indeed change in phases are not monotonous. Not sure, whether this actual or assignment error. Most likely, assignment error; Plane would not oscillate so mny times between climp and cruise or cruise and approach.

In [24]:
```
#At this point lets split the dataset into train and validation sets base
from sklearn.model_selection import train_test_split, GroupKFold, cross_v

#5 fold cv strategy
folder = GroupKFold(n_splits=5)
cvlist = list(folder.split(train, y=None, groups=train.flight_instance))

#Use first split as Hold out cv - for quick checking
tr = train.iloc[cvlist[0][0]]
val = train.iloc[cvlist[0][1]]
```

In [25]:
```
#Check to ensure we are mixing flight instances between train and validat
set(tr.flight_instance.unique()) & set(val.flight_instance.unique())
```

Out[25]:
```
set()
```

In [26]:
```
def rmse(y_true, y_pred):
    return np.sqrt(metrics.mean_squared_error(y_true, y_pred))
```

In [27]:
```
#Lets dump everything in ETR and check which features come out on top
etr = ExtraTreesRegressor(max_depth=7, n_estimators= 200, n_jobs=-1, verbo

feats = [f for f in train.columns if f not in ['FF', 'flight_instance']]
etr.fit(tr[feats], tr['FF'])
```

```
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed:  7.6min
[Parallel(n_jobs=-1)]: Done 200 out of 200 | elapsed:  8.2min finished
```

Out[27]:
```
ExtraTreesRegressor(bootstrap=False, criterion='mse', max_depth=7,
          max_features='auto', max_leaf_nodes=None,
          min_impurity_decrease=0.0, min_impurity_split=None,
          min_samples_leaf=1, min_samples_split=2,
          min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=-1,
          oob_score=False, random_state=None, verbose=1, warm_start=Fals
e)
```

In [28]:
```
#Lets see rmse on hold out validation set
print("RMSE on train set :", rmse(tr['FF'], etr.predict(tr[feats])))
print("RMSE on hold out validation set:", rmse(val['FF'], etr.predict(val
```

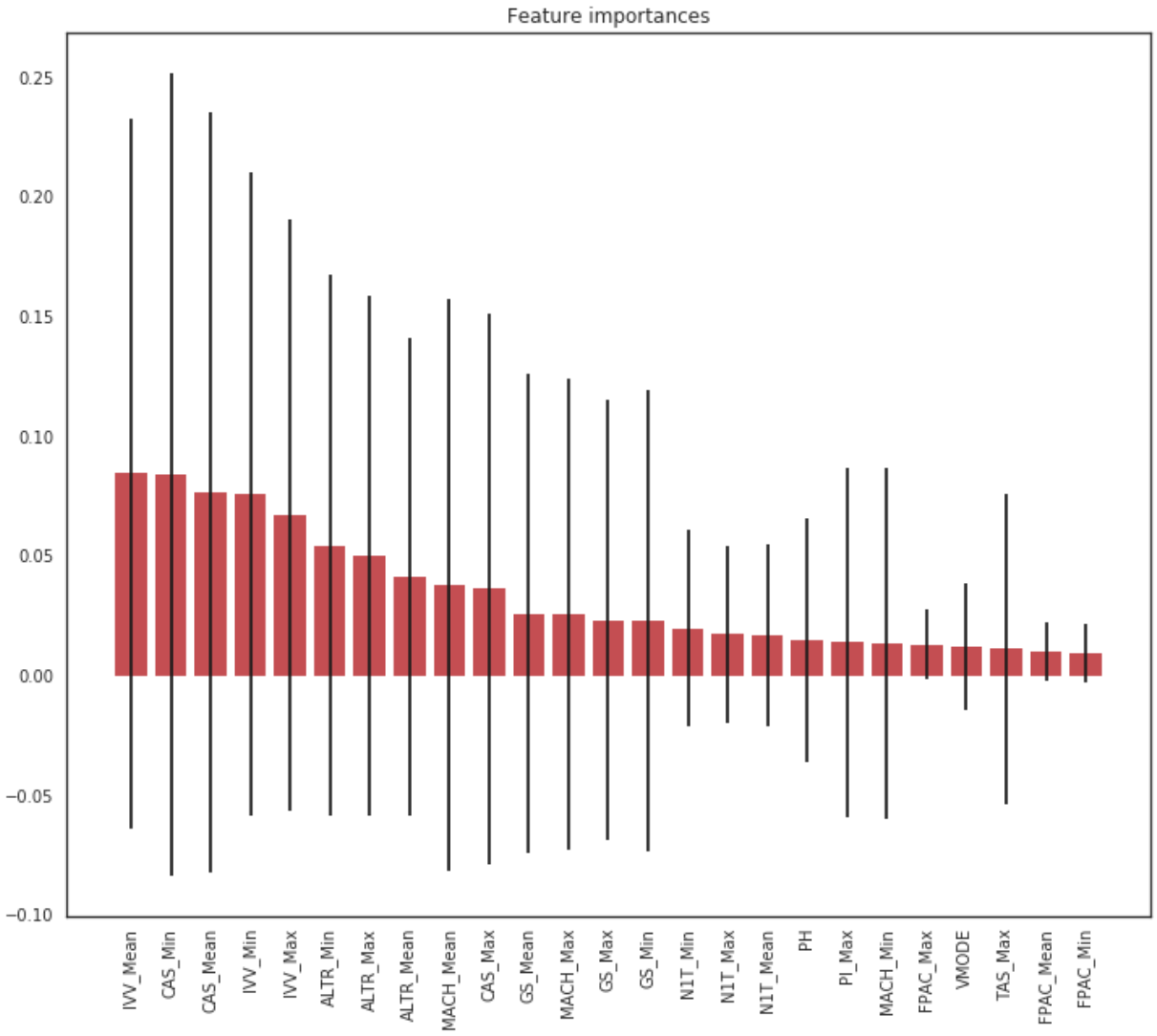
```
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.7s
[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:    4.5s
[Parallel(n_jobs=8)]: Done 200 out of 200 | elapsed:    4.9s finished

RMSE on train set : 329.0492207519591

[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.1s

RMSE on hold out validation set: 341.46706972668557

[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:    0.4s
[Parallel(n_jobs=8)]: Done 200 out of 200 | elapsed:    0.4s finished
```

In [29]:

```
def plot_importance(model, feats, n_feats):
    importances = model.feature_importances_
    std = np.std([tree.feature_importances_ for tree in model.estimators_
    indices = np.argsort(model.feature_importances_)[::-1][:n_feats]

    feats = np.array(feats)
    top_feats = feats[indices]
    #Print feature ranking
    print("Feature Ranking: ")

    for i, feat in enumerate(top_feats):
        print("{:d} {:s} ({:f}) ({:f})".format(i+1, feat, importances[ind

    plt.figure(figsize=(12,10))
    plt.title("Feature importances")
    plt.bar(range(len(top_feats)), importances[indices],
           color="r", yerr=std[indices], align="center")
    plt.xticks(range(len(top_feats)), top_feats, rotation=90)
    plt.show()
    return _, top_feats
```

In [30]:
```
_, top_feats = plot_importance(etr, feats, 25)
```

```
Feature Ranking:
1 IVV_Mean (0.084489) (0.148454)
2 CAS_Min (0.084107) (0.167770)
3 CAS_Mean (0.076386) (0.158847)
4 IVV_Min (0.075828) (0.134424)
5 IVV_Max (0.066964) (0.123356)
6 ALTR_Min (0.054371) (0.112998)
7 ALTR_Max (0.050322) (0.108571)
8 ALTR_Mean (0.041237) (0.099513)
9 MACH_Mean (0.037555) (0.119481)
10 CAS_Max (0.036193) (0.115377)
11 GS_Mean (0.025937) (0.100067)
12 MACH_Max (0.025539) (0.098464)
13 GS_Max (0.023136) (0.092056)
14 GS_Min (0.022971) (0.096472)
15 N1T_Min (0.019834) (0.041106)
16 N1T_Max (0.017343) (0.036877)
17 N1T_Mean (0.016924) (0.038210)
18 PH (0.014879) (0.050823)
19 PI_Max (0.013783) (0.072954)
20 MACH_Min (0.013265) (0.073216)
21 FPAC_Max (0.012998) (0.014419)
22 VMODE (0.012004) (0.026403)
23 TAS_Max (0.011043) (0.064733)
24 FPAC_Mean (0.010024) (0.012104)
25 FPAC_Min (0.009425) (0.012013)
```



Feature Ranking:

- PH (0.107589) (0.167965)
- LONG_Max (0.069545) (0.145660)
- IVV_Mean (0.067866) (0.147514)
- VIB_1_Mean (0.062372) (0.158833)
- VIB_1_Max (0.052972) (0.149057)
- CAS_Min (0.051805) (0.132064)
- LONG_Mean (0.047327) (0.146224)
- ALTR_Min (0.038022) (0.105542)
- IVV_Max (0.037572) (0.106540)
- VIB_1_Min (0.036064) (0.123443)
- ALTR_Mean (0.034586) (0.098909)
- CAS_Max (0.032939) (0.098710)
- CAS_Mean (0.029872) (0.098977)
- IVV_Min (0.025714) (0.068457)
- ALTR_Max (0.025460) (0.084879)
- LONG_Min (0.018638) (0.076449)
- MACH_Mean (0.017350) (0.074949)
- MACH_Min (0.016838) (0.076311)

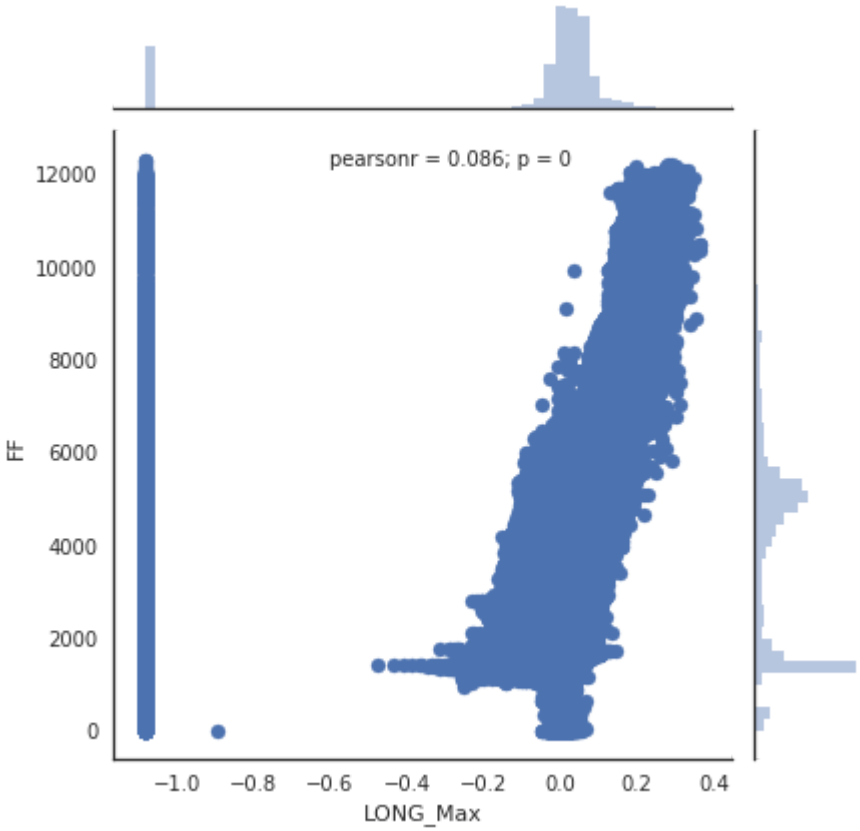
- TAS_Mean (0.014920) (0.048868)
- TAS_Max (0.014237) (0.056166)
- GS_Mean (0.013355) (0.066076)
- PI_Mean (0.012561) (0.062683)
- MACH_Max (0.009783) (0.057446)
- GS_Max (0.009533) (0.058464)
- FPAC_Mean (0.009038) (0.012956)

As expected we get Phase as one of the important variables.

Lets take a look at other variables

In [32]:
```
#Scatter plot between LONG_Max and FF (Fuel Flow)
plt.figure()
sns.jointplot("LONG_Max" , "FF", data=train)
plt.show()
```

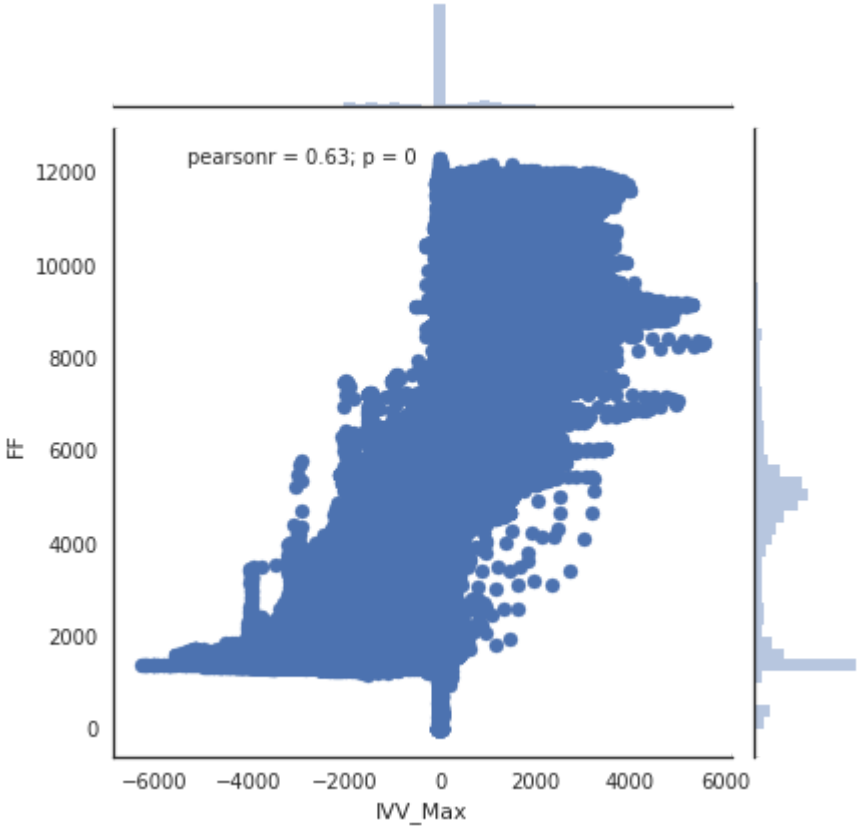
```
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "

<Figure size 432x288 with 0 Axes>
```



In [33]:
```
plt.figure()
sns.jointplot("IVV_Max" , "FF", data=train)
plt.show()
```

```
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
/home/user/anaconda3/lib/python3.6/site-packages/matplotlib/axes/_axes.p
y:6462: UserWarning: The 'normed' kwarg is deprecated, and has been repl
aced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

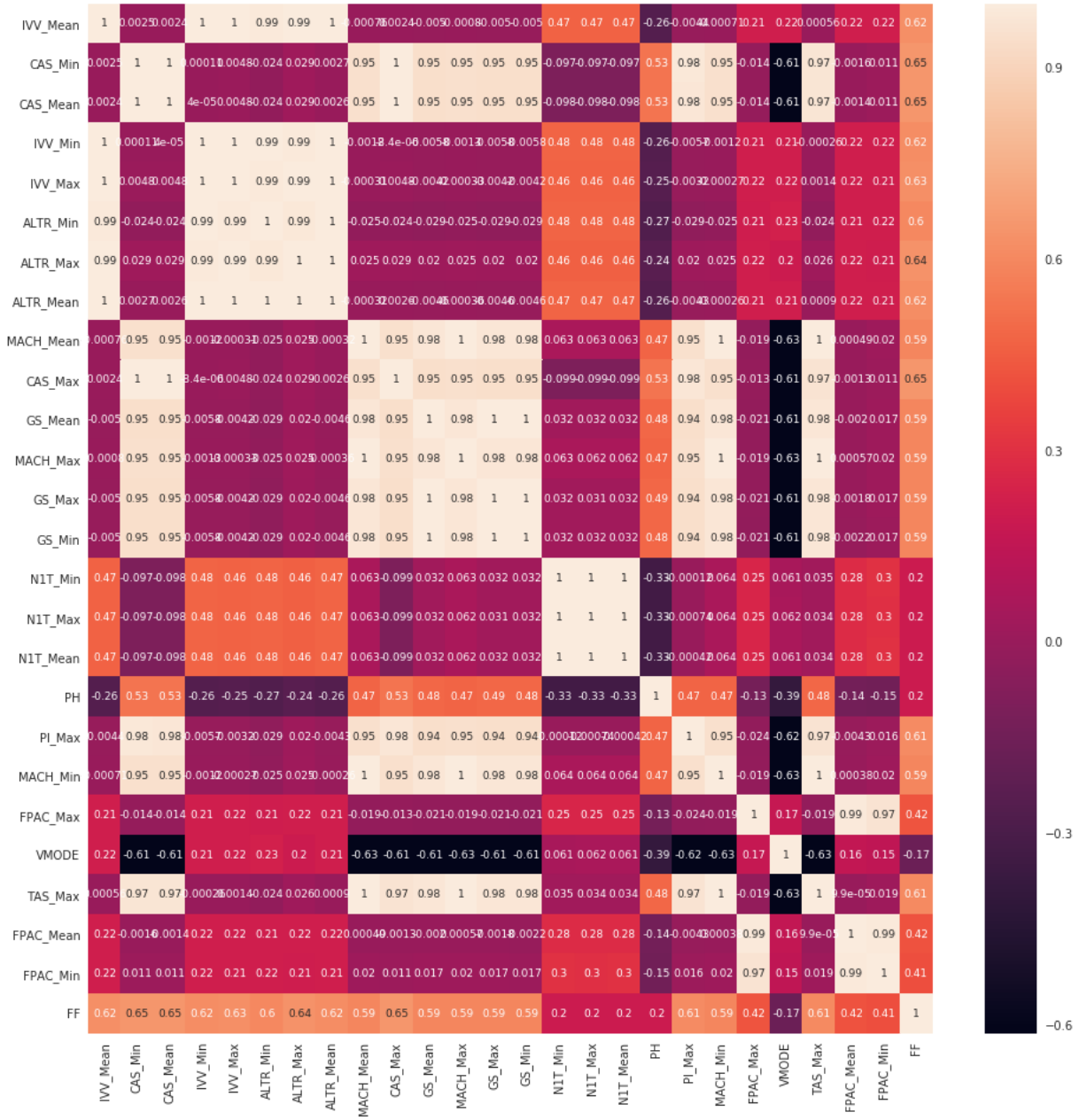
```
<Figure size 432x288 with 0 Axes>
```



Vibrations are related to acceleration, engine health and Phase. This might be agood one to dig deeper

In [34]:
```
corr_feats = list(top_feats) + ['FF']
corr_df = train[corr_feats].corr()
fig, ax = plt.subplots(figsize=(16,16))
sns.heatmap(corr_df, robust =True, annot=True, ax=ax, annot_kws={'size':9}
```

Out[34]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3fdc715c18>



Few other features that show up high are CAS(Corrected air speed), Mach, Ground speed, True air speed and Altitude related features. CAS and Mach are corelated. Also, min, max and mean are highly corelated for many features. It might be a good to remove some of the highly corelated ones.

DUMP everything into XGboost to get us a baseline

In [ ]:
```
#Lets dump everything into xgboost and see what we get.
#Warning: Not recommended to use this as final model. Remember - Garbage
import xgboost
from xgboost.sklearn import XGBRegressor

X_tr = tr[feats]
y_tr = tr['FF']

X_val = val[feats]
y_val = val['FF']

xgb_dump = XGBRegressor(max_depth=6, n_estimators=1000, colsample_bytree=
xgb_dump.fit(X_tr, y_tr, eval_set=[(X_tr, y_tr), (X_val, y_val)], eval_me
```

In [ ]:
```
#Feature importances from xgboost
from xgboost import plot_importance
fig, ax = plt.subplots(figsize=(12,30))
plot_importance(xgb_dump, ax=ax)
```

Validation RMSE - 200 (After dumping everything to xgboost)

We are overfitting by a lot here. Need to very careful about overfitting.

Some directions:

- PCA would be a good idea given so many corelated features and few with very little varince.
- Features related to groupings by phases would be my first choice
- Remove corelated features
- Features charaterizing flight instance

In [ ]:
```
```

2.6 DIFFERENCE IN VARIOUS PHASES OF FLIGHT

There are seven different phases of the flight in this dataset, marked by the values of 'PH' 0 to '7' a 'PH' value of 0 indicates that the phase is unknown.

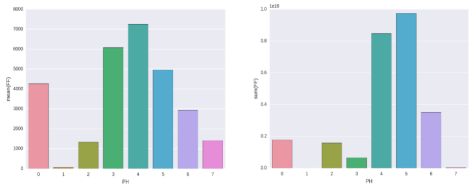


Fig 1 The mean fuel flow rate (left) and total fuel consumption (right) in each phase of flight. From these graphs, phase 4 and phase 5 appear to be the most important phases that drive the fuel consumption in an aircraft.

The plots in fig 1 show mean and total fuel consumption in different phases of flight.
The mean and standard deviation of the subset of data with unknown phase (mean = 4266, std = 2440) is similar to those of the entire dataset (mean = 4219, std = 2356).

This observation gives a strong indication that the **subset of data with PH = 0 is representative of the entire dataset and thus has a distribution of flight-phases similar to the entire dataset**.

The preflight phase is small and the fuel consumption is zero for most of the time (80% of the instances in this phase have FF = 0). The total fuel consumption is **highest** in the **climb (PH=4) and cruise (PH=5) phases**, which is obvious because these two phases are among the l**ongest phases of flights**.

The **approach (PH=6)** phase is also of a significant duration, but since the aircraft is **not accelerating anymore**, the fuel consumption is **smaller** in this phase.

The **takeoff (PH=3**) has a high rate of fuel consumption, but since this phase **does not last too long,** the total fuel consumption is rather small in this phase.

The difference between total fuel consumed in different flight instances is driven mainly by the **distance between source and destination**. Since the data does not contain the distance
information, we should derive it from the **ground speed**. The GS_Mean column has ground speed as a function of time, and thus we can obtain distance by simply integrating this column. The following plot shows the total fuel consumption as a function of total distance for each of the flight instances given in the data.

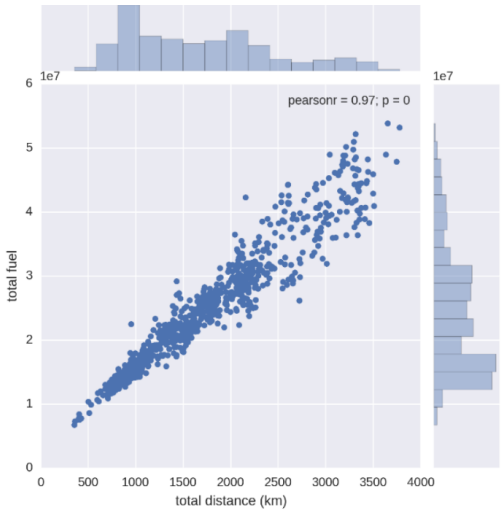


Fig 2 Joint distribution of fuel consumption and distance covered by the aircraft. A very strong linear correlation between distance and fuel consumed (pearson coefficient = 0.97) is found.

Since the taxi phase is mostly determined by the traffic and distance between aircraft bay and

runway, it is not really in the direct control of the flight operators. The rollout phase is very small and hence not particularly interesting. The other phases, namely takeoff, climb, cruise and approach are the ones which correspond to travel between source and destination and are
deemed to be more important.

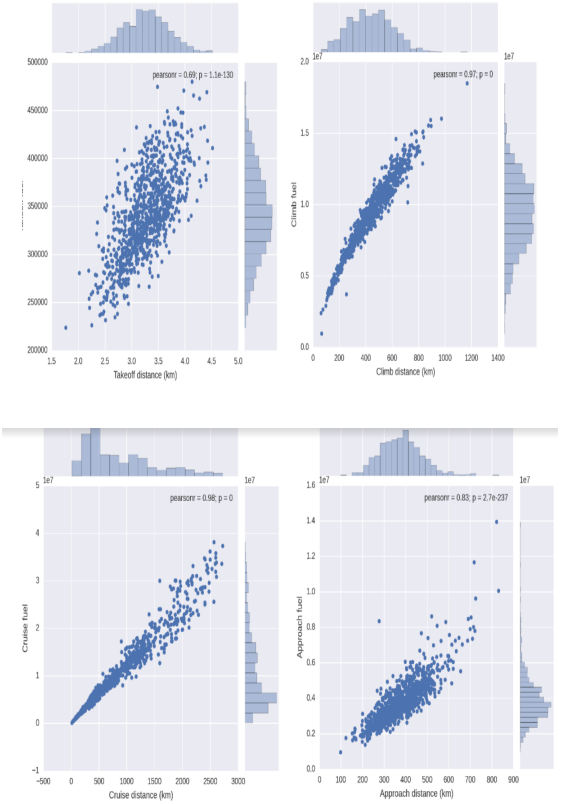
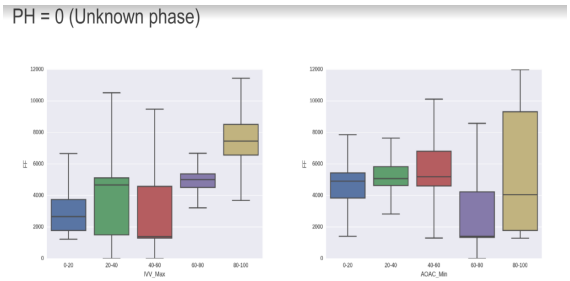


Fig 3 The joint distribution of fuel consumption and distance for takeoff, climb, cruise and approach phases. **The Climb and cruise phases have the largest fuel consumption**, but the relationship of fuel consumed with distance has a very strong correlation. On the other hand, the takeoff and approach phases have significant deviation from a linear relationship which hints at possible improvements.

A breakdown of the analysis of fuel vs distance into different phases gives an idea about **which phases need to be and can be optimized for low fuel consumption**. In fig 3, we show joint distribution of fuel and distance for different phases. This analysis has an implicit assumption that the flight **path is straight between source and destination**; which is a reasonable but not completely accurate assumption.
In the following subsections, the report will focus on **key differences in the predictors in different phases of flight**. As mentioned above, the **takeoff, climb, cruise and the approach** phases are the important phases as far as optimization of fuel consumption is concerned.



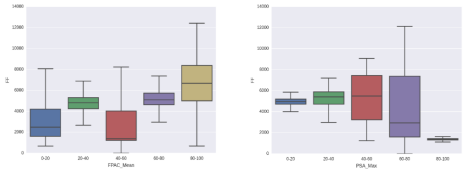


Fig 4 The relationship of fuel flow in the unknown phase with top individual predictors - *IVV_Max*, *AOAC_Min*, *FPAC_Mean* and *PSA_Max*. Here the x-axis is five equally sized groups of samples such that the first box corresponds to values between 0 and 20th percentile of predictor, the second box corresponds to values between 20th and 40th percentile and so on. Clearly, there is a significant relationship between fuel flow rate and these predictors but none of these four relationships are linear.

As mentioned before, PH=0 is actually a mini version of the entire dataset. The top individual predictors for this part of data were IVV_Max, AOAC_Min, FPAC_Mean and PSA_Max. The top combined feature for this phase was a combination of IVV_Mean, PT_Min and CAS_Max. The relationship of the important individual predictors with the fuel consumption is shown in the fig 4.

PH = 1 (Preflight)

The preflight phase is a small phase and **80% of the time in this phase does not consume any fuel.** The average fuel consumption for the total time in this phase is just **61** which is very small and insignificant compared to the average of all flight phases (4219). The top four individual predictors of fuel consumption for this phase are **VIB_1_Mean, OIT_4, OIPL and N1T_Max** and the top combined predictor was a combination of **SAT, PAC_Mean, TAS_Mean and IVV_Min.**

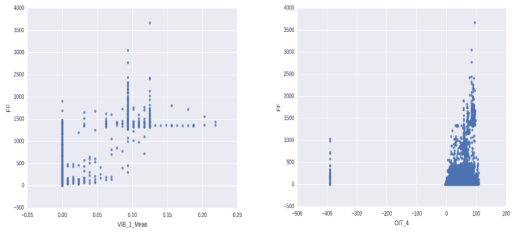


Fig 5 The relationship of fuel flow in the preflight phase with top individual predictors - **VIB_1_Mean, OIT_4, OIPL andN1T_Max**. Because of a skewed distribution of fuel flow (most of the samples having zero fuel flow), the distribution of fuel flow as a function of percentiles of predictors was **not informative**

PH = 2 (Taxi)

There is a significant amount of time spent in this phase (**20%** of the time in flights data given in this competition). The average fuel consumption (**1334**) is just above half of the overall average,and thus this phase accounts for about **6% of the total fuel consumed**. The top four individual predictors of fuel consumption for this phase are **LONG_Max, OIPL, VIB_1_Mean and FQTY**_4and the top combined predictor is a combination of **PT_Mean, OIPL and FPAC_Min.** The following graph shows the relationship of fuel flow rate with respect to these four individual predictors.

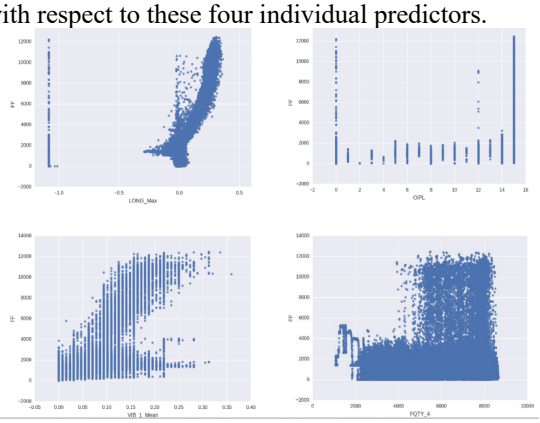


Fig 6 The relationship of fuel flow in the taxi phase with top individual predictors - **LONG_Max, OIPL, VIB_1_Mean and FQTY_4.Only** LONG_Max shows a clear relationship, for which the fuel flow rate increases with increase in the value of LONG_Max. The fuel flow was VIB_1_Mean also has a faint visible pattern, but the other two graphs (OIPL and FQTY_4) are very uninformative. The validation root mean squared error in this phase was about 120, so the absence of a clear visual relationship between fuel flow and top predictors is indicative of strong non-linear interaction between predictors.

PH = 3 (Takeoff)

The average fuel flow rate in this stage is very large **(6077)** but since this phase lasts for a **small time**, it accounts for only about **2.5%** of the total fuel consumption. The top individual predictors are **LONG_Max, FLAP, IVV_Max and AOAC_Min**. The best combination of features for prediction of fuel consumption in this phase was **GS_Min, N1T_Max, IVV_Min and FPAC_Min.**

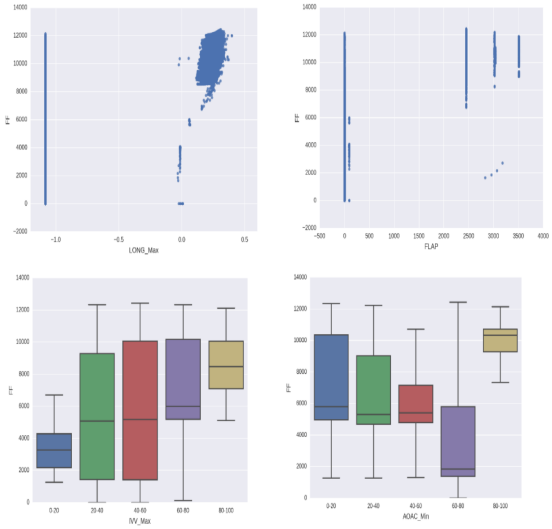


Fig 7 The relationship of fuel flow in the takeoff phase with top individual predictors - **LONG_Max, FLAP, IVV_Max and AOAC_Min**.

The first two graphs are shown as simple scatterplots, while the latter two are shown as distribution of fuel flow rate in different percentile ranges of predictors. The graph format is chosen for best possible clarity.
The relationship of fuel flow rate with LONG_Max and IVV_Max (somewhat linearly increasing) and AOAC_Min (nonlinear relationship) are visible from the plots in fig 7.

PH = 4 (Climb)

This is the phase with the **highest average fuel flow rate (7254)** and it accounts for about **33%** of the total fuel consumption during a flight. The most important predictors for this phase are **PT_Mean, LONG_Max, FLAP and ALTR_Mean**. The fuel flow rate increases with increase in PT_Mean, LONG_Max, and ALTR_Mean as is clear from boxplots in fig 8 below. However, the relationship of fuel flow rate with FLAP is not clear from a plot, which implies a nonlinear relation or a strong interaction of FLAP with other features.

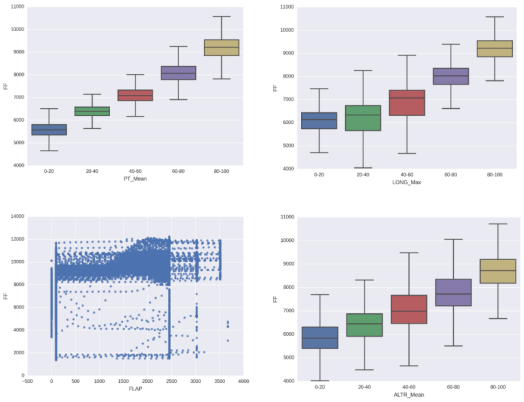


Fig 8 The relationship of fuel flow in the climb phase with top individual predictors - **PT_Mean, LONG_Max, FLAP and ALTR_Mean**. The third graph is shown as simple scatterplots, while the other three are shown as distribution of fuel flow rate in different percentile ranges of predictors. The graph format is chosen for best possible clarity.**The best combination of features as a predictor for climb phase was LONG_Max, OIT_4 and PT_Max.**

PH = 5 (Cruise)

This is the **longest phase of flights**, and constitutes on an average **32%** of total flight time, and so the total fuel consumption in this phase (~**37.8%**) is more than any other phase. The average fuel consumption **(4966) i**s smaller than the **takeoff and climb phase** because the average altitude is very high, and the air is thin. Thus the work done against the drag of the atmosphere is small in the cruise phase, even if the ground speed is very high. The important individual predictors of fuel consumption in this phase are **CASS, N1T_Max, LONG_Max and**

FPAC_Max, whereas the combined features which could best predict the fuel flow is a group of **PT_Min, CAS_Min, PI_Min and LONG_Min.**

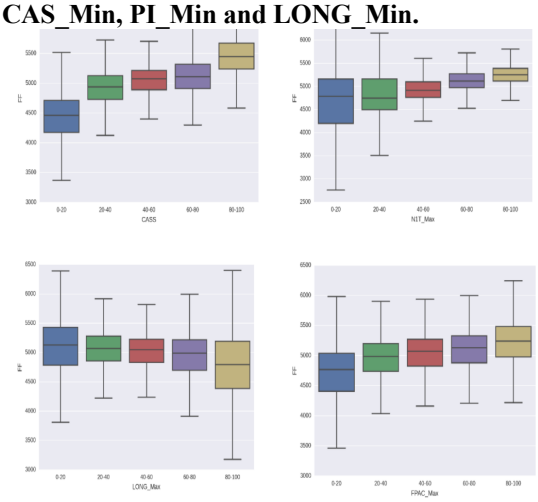


Fig 9 The distribution of fuel flow rate for cruise phase in different percentile ranges of top individual predictors - **CASS, N1T_Max, LONG_Max and FPAC_Max**. The fuel flow rate slightly increases with increase in **CASS, N1T_Max and FPAC_Max**, whereas it decreases slightly with increase in LONG_Max.

PH = 6 (Approach)

At approach, the average fuel consumption **(2940)** is much smaller than **climb, takeoff** or **cruise.**But this phase still accounts for about **13.6%** of total fuel consumed, as the average time duration of this phase is significant (**19.5% of total flight time**). The best individual predictors for this phase are **N1T_Max, MACH_Min, LONG_Max and IVV_Min**. There was no combination of four or less features which could explain even **40%** of the variance in fuel consumption for this phase. Fig 10 below shows a rather weak relationship between fuel flow rate and the top individual predictors.

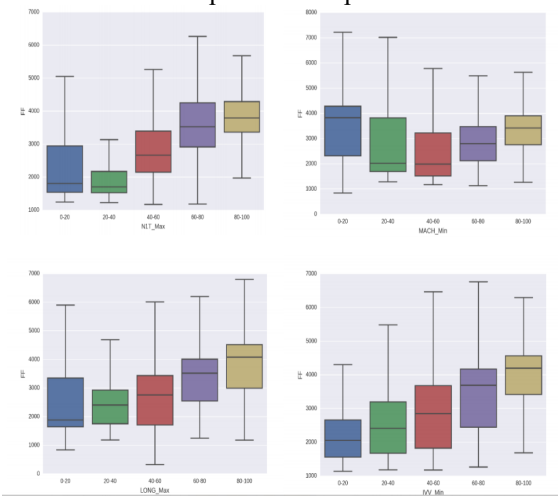


Fig 10 The distribution of fuel flow rate for approach phase in different percentile ranges of top individual predictors -**N1T_Max, MACH_Min, LONG_Max and IVV_Min**.

The fuel flow rate slightly increases with increase in N1T_Max,LONG_Max and IVV_Min, whereas it first decreases and then slightly increases with increase in MACH_Min. The visual dependence in each of these graphs is very poor.

PH = 7 (Rollout)

This phase consists of the **smallest amount of time and smallest total fuel consumption**, which is just **0.12%** of the total fuel consumed. Thus this phase does **not** have any **meaningful influence** on the amount of fuel used. Nevertheless, we did modelling to predict fuel flow for this phase and found that the top predictors were **LONG_Max, FADS, FPAC_Mean and CAS_Mean**. The relationship of fuel flow rate with LONG_Max, and FPAC_Mean is shown as scatterplot in fig 11, and those with FADS and CAS_Mean are shown as boxplots with various values and various percentile ranges of predictors respectively.

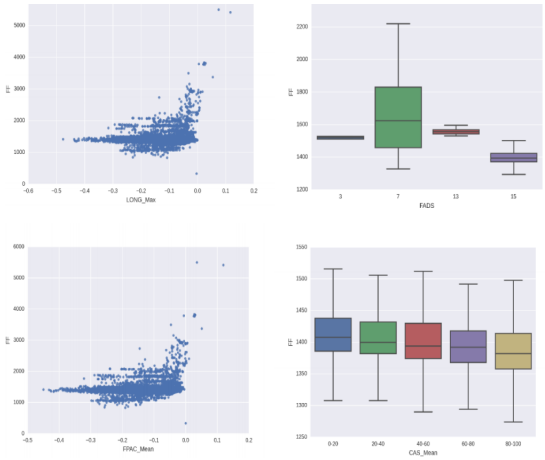


Fig 11 The distribution of fuel flow rate for rollout phase in different percentile ranges of top individual predictors **-LONG_Max, FADS, FPAC_Mean and CAS_Mean**. The dependence of fuel flow on these predictors is not visually clear from the graphs. The format of graph (scatter plot vs boxplot) is chosen for best possible clarity.

2.7 CONCLUSION

The most important phases for **optimizing fuel consumption** are **climb, cruise and approach**. The overall important features are **rate of change of altitude, longitudinal acceleration** and **ground speed**. The clearest visible trend between predictors and fuel flow rate is in the **climb phase**. In other phases, some of the predictors have a weakly visible trend, but since the root mean squared error is small, it is assumed that the features have strong nonlinear interactions which are not clearly visible in simple plots.

2.8 REFERENCES

- Yashovardhan S. Chati*, Hamsa Balakrishnan*,
  "**STATISTICAL MODELING OF AIRCRAFT ENGINE FUEL FLOW RATE**" , International Council of the Aeronautical Sciences, 2016 , 10 pages

- **Predict fuel consumption of airplanes** , Magic Data ,

- **Predict fuel consumption of airplanes during different phases of flight ,** Crowdanalytix

- **Data Science With Python course** , Datacamp ,

- **Machine Learning Nanodegree Program** , Udacity

- **Data Science for all** channel , Youtube

1. TV, halftime shows, and the Big Game

Whether or not you like American football, the Super Bowl is a spectacle. There is always a little something for everyone. For the die-hard fans, there is the game itself with blowouts, comebacks, and controversy. For the not so die-hard fans, there are the ridiculously expensive ads that are hilarious, gut-wrenching, thought-provoking, and sometimes weird. And of course, there are the halftime shows with the biggest musicians in the world entertaining us by riding a giant mechanical tiger (https://youtu.be/ZD1Qrle--_Y?t=14) or leaping from the roof of the stadium (https://youtu.be/mjrdywp5nyE?t=62). It is a grand show! In this notebook, we're going to explore how some of these elements interact with each other. After exploring and cleaning the data, we're going to answer questions like:

- What are the most extreme game outcomes?
- How does the score difference affect television viewership?
- How have viewership, TV ratings, and advertisement costs evolved?
- Who are the most prolific musicians in terms of halftime show performances?

Left Shark Steals The Show (https://www.flickr.com/photos/huntleypaton/16464994135/in/photostream/). Katy Perry performing at halftime of Super Bowl XLIX. Photo by Huntley Paton. Attribution-ShareAlike 2.0 Generic (CC BY-SA 2.0).

The dataset we'll use was scraped (https://en.wikipedia.org/wiki/Web_scraping) and polished from Wikipedia. It is made up of three CSV files, one with game data (https://en.wikipedia.org/wiki/List_of_Super_Bowl_champions), one with TV data (https://en.wikipedia.org/wiki/Super_Bowl_television_ratings), and one with halftime musician data (https://en.wikipedia.org/wiki/List_of_Super_Bowl_halftime_shows) for all 52 Super Bowls through 2018.

In [12]:

```
# Load packages
library(tidyverse)

# Load the CSV data
super_bowls  <-  read_csv("datasets/super_bowls.csv")
tv  <-  read_csv("datasets/tv.csv")
halftime_musicians  <-  read_csv("datasets/halftime_musicians.csv")

# Display the first six rows of each tibble
head(super_bowls)
head(tv)
head(halftime_musicians)
```

```
Parsed with column specification:
cols(
  date = col_date(format = ""),
  super_bowl = col_double(),
  venue = col_character(),
  city = col_character(),
  state = col_character(),
  attendance = col_double(),
  team_winner = col_character(),
  winning_pts = col_double(),
  qb_winner_1 = col_character(),
  qb_winner_2 = col_character(),
  coach_winner = col_character(),
  team_loser = col_character(),
  losing_pts = col_double(),
  qb_loser_1 = col_character(),
  qb_loser_2 = col_character(),
  coach_loser = col_character(),
  combined_pts = col_double(),
  difference_pts = col_double()
)
Parsed with column specification:
cols(
  super_bowl = col_double(),
  network = col_character(),
  avg_us_viewers = col_double(),
  total_us_viewers = col_double(),
  rating_household = col_double(),
  share_household = col_double(),
  rating_18_49 = col_double(),
  share_18_49 = col_double(),
  ad_cost = col_double()
)
Parsed with column specification:
cols(
  super_bowl = col_double(),
  musician = col_character(),
  num_songs = col_double()
)
```

A tibble: 6 x 18

date	super_bowl	venue	city	state	attendance	team_winner	winning_pts	qb_winner_1	qb_winner_2	coach_winner
<date>	<dbl>	<chr>	<chr>	<chr>	<dbl>	<chr>	<dbl>	<chr>	<chr>	<chr>
2018-02-04	52	U.S. Bank Stadium	Minneapolis	Minnesota	67612	Philadelphia Eagles	41	Nick Foles	NA	Doug Pederson
2017-02-05	51	NRG Stadium	Houston	Texas	70807	New England Patriots	34	Tom Brady	NA	Bill Belichick
2016-02-07	50	Levi's Stadium	Santa Clara	California	71088	Denver Broncos	24	Peyton Manning	NA	Gary Kubiak
2015-02-01	49	University of Phoenix Stadium	Glendale	Arizona	70288	New England Patriots	28	Tom Brady	NA	Bill Belichick
2014-02-02	48	MetLife Stadium	East Rutherford	New Jersey	82529	Seattle Seahawks	43	Russell Wilson	NA	Pete Carroll
2013-02-03	47	Mercedes-Benz Superdome	New Orleans	Louisiana	71024	Baltimore Ravens	34	Joe Flacco	NA	John Harbaugh

A tibble: 6 x 9

super_bowl	network	avg_us_viewers	total_us_viewers	rating_household	share_household	rating_18_49	share_18_49	ad_cost
<dbl>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
52	NBC	103390000	NA	43.1	68	33.4	78	5000000
51	Fox	111319000	172000000	45.3	73	37.1	79	5000000
50	CBS	111864000	167000000	46.6	72	37.7	79	5000000
49	NBC	114442000	168000000	47.5	71	39.1	79	4500000
48	Fox	112191000	167000000	46.7	69	39.3	77	4000000
47	CBS	108693000	164100000	46.3	69	39.7	77	4000000

A tibble: 6 x 3

super_bowl	musician	num_songs
<dbl>	<chr>	<dbl>
52	Justin Timberlake	11
52	University of Minnesota Marching Band	1
51	Lady Gaga	7
50	Coldplay	6
50	Beyonc<U+00E9>	3
50	Bruno Mars	3

In [13]:

```
# These packages need to be loaded in the first @tests cell.
library(testthat)
library(IRkernel.testthat)

soln_super_bowls  <-  read_csv("datasets/super_bowls.csv")
soln_tv = read_csv("datasets/tv.csv")
soln_halftime_musicians = read_csv("datasets/halftime_musicians.csv")

run_tests({
    test_that("packages are loaded", {
        expect_true("tidyverse" %in% .packages(), info = "Did you load the tidyverse package?")
      })
    test_that("The .csv were read in correctly", {
        expect_is(super_bowls, "tbl_df", info = "Did you read in super_bowls.csv with read_csv?")
        expect_equal(super_bowls, soln_super_bowls,
                     info = "super_bowls contains the wrong values. Did you load the correct .csv file?")

        expect_is(tv, "tbl_df", info = "Did you read in tv.csv with read_csv?")
        expect_equal(tv, soln_tv,
                     info = "tv contains the wrong values. Did you load the correct .csv file?")

        expect_is(halftime_musicians, "tbl_df", info = "Did you read in halftime_musicians.csv with read_csv?")
        expect_equal(halftime_musicians, soln_halftime_musicians,
                     info = "halftime_musicians contains the wrong values. Did you load the correct .csv file?")
    })
})
```

```
Attaching package: 'testthat'

The following object is masked from 'package:dplyr':

    matches

The following object is masked from 'package:purrr':

    is_null

The following object is masked from 'package:tidyr':

    matches

Parsed with column specification:
cols(
  date = col_date(format = ""),
  super_bowl = col_double(),
  venue = col_character(),
  city = col_character(),
  state = col_character(),
  attendance = col_double(),
  team_winner = col_character(),
  winning_pts = col_double(),
  qb_winner_1 = col_character(),
  qb_winner_2 = col_character(),
  coach_winner = col_character(),
  team_loser = col_character(),
  losing_pts = col_double(),
  qb_loser_1 = col_character(),
  qb_loser_2 = col_character(),
  coach_loser = col_character(),
  combined_pts = col_double(),
  difference_pts = col_double()
)
Parsed with column specification:
cols(
  super_bowl = col_double(),
  network = col_character(),
  avg_us_viewers = col_double(),
  total_us_viewers = col_double(),
  rating_household = col_double(),
  share_household = col_double(),
  rating_18_49 = col_double(),
  share_18_49 = col_double(),
  ad_cost = col_double()
)
Parsed with column specification:
cols(
  super_bowl = col_double(),
  musician = col_character(),
  num_songs = col_double()
)

2/2 tests passed
```

2. Taking note of dataset issues

From the quick look at the Super Bowl game data, we can see that the dataset appears whole except for missing values in the backup quarterback columns (`qb_winner_2` and `qb_loser_2`), which make sense given most starting QBs in the Super Bowl (`qb_winner_1` and `qb_loser_1`) play the entire game.

From the visual inspection of TV and halftime musicians data, there is only one missing value displayed, but I've got a hunch there are more. The first Super Bowl was played on January 15, 1967, and I'm guessing some data (e.g., the number of songs performed) probably weren't tracked reliably over time. Wikipedia is great but not perfect.

Looking at a summary of the datasets shows us that there are multiple columns with null values.

In [14]:

```
# Summary of the TV data
summary(tv)

# Summary of the halftime musician data
summary(halftime_musicians)
```

```
   super_bowl      network          avg_us_viewers      total_us_viewers
 Min.   : 1.00   Length:53          Min.   : 24430000   Min.   : 51180000
 1st Qu.:13.00   Class :character   1st Qu.: 73852000   1st Qu.:142900000
 Median :26.00   Mode  :character   Median : 85240000   Median :153400000
 Mean   :26.02                      Mean   : 80709585   Mean   :148872000
 3rd Qu.:39.00                      3rd Qu.: 92570000   3rd Qu.:165550000
 Max.   :52.00                      Max.   :114442000   Max.   :172000000
                                                        NA's   :38
 rating_household share_household  rating_18_49    share_18_49
 Min.   :18.5     Min.   :36.00   Min.   :33.40   Min.   :77.00
 1st Qu.:41.3     1st Qu.:63.00   1st Qu.:36.90   1st Qu.:77.25
 Median :43.3     Median :67.00   Median :37.90   Median :78.50
 Mean   :42.7     Mean   :66.38   Mean   :38.01   Mean   :78.17
 3rd Qu.:46.0     3rd Qu.:71.00   3rd Qu.:39.50   3rd Qu.:79.00
 Max.   :49.1     Max.   :78.00   Max.   :41.20   Max.   :79.00
                                  NA's   :38      NA's   :47
    ad_cost
 Min.   :  37500
 1st Qu.: 185000
 Median : 850000
 Mean   :1456712
 3rd Qu.:2385365
 Max.   :5000000


   super_bowl      musician           num_songs
 Min.   : 1.00   Length:134         Min.   : 1.000
 1st Qu.:17.25   Class :character   1st Qu.: 1.000
 Median :31.50   Mode  :character   Median : 2.000
 Mean   :29.09                      Mean   : 2.955
 3rd Qu.:41.00                      3rd Qu.: 3.250
 Max.   :52.00                      Max.   :11.000
                                    NA's   :46
```

In [15]:

```
run_tests({
    test_that("the sky is blue", {
        expect_true("blue sky" == "blue sky",
                    info = "Not testing anything.")
    })
})
```

```
1/1 tests passed
```

3. Combined points distribution

In the TV data, the following columns have a lot of missing values:

- `total_us_viewers` (amount of U.S. viewers who watched at least some part of the broadcast)
- `rating_18_49` (average % of U.S. adults 18-49 who live in a household with a TV that were watching for the entire broadcast)
- `share_18_49` (average % of U.S. adults 18-49 who live in a household with a TV *in use* that were watching for the entire broadcast)

In halftime musician data, there are missing numbers of songs performed (`num_songs`) for about a third of the musicians.

There are a lot of potential reasons for missing values. Were the data ever tracked? Would the research effort to fill in the gaps be worth it? Maybe. Watching every Super Bowl halftime show to get song counts could be pretty fun. But we don't have time to do that now! Let's take note of where the datasets are not perfect and start uncovering some insights.

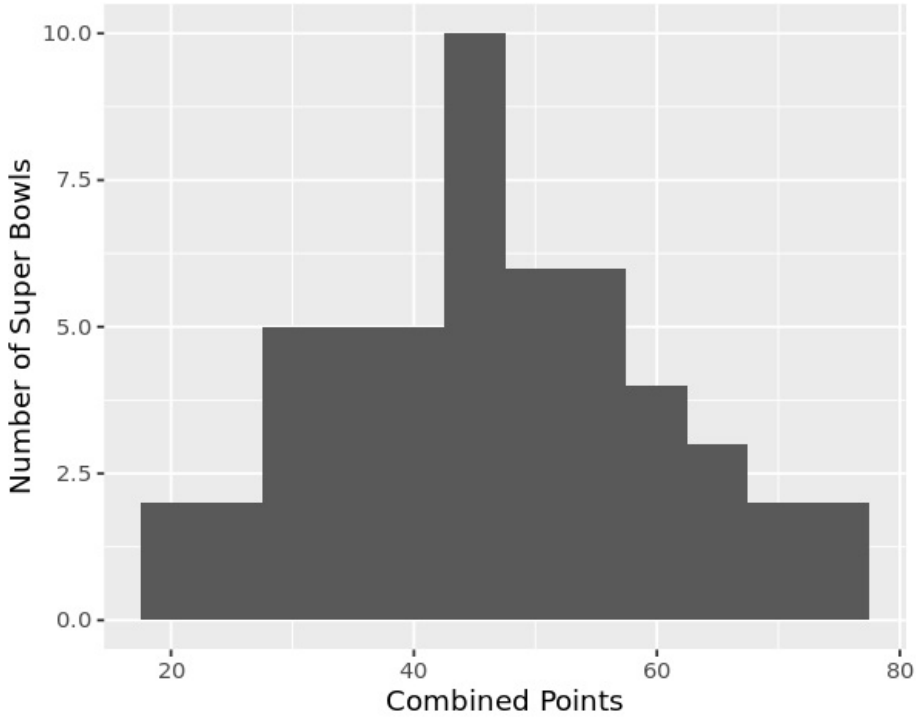
We'll start by visualizing the distribution of combined points for each Super Bowl. Let's also find the Super Bowls with the highest and lowest scores.

In [16]:

```
# Reduce the size of the plots
options(repr.plot.width = 5, repr.plot.height = 4)

# Plot a histogram of combined points
ggplot(super_bowls, aes(combined_pts)) +
 geom_histogram(binwidth = 5) +
 labs(x = "Combined Points", y = "Number of Super Bowls")

# Display the highest- and lowest-scoring Super Bowls
super_bowls  %>%
 filter(combined_pts > 70 | combined_pts < 25)
```

A spec_tbl_df: 5 x 18

date	super_bowl	venue	city	state	attendance	team_winner	winning_pts	qb_winner_1	qb_winner_2	coach_winner	t
<date>	<dbl>	<chr>	<chr>	<chr>	<dbl>	<chr>	<dbl>	<chr>	<chr>	<chr>	
2018-02-04	52	U.S. Bank Stadium	Minneapolis	Minnesota	67612	Philadelphia Eagles	41	Nick Foles	NA	Doug Pederson	
1995-01-29	29	Joe Robbie Stadium	Miami Gardens	Florida	74107	San Francisco 49ers	49	Steve Young	NA	George Seifert	
1975-01-12	9	Tulane Stadium	New Orleans	Louisiana	80997	Pittsburgh Steelers	16	Terry Bradshaw	NA	Chuck Noll	
1973-01-14	7	Memorial Coliseum	Los Angeles	California	90182	Miami Dolphins	14	Bob Griese	NA	Don Shula	V
1969-01-12	3	Orange Bowl	Miami	Florida	75389	New York Jets	16	Joe Namath	NA	Weeb Ewbank	



In [17]:

```
stud_plot <- last_plot()
soln_plot  <- ggplot(soln_super_bowls, aes(combined_pts)) +
 geom_histogram(binwidth = 5) +
 labs(x = "Combined Points", y = "Number of Super Bowls")

run_tests({
    test_that("the plot was created correctly", {
        expect_equal(stud_plot, soln_plot,
                     info = "The historgram of combined points was not created correctly. \nCheck that combined_p
ts is the aesthetic and the binwidth = 5.")
    })
})
```

```
1/1 tests passed
```

4. Point difference distribution

Most of the combined scores are between 40 and 50 points, with the extremes being roughly equal distance away in opposite directions. At the highest combined scores of 74 and 75, are two games featuring dominant quarterback performances. One happened last year - Super Bowl LII when Tom Brady's Patriots lost to Nick Foles' underdog Eagles 33 to 41, for a combined score of 74.

On the other end of the spectrum, we have Super Bowl III and VII, which featured tough defenses that dominated the games. We also have Super Bowl IX in New Orleans in 1975, whose 16-6 score can be attributed to inclement weather. Overnight rain made the field slick, and it was cold (46 °F / 8 °C), making it hard for the Steelers and Vikings to do much offensively. This was the second-coldest Super Bowl ever and the last to be played in inclement weather for over 30 years. The NFL realized people like points, I guess.

UPDATE: In Super Bowl LIII in 2019, the Patriots and Rams broke the record for the lowest-scoring Super Bowl with a combined score of 16 points (13-3 for the Patriots).

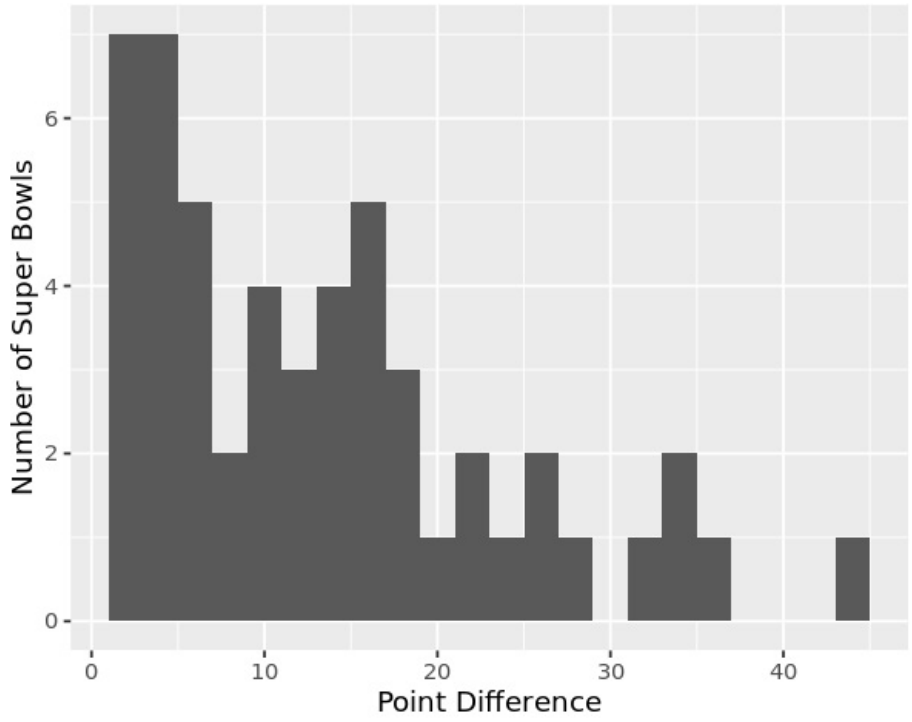
Now let's take a look at the point difference between teams in each Super Bowl.

In [18]:

```
# Reduce the size of the plots
options(repr.plot.width = 5, repr.plot.height = 4)

# Plot a histogram of point differences
ggplot(super_bowls, aes(difference_pts)) +
 geom_histogram(binwidth = 2) +
 labs(x = "Point Difference", y = "Number of Super Bowls")

# Display the closest game and largest blow out
super_bowls  %>%
 filter(difference_pts == min(difference_pts) | difference_pts == max(difference_pts))
```

A spec_tbl_df: 2 x 18

date	super_bowl	venue	city	state	attendance	team_winner	winning_pts	qb_winner_1	qb_winner_2	coach_winner	t
<date>	<dbl>	<chr>	<chr>	<chr>	<dbl>	<chr>	<dbl>	<chr>	<chr>	<chr>	
1991-01-27	25	Tampa Stadium	Tampa	Florida	73813	New York Giants	20	Jeff Hostetler	NA	Bill Parcells	E
1990-01-28	24	Louisiana Superdome	New Orleans	Louisiana	72919	San Francisco 49ers	55	Joe Montana	NA	George Seifert	



In [19]:

```
stud_plot <- last_plot()
soln_plot  <- ggplot(soln_super_bowls, aes(difference_pts)) +
 geom_histogram(binwidth = 2) +
 labs(x = "Point Difference", y = "Number of Super Bowls")

run_tests({
    test_that("the plot was created correctly", {
        expect_equal(stud_plot, soln_plot,
                     info = "The historgram of point differences was not created correctly.\nCheck that differenc
e_pts is the aesthetic and binwidth is correctly set.")
    })
})
```

```
1/1 tests passed
```

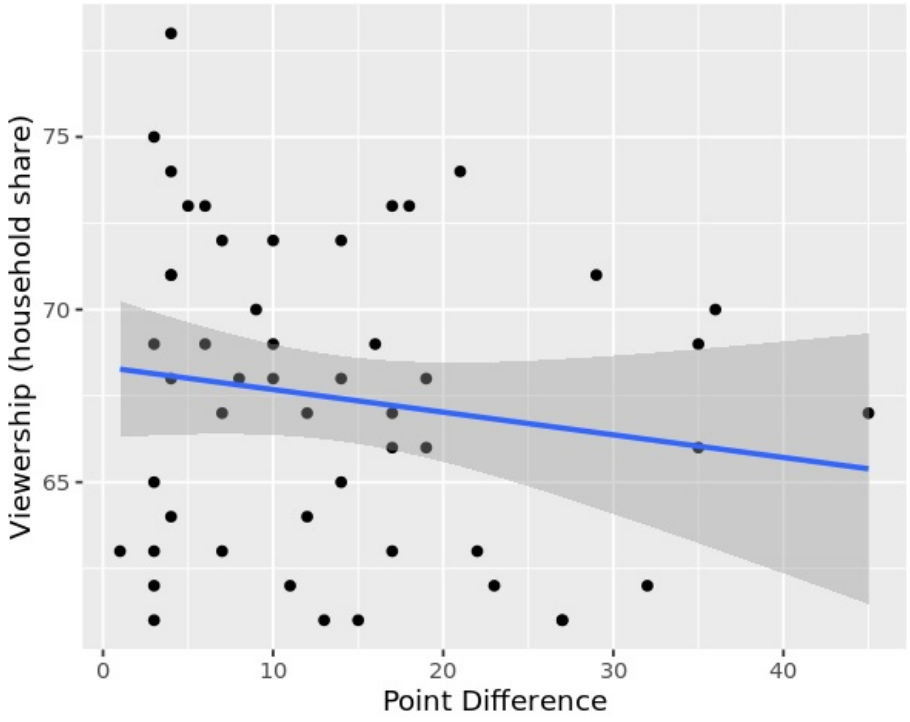
5. Do blowouts translate to lost viewers?

The vast majority of Super Bowls are close games. Makes sense. Both teams are the best in their conference if they've made it this far. The closest game ever was the Buffalo Bills' 1-point loss to the New York Giants in 1991, which is best remembered for Scott Norwood's last-second missed field goal attempt that went *wide right (https://www.youtube.com/watch?v=RPFZCGgjDSg)*, kicking off four Bills Super Bowl losses in a row. Poor Scott. The biggest point spread so far is 45 points (!) when Hall of Famer, Joe Montana, led the San Francisco 49ers to victory in 1990, one year before the closest game ever.

I remember watching the Seahawks crush the Broncos by 35 points (43-8) in 2014, which was a boring experience in my opinion. The game was never really close. I'm pretty sure we changed the channel at the end of the third quarter. Let's combine the game data and TV data to see if this is a universal phenomenon. Do large point differences translate to lost viewers? We can plot household share (https://en.wikipedia.org/wiki/Nielsen_ratings) *(average percentage of U.S. households with a TV in use that were watching for the entire broadcast)* vs. point difference to find out.

In [20]:

```
# Filter out Super Bowl I and join the game data and TV data
games_tv <- tv  %>%
 filter(super_bowl != 1)  %>%
 inner_join(super_bowls, by = "super_bowl")

# Create a scatter plot with a linear regression model
ggplot(games_tv, aes(difference_pts, share_household)) +
 geom_point() +
 geom_smooth(method = "lm") +
 labs(x = "Point Difference", y = "Viewership (household share)")
```



In [21]:

```
stud_plot <- last_plot()

soln_games_tv <- soln_tv  %>%
 filter(super_bowl != 1)  %>%
 inner_join(soln_super_bowls, by = "super_bowl")

soln_plot  <- ggplot(soln_games_tv, aes(difference_pts, share_household)) +
 geom_point() +
 geom_smooth(method = "lm") +
 labs(x = "Point Difference", y = "Viewership (household share)")

run_tests({
    test_that("games_tv was created correctly", {
        expect_equal(games_tv, soln_games_tv,
                     info = "games_tv was not created correctly. \nCheck that you use "!=" to remove SB one, and
joined to super_bowls.")
    })
    test_that("the plot was created correctly", {
        expect_equal(stud_plot, soln_plot,
                     info = "The scatterplot is not correct.\nCheck the x and y aesthetics and that you used meth
od = 'lm' in geom_smooth().")
    })
})
```

```
2/2 tests passed
```

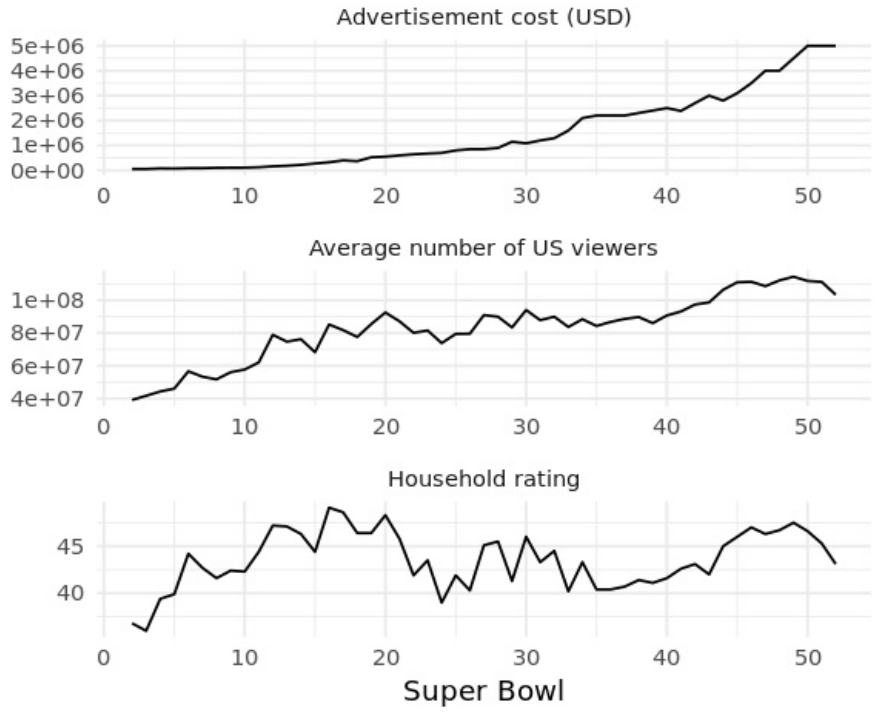
6. Viewership and the ad industry over time

The downward sloping regression line and the 95% confidence interval for that regression *suggest* that bailing on the game if it is a blowout is common. Though it matches our intuition, we must take it with a grain of salt because the linear relationship in the data is weak due to our small sample size of 52 games.

Regardless of the score, I bet most people stick it out for the halftime show, which is good news for the TV networks and advertisers. A 30-second spot costs a pretty $5 million (https://www.businessinsider.com/super-bowl-commercials-cost-more-than-eagles-quarterback-earns-2018-1) now, but has it always been that much? And how has the number of viewers and household ratings trended alongside advertisement cost? We can find out using line plots that share a "Super Bowl" x-axis.

In [22]:

```
# Convert the data format for plotting
games_tv_plot  <- games_tv %>%
    gather(key = "category", value = "value", avg_us_viewers, rating_household, ad_cost)  %>%
    mutate(cat_name = case_when(category == "avg_us_viewers" ~ "Average number of US viewers",
                                category == "rating_household" ~ "Household rating",
                                category == "ad_cost" ~ "Advertisement cost (USD)",
                                TRUE ~ as.character(category)))

# Plot the data
ggplot(games_tv_plot, aes(super_bowl, value)) +
 geom_line() +
 facet_wrap(~ cat_name, scales = "free", nrow = 3) +
 labs(x = "Super Bowl", y = "") +
 theme_minimal()
```



In [23]:

```
stud_plot <- last_plot()

soln_games_tv_plot  <- soln_games_tv %>%
    gather(key = "category", value = "value", avg_us_viewers, rating_household, ad_cost)  %>%
    mutate(cat_name = case_when(category == "avg_us_viewers" ~ "Average number of US viewers",
                                category == "rating_household" ~ "Household rating",
                                category == "ad_cost" ~ "Advertisement cost (USD)",
                                TRUE ~ as.character(category)))

soln_plot <- ggplot(soln_games_tv_plot, aes(super_bowl, value)) +
 geom_line() +
 facet_wrap(~ cat_name, scales = "free", nrow = 3) +
 labs(x = "Super Bowl", y = "") +
 theme_minimal()

run_tests({
    test_that("wide_games_tv was created correctly", {
        expect_equal(games_tv_plot, soln_games_tv_plot,
                     info = "wide_games_tv was not created correctly. \nYou need to gather the three columns you
are going to plot.")
        })
    test_that("the plot was created correctly", {
        expect_equal(stud_plot, soln_plot,
                     info = "The faceted line plot is not correct.\nCheck the x and y aesthetics and the theme.")
        })
})
```

```
2/2 tests passed
```

7. Halftime shows weren't always this great

We can see that the number of viewers increased before advertisement costs did. Maybe the networks weren't very data savvy and were slow to react? Makes sense since DataCamp didn't exist back then.

Another hypothesis: maybe halftime shows weren't as entertaining in the earlier years? The modern spectacle that is the Super Bowl has a lot to do with of big halftime acts. I went down a YouTube rabbit hole, and it turns out that older halftime shows were not quite the spectacle they are today. Some examples:

- Super Bowl XXVI (https://youtu.be/6wMXHxWO4ns?t=263) in 1992: A Frosty The Snowman rap performed by children.
- Super Bowl XXIII (https://www.youtube.com/watch?v=PKQTL1PYSag) in 1989: An Elvis impersonator who did magic tricks and didn't even sing one Elvis song.
- Super Bowl XXI (https://youtu.be/oSXMNbK2e98?t=436) in 1987: Tap dancing ponies. Okay, that was pretty awesome actually.

It turns out that Michael Jackson's Super Bowl XXVII performance, one of the most watched events in American TV history, was when the NFL realized that the having big-name halftime acts brought in more viewers. Let's look at the halftime acts before Michael Jackson brought the NFL and entertainment industry together.

In [24]:

```
# Filter and diplay halftime musicians before and including Super Bowl XXVII
(pre_MJ  <- halftime_musicians  %>%
 filter(super_bowl <= 27) )
```

A spec_tbl_df: 54 x 3

super_bowl	musician	num_songs
<dbl>	<chr>	<dbl>
27	Michael Jackson	5
26	Gloria Estefan	2
26	University of Minnesota Marching Band	NA
25	New Kids on the Block	2
24	Pete Fountain	1
24	Doug Kershaw	1
24	Irma Thomas	1
24	Pride of Nicholls Marching Band	NA
24	The Human Jukebox	NA
24	Pride of Acadiana	NA
23	Elvis Presto	7
22	Chubby Checker	2
22	San Diego State University Marching Aztecs	NA
22	Spirit of Troy	NA
21	Grambling State University Tiger Marching Band	8
21	Spirit of Troy	8
20	Up with People	NA
19	Tops In Blue	NA
18	The University of Florida Fightin' Gator Marching Band	7
18	The Florida State University Marching Chiefs	7
17	Los Angeles Unified School District All City Honor Marching Band	NA
16	Up with People	NA
15	The Human Jukebox	NA
15	Helen O'Connell	NA
14	Up with People	NA
14	Grambling State University Tiger Marching Band	NA
13	Ken Hamilton	NA
13	Gramacks	NA
12	Tyler Junior College Apache Band	NA
12	Pete Fountain	NA
12	Al Hirt	NA
11	Los Angeles Unified School District All City Honor Marching Band	NA
10	Up with People	NA

9	Mercer Ellington	NA
9	Grambling State University Tiger Marching Band	NA
8	University of Texas Longhorn Band	NA
8	Judy Mallett	NA
7	University of Michigan Marching Band	NA
7	Woody Herman	NA
7	Andy Williams	NA
6	Ella Fitzgerald	NA
6	Carol Channing	NA
6	Al Hirt	NA
6	United States Air Force Academy Cadet Chorale	NA
5	Southeast Missouri State Marching Band	NA
4	Marguerite Piazza	NA
4	Doc Severinsen	NA
4	Al Hirt	NA
4	The Human Jukebox	NA
3	Florida A&M University Marching 100 Band	NA
2	Grambling State University Tiger Marching Band	NA
1	University of Arizona Symphonic Marching Band	NA
1	Grambling State University Tiger Marching Band	NA
1	Al Hirt	NA

In [25]:

```
soln_pre_MJ  <- soln_halftime_musicians  %>%
 filter(super_bowl <= 27)

run_tests({
    test_that("pre_MJ created correctly", {
        expect_equal(pre_MJ, soln_pre_MJ,
                     info = "pre_MJ was not created correctly. \nCheck that you filtered for super_bowl <= 27.")
        })
    })
```

```
1/1 tests passed
```

8. Who has the most halftime show appearances?

Now that's a lot of marching bands! There was also the American jazz clarinetist, Pete Fountain, and Miss Texas 1973 played the violin. Nothing against those performers - they are just simply not Beyoncé (https://www.youtube.com/watch?v=sulg9kTGBVI). To be fair, no one is.

Let's find all the musicians who performed at the Super Bowl more than once and count their performances.

In [26]:

```
# Display the musicians who performed more than once
halftime_musicians  %>%
    count(musician, sort = TRUE)  %>%
    filter(n > 1)
```

A spec_tbl_df: 14 x 2

musician	n
<chr>	<int>
Grambling State University Tiger Marching Band	6
Al Hirt	4
Up with People	4
The Human Jukebox	3
Beyonc<U+00E9>	2
Bruno Mars	2
Florida A&M University Marching 100 Band	2
Gloria Estefan	2
Justin Timberlake	2
Los Angeles Unified School District All City Honor Marching Band	2
Nelly	2
Pete Fountain	2
Spirit of Troy	2
University of Minnesota Marching Band	2

In [27]:

```
out <- .Last.value

soln_test  <- soln_halftime_musicians  %>%
    count(musician, sort = TRUE)  %>%
    filter(n > 1)


run_tests({
    test_that("You found the musicians with more than 1 appearance", {
        expect_equal(out, soln_test,
                     info = "The display is not correct. \nCheck that you filtered for musicians who appeared mor
e than once, and arranged the counts in descending order.")
        })
    })
```

```
1/1 tests passed
```

9. Who performed the most songs in a halftime show?

The world-famous Grambling State University Tiger Marching Band (https://www.youtube.com/watch?v=RL_3oqpHiDg) takes the crown with six appearances. Beyoncé, Justin Timberlake, Nelly, and Bruno Mars are the only post-Y2K musicians with multiple appearances (two each).

Now let's look at the number of songs performed in a halftime show. From our previous inspections, the `num_songs` column has a lot of missing values:

- A lot of the marching bands don't have `num_songs` entries.
- For non-marching bands, there is a lot of missing data before Super Bowl XX.

Let's filter out marching bands by using a string match for "Marching" and "Spirit" (a common naming convention for marching bands is "Spirit of [something]"). We'll only keep data from Super Bowls XX and later to address the missing data issue, and *then* let's see who performed the most number of songs.

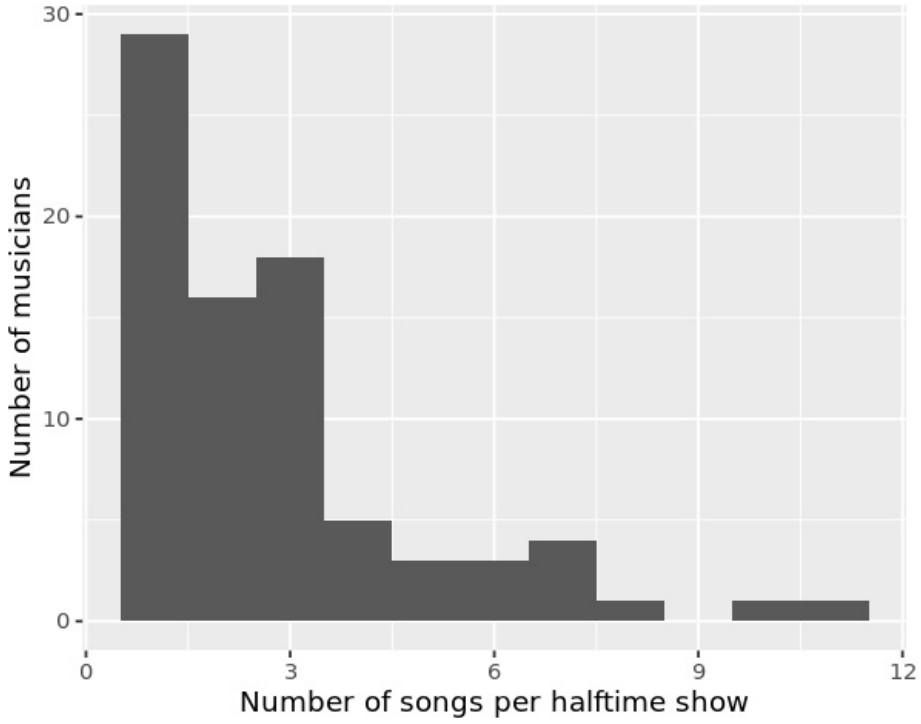
In [28]:

```
# Remove marching bands and data before Super Bowl XX
musicians_songs  <- halftime_musicians  %>%
    filter(!str_detect(musician, "Marching"),
           !str_detect(musician, "Spirit"),
          super_bowl > 20)

# Plot a histogram of the number of songs per performance
ggplot(musicians_songs, aes(num_songs)) +
    geom_histogram(binwidth = 1) +
    labs(x = "Number of songs per halftime show", y = "Number of musicians")

# Display the musicians with more than four songs per show
musicians_songs  %>%
    filter(num_songs > 4)  %>%
    arrange(desc(num_songs))
```

```
Warning message:
"Removed 2 rows containing non-finite values (stat_bin)."
```

A spec_tbl_df: 13 x 3

super_bowl	musician	num_songs
<dbl>	<chr>	<dbl>
52	Justin Timberlake	11
30	Diana Ross	10
49	Katy Perry	8
51	Lady Gaga	7
47	Beyonc<U+00E9>	7
41	Prince	7
23	Elvis Presto	7
50	Coldplay	6
48	Bruno Mars	6
45	The Black Eyed Peas	6
46	Madonna	5
44	The Who	5
27	Michael Jackson	5



In [29]:

```
stud_plot  <-  last_plot()

soln_musicians_songs  <-soln_halftime_musicians  %>%
    filter(!str_detect(musician, "Marching"),
           !str_detect(musician, "Spirit"),
          super_bowl > 20)

soln_plot <- ggplot(soln_musicians_songs, aes(num_songs)) +
    geom_histogram(binwidth = 1) +
    labs(x = "Number of songs per halftime show", y = "Number of musicians")

run_tests({
    test_that("musicians_songs is correct", {
        expect_equal(musicians_songs, soln_musicians_songs,
                     info = "musicians_songs is not correct. \nCheck the filtering.")
    })
    test_that("the histogram is correct", {
        expect_equal(stud_plot, soln_plot, info = "The plot is not correct. Did you use num_songs as the asetheti
c with a binwidth = 1?")
    })
})
```

```
2/2 tests passed
```

10. Conclusion

Most non-band musicians do 1 to 3 songs per halftime show. It's important to note that the duration of the halftime show is fixed (roughly 12 minutes) so songs per performance is more a measure of how many hit songs you have (cram as many hit songs in as you can!). Timberlake went off in 2018 with 11 songs! Wow! Diana Ross comes in second with a ten song medley in 1996.

In this notebook, we loaded, cleaned, and explored Super Bowl game, television, and halftime show data. We visualized the distributions of combined points, point differences, and halftime show performances using histograms. We used line plots to see how advertisement cost increases lagged behind viewership increases. And, we discovered that blowouts appear to lead to a drop in viewership.

This year's Big Game will be here before you know it. Who do you think will win Super Bowl LIII?

UPDATE: *Spoiler alert (https://en.wikipedia.org/wiki/Super_Bowl_LIII)*.

In [30]:

```
# 2018-2019 conference champions
patriots  <-  "New England Patriots"
rams  <-  "Los Angeles Rams"

# Who will win Super Bowl LIII?
super_bowl_LIII_winner  <-  "Elvis"
paste('The winner of Super Bowl LIII will be the', super_bowl_LIII_winner)
```

'The winner of Super Bowl LIII will be the Elvis'

In [31]:

```
run_tests({
    test_that("You found a magical unicorn!", {
        expect_is(patriots, "character")
        expect_is(rams, "character")
        })
    })
```

```
1/1 tests passed
```

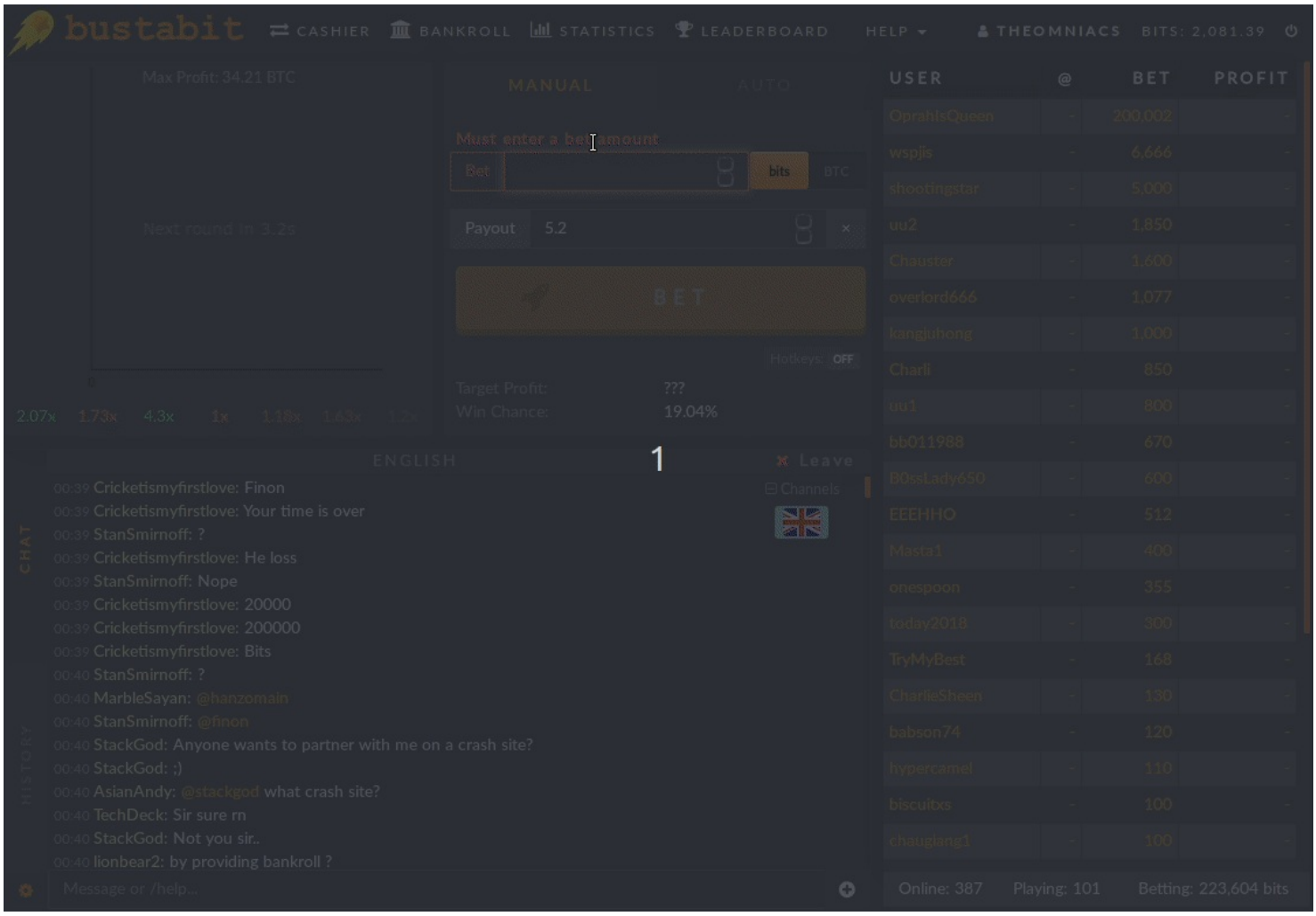
1. A preliminary look at the Bustabit data

The similarities and differences in the behaviors of different people have long been of interest, particularly in psychology and other social science fields. Understanding human behavior in particular contexts can help us to make informed decisions. Consider a game of poker - understanding why players raise, call, and fold in various situations can provide a distinct advantage competitively.

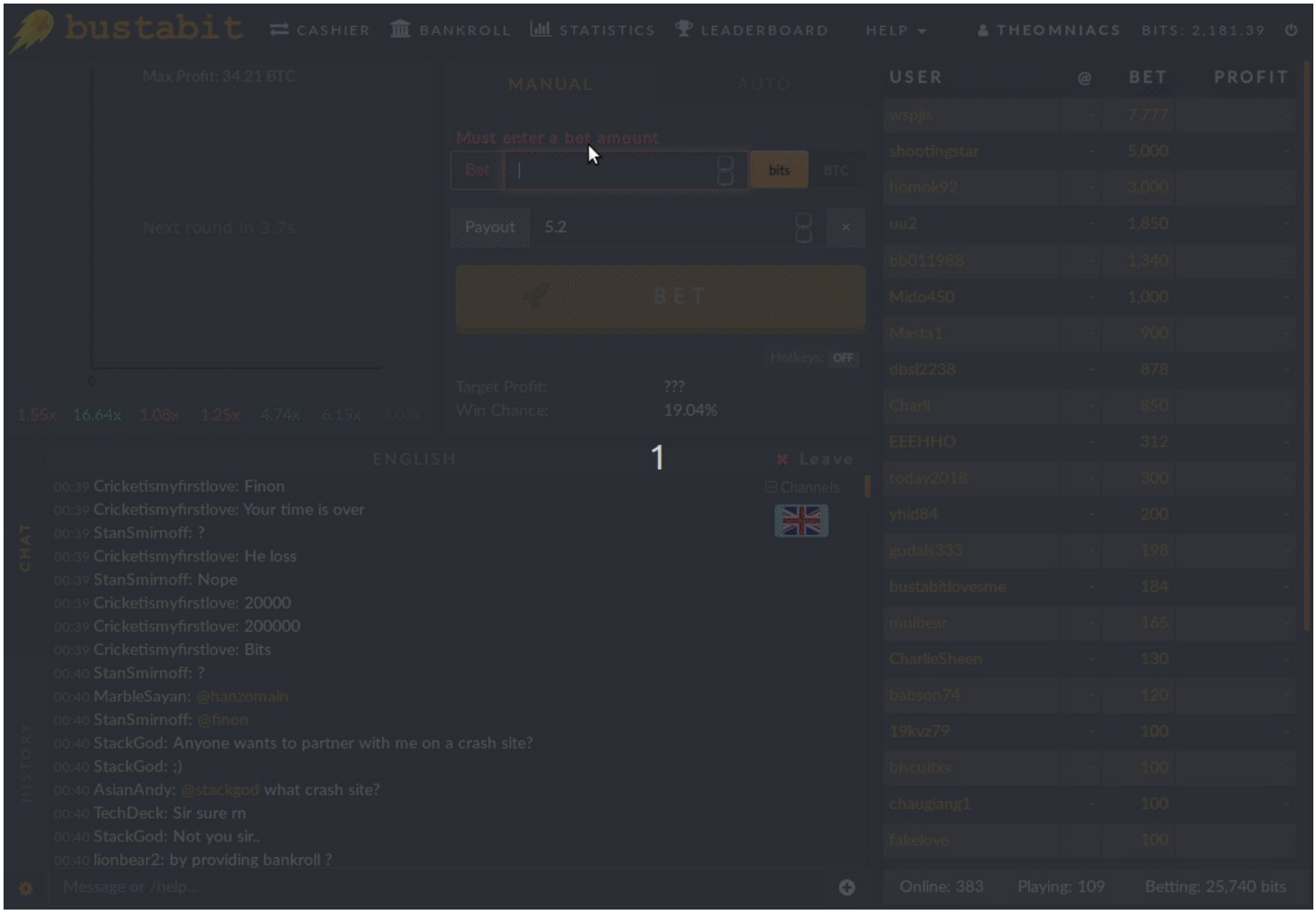
Along these lines, we are going to focus on the behavior on **online gamblers** from a platform called Bustabit (https://www.bustabit.com). There are a few basic rules for playing a game of Bustabit:

1. You bet a certain amount of money (in Bits, which is 1 / 1,000,000th of a Bitcoin) and you win if you cash out before the game **busts**.
2. Your win is calculated by the multiplier value at the moment you cashed out. For example, if you bet 100 and the value was 2.50x at the time you cashed out, you win 250. In addition, a percentage `Bonus` per game is multiplied with your bet and summed to give your final `Profit` in a winning game. Assuming a `Bonus` of 1%, your `Profit` for this round would be `(100 x 2.5) + (100 x .01) - 100 = 151`
3. The multiplier increases as time goes on, but if you wait too long to cash out, you may bust and lose your money.
4. Lastly, the house maintains slight advantages because in 1 out of every 100 games, everyone playing busts.

Below we see an example of a winning game:



And a losing game, for comparison:



Our goal will be to define relevant **groups** or **clusters** of Bustabit users to identify what patterns and behaviors of gambling persist. Can we describe a particular group as risk-averse? Is there a set of gamblers that have a strategy that seems to be more successful in the long term?

The data you will be working with includes over 40000 games of Bustabit by a bit over 4000 different players, for a total of 50000 rows (one game played by one player). The data includes the following variables:

1. **Id** - Unique identifier for a particular row (game result for one player)
2. **GameID** - Unique identifier for a particular game
3. **Username** - Unique identifier for a particular player
4. **Bet** - The number of Bits (1 / 1,000,000th of a Bitcoin) bet by the player in this game
5. **CashedOut** - The multiplier at which this particular player cashed out
6. **Bonus** - The bonus award (in percent) awarded to this player for the game
7. **Profit** - The amount this player won in the game, calculated as (Bet * CashedOut) + (Bet * Bonus) - Bet
8. **BustedAt** - The multiplier value at which this game busted
9. **PlayDate** - The date and time at which this game took place

Let's begin by doing an exploratory dive into the Bustabit data!

In [11]:

```
# Load the tidyverse
library(tidyverse)

# Read in the bustabit gambling data
bustabit <- read_csv("datasets/bustabit.csv")

# Look at the first five rows of the data
head(bustabit, n = 5)

# Find the highest multiplier (BustedAt value) achieved in a game
bustabit %>%
    arrange(desc(BustedAt)) %>%
    slice(1)
```

```
Parsed with column specification:
cols(
  Id = col_double(),
  GameID = col_double(),
  Username = col_character(),
  Bet = col_double(),
  CashedOut = col_double(),
  Bonus = col_double(),
  Profit = col_double(),
  BustedAt = col_double(),
  PlayDate = col_datetime(format = "")
)
```

A tibble: 5 x 9

Id	GameID	Username	Bet	CashedOut	Bonus	Profit	BustedAt	PlayDate
<dbl>	<dbl>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
14196549	3366002	papai	5	1.20	0.0	1.00	8.24	2016-11-20 19:44:19
10676217	3343882	znay22	3	NA	NA	NA	1.40	2016-11-14 14:21:50
15577107	3374646	rrrrrrrr	4	1.33	3.0	1.44	3.15	2016-11-23 06:39:15
25732127	3429241	sanya1206	10	NA	NA	NA	1.63	2016-12-08 18:13:55
17995432	3389174	ADM	50	1.50	1.4	25.70	2.29	2016-11-27 08:14:48

A spec_tbl_df: 1 x 9

Id	GameID	Username	Bet	CashedOut	Bonus	Profit	BustedAt	PlayDate
<dbl>	<dbl>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
19029273	3395044	Shadowshot	130	2	2.77	133.6	251025.1	2016-11-29 00:03:05

In [12]:

```
# These packages need to be loaded in the first @tests cell.
library(testthat)
library(IRkernel.testthat)

run_tests({
    test_that("packages are loaded", {
    expect_true("tidyverse" %in% .packages(), info = "Did you load the tidyverse package?")
    })

    test_that("bustabit is correct", {
    expect_is(bustabit, "tbl_df",
        info = "Did you read in the bustabit data with read_csv() (not read.csv())?")
    expect_equal(nrow(bustabit), 50000,
        info = "Did you read in the bustabit data with read_csv() (not read.csv())?")
    })
})
```

```
Attaching package: 'testthat'

The following object is masked from 'package:dplyr':

    matches

The following object is masked from 'package:purrr':

    is_null

The following object is masked from 'package:tidyr':

    matches


2/2 tests passed
```

2. Deriving relevant features for clustering

The Bustabit data provides us with many features to work with, but to better quantify player behavior, we need to derive some more variables. Currently, we have a `Profit` column which tells us the amount won in that game, but no indication of how much was lost if the player busted, and no indicator variable quantifying whether the game itself was a win or loss overall. Hence, we will derive or modify the following variables:

1. **CashedOut** - If the value for `CashedOut` is `NA`, we will set it to be 0.01 greater than the `BustedAt` value to signify that the user failed to cash out before busting
2. **Profit** - If the value for `Profit` is `NA`, we will set it to be zero to indicate no profit for the player in that game
3. **Losses** - If the new value for `Profit` is zero, we will set this to be the amount the player lost in that game, otherwise we will set it to zero. This value should always be *zero or negative*
4. **GameWon** - If the user made a profit in this game, the value should be 1, and 0 otherwise
5. **GameLost** If the user had a loss in this game, the value should be 1, and 0 otherwise

In [13]:

```
# Create the new feature variables
bustabit_features <- bustabit %>%
  mutate(CashedOut = ifelse(is.na(CashedOut), BustedAt + .01, CashedOut),
         Profit = ifelse(is.na(Profit), 0, Profit),
         Losses = ifelse(Profit == 0, -1 * Bet, 0),
         GameWon = ifelse(Profit == 0, 0, 1),
         GameLost = ifelse(Profit == 0, 1, 0))

# Look at the first five rows of the data
head(bustabit_features, n = 5)
```

A tibble: 5 x 12

Id	GameID	Username	Bet	CashedOut	Bonus	Profit	BustedAt	PlayDate	Losses	GameWon	GameLost
<dbl>	<dbl>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>	<dbl>	<dbl>	<dbl>
14196549	3366002	papai	5	1.20	0.0	1.00	8.24	2016-11-20 19:44:19	0	1	0
10676217	3343882	znay22	3	1.41	NA	0.00	1.40	2016-11-14 14:21:50	-3	0	1
15577107	3374646	rrrrrrrr	4	1.33	3.0	1.44	3.15	2016-11-23 06:39:15	0	1	0
25732127	3429241	sanya1206	10	1.64	NA	0.00	1.63	2016-12-08 18:13:55	-10	0	1
17995432	3389174	ADM	50	1.50	1.4	25.70	2.29	2016-11-27 08:14:48	0	1	0

In [14]:

```
bustabit_features_soln <- bustabit %>%
  mutate(CashedOut = ifelse(is.na(CashedOut), BustedAt + .01, CashedOut),
         Profit = ifelse(is.na(Profit), 0, Profit),
         Losses = ifelse(Profit == 0, -1 * Bet, 0),
         GameWon = ifelse(Profit == 0, 0, 1),
         GameLost = ifelse(Profit == 0, 1, 0))

run_tests({
    test_that("bustabit_features is correct", {
    expect_is(bustabit_features, "tbl_df",
        info = "Did you use tidyverse routines to create bustabit_features?")
    expect_identical(bustabit_features$Losses, bustabit_features_soln$Losses,
        info = "Did you compute the Losses column correctly by taking the -1 * Bet (for a lost game) or 0 (for a
winning game)?")
    expect_identical(bustabit_features$GameWon, bustabit_features_soln$GameWon,
        info = "Did you compute the GameLost column by using the value 0 for a lost game, and 1 for a winning gam
e?")
    expect_identical(bustabit_features$GameLost, bustabit_features_soln$GameLost,
        info = "Did you compute the GameLost column by using the value 1 for a lost game, and 0 for a winning gam
e?")
    })
})
```

```
1/1 tests passed
```

3. Creating per-player statistics

The primary task at hand is to cluster Bustabit **players** by their respective gambling habits. Right now, however, we have features at the per-game level. The features we've derived would be great if we were interested in clustering properties of the games themselves - we know things about the BustedAt multiplier, the time the game took place, and lots more. But to better quantify player behavior, we must group the data by player (`Username`) to begin thinking about the relationship and similarity between groups of players. Some per-player features we will create are:

1. **AverageCashedOut** - The average multiplier at which the player cashes out
2. **AverageBet** - The average bet made by the player
3. **TotalProfit** - The total profits over time for the player
4. **TotalLosses** - The total losses over time for the player
5. **GamesWon** - The total number of individual games the player won
6. **GamesLost** - The total number of individual games the player lost

With these variables, we will be able to potentially group similar users based on their typical Bustabit gambling behavior.

In [15]:

```
# Group by players to create per-player summary statistics
bustabit_clus <- bustabit_features %>%
  group_by(Username) %>%
  summarize(AverageCashedOut = mean(CashedOut),
            AverageBet = mean(Bet),
            TotalProfit = sum(Profit),
            TotalLosses = sum(Losses),
            GamesWon = sum(GameWon),
            GamesLost = sum(GameLost))

# View the first five rows of the data
head(bustabit_clus, n = 5)
```

```
`summarise()` ungrouping output (override with `.groups` argument)
```

A tibble: 5 x 7

Username	AverageCashedOut	AverageBet	TotalProfit	TotalLosses	GamesWon	GamesLost
<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
----------	1.040000	10.33333	0.70	-8	2	1
--dilib--	1.505000	210.75000	371.37	-1239	2	6
-31337-	1.220000	32.50000	21.50	-55	3	1
-Nothing-	1.417077	322.87692	3794.31	-6915	42	23
-Tachyon	1.246000	1024.80000	629.68	-2124	4	1

In [16]:

```
bustabit_clus_soln <- bustabit_features_soln %>%
  group_by(Username) %>%
  summarize(AverageCashedOut = mean(CashedOut),
            AverageBet = mean(Bet),
            TotalProfit = sum(Profit),
            TotalLosses = sum(Losses),
            GamesWon = sum(GameWon),
            GamesLost = sum(GameLost))

run_tests({
    test_that("bustabit_clus is correct", {
    expect_is(bustabit, "tbl_df",
        info = "Did you use tidyverse routines to create bustabit_clus?")
    expect_equal(nrow(bustabit_clus), 4149,
        info = "bustabit_clus does not have the correct number of rows. Please make sure you grouped the data by
Username.")
    expect_identical(bustabit_clus$AverageBet, bustabit_clus_soln$AverageBet,
        info = "Did you compute the AverageBet column correctly by averaging all past bets?")
    expect_identical(bustabit_clus$TotalLosses, bustabit_clus_soln$TotalLosses,
        info = "Did you compute the TotalLosses column correctly by summing all past losses?")
    expect_identical(bustabit_clus$GamesLost, bustabit_clus_soln$GamesLost,
        info = "Did you compute the GamesLost column by counting all past losing games?")
    })
})
```

```
`summarise()` ungrouping output (override with `.groups` argument)
```

```
1/1 tests passed
```

4. Scaling and normalization of the derived features

The variables are on very different **scales** right now. For example, `AverageBet` is in bits (1/1000000 of a Bitcoin), `AverageCashedOut` is a multiplier, and `GamesLost` and `GamesWon` are counts. As a result, we would like to **normalize** the variables such that across clustering algorithms, they will have approximately equal weighting.

One thing to think about is that in many cases, we may actually want a particular numeric variable to maintain a higher weight. This could occur if there is some prior knowledge regarding, for example, which variable might be most important in terms of defining similar Bustabit behavior. In this case, without that prior knowledge, we will forego the weighting of variables and scale everything. We are going to use **mean-sd** standardization to scale the data. Note that this is also known as a **Z-score**.

Note that we could compute the Z-scores by using the base R function `scale()`, but we're going to write our own function in order to get the practice.

In [17]:

```
# Create the mean-sd standardization function
mean_sd_standard <- function(x) {
    (x - mean(x)) / sd(x)
}

# Apply this function to each numeric variable in the clustering set
bustabit_standardized <- bustabit_clus %>%
    mutate_if(is.numeric, mean_sd_standard)

# Summarize our standardized data
summary(bustabit_standardized)
```

```
   Username          AverageCashedOut     AverageBet       TotalProfit
 Length:4149        Min.   :-0.76289   Min.   :-0.1773   Min.   :-0.09052
 Class :character   1st Qu.:-0.28157   1st Qu.:-0.1765   1st Qu.:-0.09050
 Mode  :character   Median :-0.18056   Median :-0.1711   Median :-0.08974
                    Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.00000
                    3rd Qu.: 0.02752   3rd Qu.:-0.1384   3rd Qu.:-0.08183
                    Max.   :41.72651   Max.   :24.9971   Max.   :40.73652
  TotalLosses          GamesWon          GamesLost
 Min.   :-41.84541   Min.   :-0.4320   Min.   :-0.41356
 1st Qu.:  0.09837   1st Qu.:-0.3696   1st Qu.:-0.41356
 Median :  0.10847   Median :-0.3071   Median :-0.33306
 Mean   :  0.00000   Mean   : 0.0000   Mean   : 0.00000
 3rd Qu.:  0.10916   3rd Qu.:-0.1196   3rd Qu.:-0.09156
 Max.   :  0.10916   Max.   :13.2534   Max.   :19.30911
```

In [18]:

```
mean_sd_standard_soln <- function(x) {
    (x - mean(x)) / sd(x)
}

bustabit_standardized_soln <- bustabit_clus_soln %>%
    mutate_if(funs(is.numeric), mean_sd_standard_soln)

run_tests({
    test_that("mean_sd_standard is correct", {
        expect_identical(mean_sd_standard(1:10), mean_sd_standard_soln(1:10),
            info = "Did you perform mean-sd standardization by subtracting the mean and dividing by the standard
deviation?")
    })

    test_that("bustabit_standardized is correct", {
    expect_is(bustabit_standardized, "tbl_df",
        info = "Did you use dplyr routines to apply this function to each numeric column?")
    expect_equal(bustabit_standardized$AverageCashedOut, mean_sd_standard_soln(bustabit_clus$AverageCashedOut),
        info = "Is your mean_sd_standard function correctly applied to the data columns?")
    })
})
```

```
Warning message:
"`funs()` is deprecated as of dplyr 0.8.0.
Please use a list of either functions or lambdas:

  # Simple named list:
  list(mean = mean, median = median)

  # Auto named with `tibble::lst()`:
  tibble::lst(mean, median)

  # Using lambdas
  list(~ mean(., trim = .2), ~ median(., na.rm = TRUE))
This warning is displayed once every 8 hours.
Call `lifecycle::last_warnings()` to see where this warning was generated."

2/2 tests passed
```

5. Cluster the player data using K means

With standardized data of per-player features, we are now ready to use K means clustering in order to cluster the players based on their online gambling behavior. K means is implemented in R in the `kmeans()` function from the stats package. This function requires the `centers` parameter, which represents the number of clusters to use.

Without prior knowledge, it is often difficult to know what an appropriate choice for the number of clusters is. We will begin by choosing **five**. This choice is rather arbitrary, but represents a good initial compromise between choosing too many clusters (which reduces the interpretability of the final results), and choosing too few clusters (which may not capture the distinctive behaviors effectively). Feel free to play around with other choices for the number of clusters and see what you get instead!

One subtlety to note - because the K means algorithm uses a random start, we are going to set a random seed first in order to ensure the results are reproducible.

In [19]:

```
# Choose 20190101 as our random seed
set.seed(20190101)

# Cluster the players using kmeans with five clusters
cluster_solution <- kmeans(bustabit_standardized[,-1], centers = 5)

# Store the cluster assignments back into the clustering data frame object
bustabit_clus$cluster <- factor(cluster_solution$cluster)

# Look at the distribution of cluster assignments
table(bustabit_clus$cluster)
```

```
   1    2    3    4    5
 412   16   17 3626   78
```

In [20]:

```
bustabit_clus_soln$cluster <- factor(cluster_solution$cluster)

run_tests({
    test_that("The cluster assignment was performed correctly", {
        expect_is(cluster_solution, "kmeans",
                  info = "Did you use the kmeans() function to create your cluster assignment?")
        expect_equal(length(unique(cluster_solution$cluster)), 5,
            info = "Did you choose 5 clusters?")
    })


    test_that("The assignments were stored in the cluster data frame", {
    expect_is(bustabit_clus$cluster, "factor",
        info = "Did you store the cluster assignment in the bustabit_clus object and explicitly coerce the object
to be a factor?")
    })
})
```

```
2/2 tests passed
```

6. Compute averages for each cluster

We have a clustering assignment which maps every Bustabit gambler to one of five different groups. To begin to assess the quality and distinctiveness of these groups, we are going to look at **group averages** for each cluster across the original variables in our clustering dataset. This will, for example, allow us to see which cluster tends to make the largest bets, which cluster tends to win the most games, and which cluster tends to lose the most money. This will provide us with our first clear indication as to whether the behaviors of the groups appear distinctive!

In [21]:

```
# Group by the cluster assignment and calculate averages
bustabit_clus_avg <- bustabit_clus %>%
    group_by(cluster) %>%
    summarize_if(is.numeric, mean)

# View the resulting table
bustabit_clus_avg
```

A tibble: 5 x 7

cluster	AverageCashedOut	AverageBet	TotalProfit	TotalLosses	GamesWon	GamesLost
<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	1.915776	1633.2292	19362.9909	-19205.1165	27.0606796	21.036408
2	2.470024	298945.6618	1198191.1631	-1056062.1875	10.5625000	8.062500
3	27.448235	1278.2574	619.4041	-581.2941	0.7058824	1.529412
4	1.699993	4024.1102	4272.6656	-4365.7788	2.9109211	2.128792
5	1.758407	432.1163	18568.1141	-16724.0641	87.1794872	61.205128

In [22]:

```
cluster_mean_solution <- bustabit_clus_soln %>%
    group_by(cluster) %>%
    summarize_if(funs(is.numeric), mean)

run_tests({
    test_that("The bustabit_clus_avg was created correctly", {
        expect_is(bustabit_clus_avg, "tbl_df",
                  info = "Did you use tidyverse routines to compute the cluster averages?")
        expect_identical(bustabit_clus_avg, cluster_mean_solution,
            info = "Did you correctly compute the group averages?")
    })
})
```

```
1/1 tests passed
```

7. Visualize the clusters with a Parallel Coordinate Plot

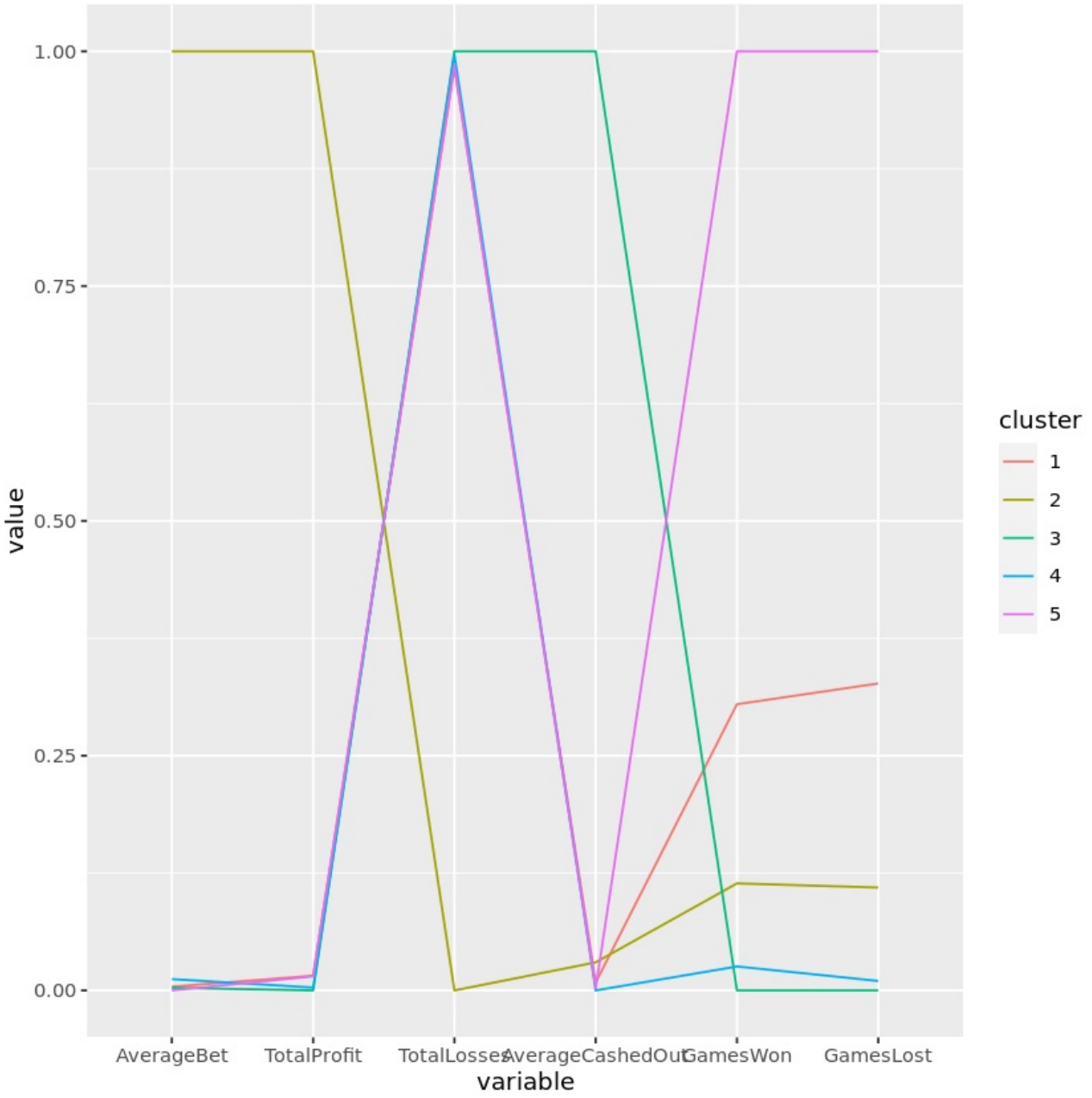
We can already learn a bit about our cluster groupings by looking at the previous table. We can clearly see that there is a group that makes very large bets, a group that tends to cash out at very high multiplier values, and a group that has played many games of Bustabit. We can visualize these group differences graphically using a Parallel Coordinate Plot or PCP. To do so, we will introduce one more kind of scaling: min-max scaling, which forces each variable to fall between 0 and 1.

Other choices of scaling, such as the Z-score method from before, can work effectively as well. However, min-max scaling has the advantage of **interpretability** - a value of 1 for a particular variable indicates that cluster has the highest value compared to all other clusters, and a value of 0 indicates that it has the lowest. This can help make relative comparisons between the clusters more clear.

The `ggparcoord()` function from `GGally` will be used to produce a Parallel Coordinate Plot. Note that this has a built-in argument `scale` to perform different scaling options, including min-max scaling. We will set this option to `"globalminmax"` to perform no scaling, and write our own scaling routine for practice. If you are interested, you can look at the function definition for `ggparcoord()` to help you write our scaling function!

In [23]:

```
# Create the min-max scaling function
min_max_standard <- function(x) {
    (x - min(x)) / (max(x) - min(x))
}

# Apply this function to each numeric variable in the bustabit_clus_avg object
bustabit_avg_minmax <- bustabit_clus_avg %>%
    mutate_if(is.numeric, min_max_standard)

# Load the GGally package
library(GGally)

# Create a parallel coordinate plot of the values
ggparcoord(bustabit_avg_minmax, columns = 2:ncol(bustabit_avg_minmax),
           groupColumn = "cluster", scale = "globalminmax", order = "skewness")
```


1.00
0.75
0.50
0.25
0.00
value
AverageBet
TotalProfit
TotalLosses
AverageCashedOut
GamesWon
GamesLost
variable
cluster
1
2
3
4
5

In [24]:

```
min_max_standard_soln <- function(x) {
    (x - min(x)) / (max(x) - min(x))
}

bustabit_avg_minmax_soln <- cluster_mean_solution %>%
    mutate_if(is.numeric, funs(min_max_standard_soln))

student_plot <- last_plot()
solution_plot <- ggparcoord(bustabit_avg_minmax, columns = 2:ncol(bustabit_avg_minmax),
           groupColumn = "cluster", scale = "globalminmax", order = "skewness")

run_tests({
    test_that("min_max_standard is correct", {
        expect_identical(min_max_standard(1:10), min_max_standard_soln(1:10),
            info = "Did you perform min-max standardization by subtracting the min and dividing by the range?")
    })

    test_that("bustabit_avg_minmax is correct", {
        expect_is(bustabit_avg_minmax, "tbl_df",
            info = "Did you use dplyr routines to apply bustabit_avg_minmax to each numeric column?")
        expect_identical(bustabit_avg_minmax, bustabit_avg_minmax_soln,
            info = "Did you correctly apply min_max_standard to each numeric column in bustabit_avg_minmax?")
    })

    test_that("The plot is drawn correctly", {
        expect_s3_class(student_plot, "ggplot")
        expect_identical(
            student_plot$data,
            solution_plot$data,
            info = 'The plot data is incorrect. Did you use `bustabit_avg_minmax`?'
        )
        expect_identical(
            deparse(student_plot$mapping$x),
            deparse(solution_plot$mapping$x),
            info = 'The `x` aesthetic is incorrect. Did you use `ggparcoord` to produce the plot?'
        )
        expect_identical(
            deparse(student_plot$mapping$y),
            deparse(solution_plot$mapping$y),
            info = 'The `y` aesthetic is incorrect. Did you use `ggparcoord` to produce the plot?'
        )
        expect_identical(
            deparse(student_plot$mapping$group),
            deparse(solution_plot$mapping$group),
            info = 'The `group` aesthetic is incorrect. Did you specify the `groupColumn` argument for ggparcoord
as cluster?'
        )
    })
})
```

```
3/3 tests passed
```

8. Visualize the clusters with Principal Components

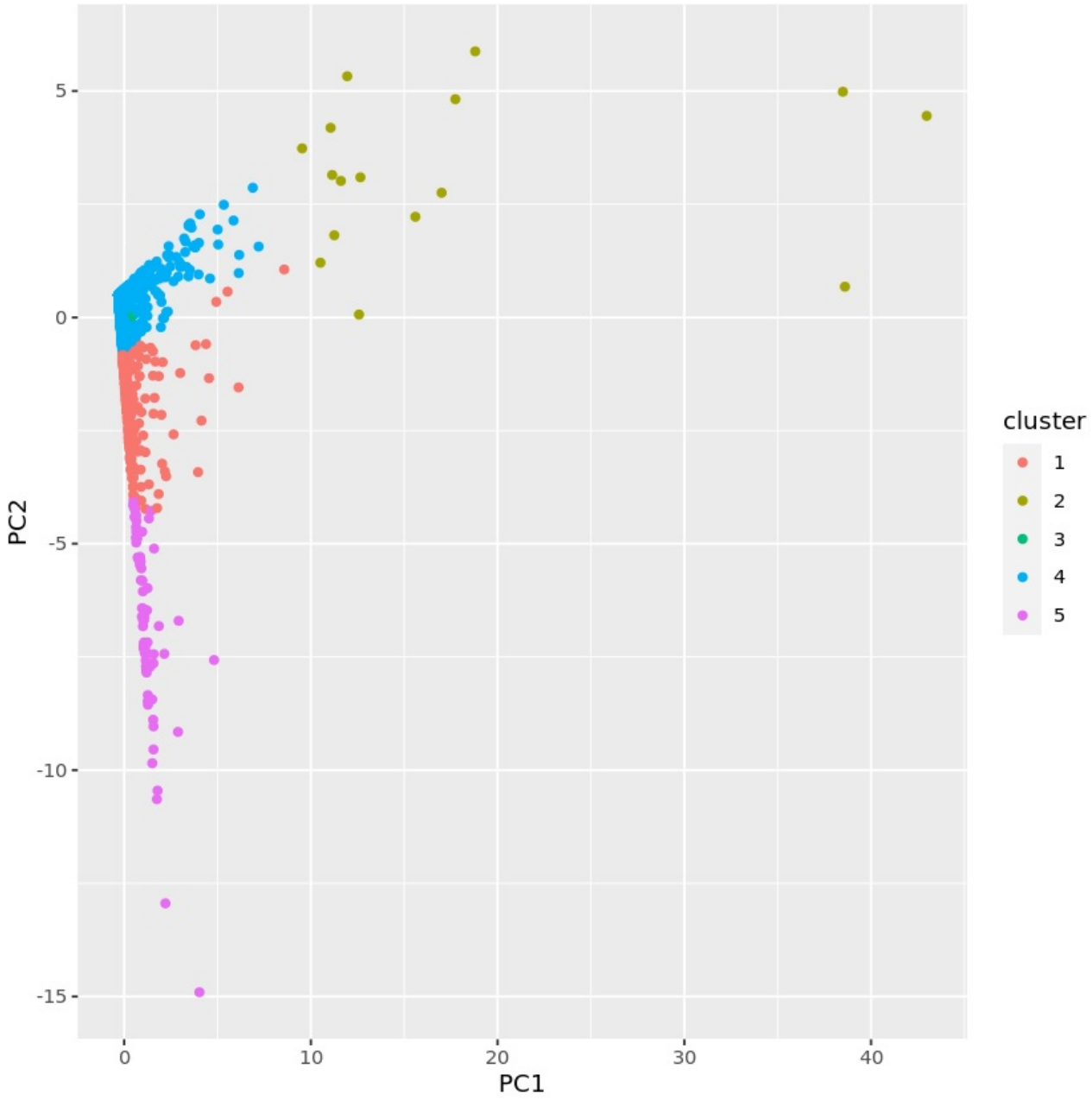
One issue with plots like the previous is that they get more unwieldy as we continue to add variables. One way to solve this is to use the Principal Components of a dataset in order to reduce the dimensionality to aid in visualization. Essentially, this is a two-stage process:

1. We extract the principal components in order to reduce the dimensionality of the dataset so that we can produce a scatterplot in two dimensions that captures the underlying structure of the higher-dimensional data.
2. We then produce a scatterplot of each observation (in this case, each player) across the two Principal Components and color according to their cluster assignment in order to visualize the separation of the clusters.

This plot provides interesting information in terms of the similarity of any two players. In fact, you will see that players who fall close to the boundaries of clusters might be the ones that exhibit the gambling behavior of a couple of different clusters. After you produce your plot, try to determine which clusters seem to be the most "different." Also, try playing around with different projections of the data, such as PC3 vs. PC2, or PC3 vs. PC1, to see if you can find one that better differentiates the groups.

In [25]:

```
# Extract the first two principle components from the data
my_pc <- as.data.frame(prcomp(bustabit_standardized[,-1])$x)

# Store the cluster assignment in this object
my_pc$cluster <- bustabit_clus$cluster

# Use ggplot to plot PC2 vs PC1, and color by the cluster assignment
p1 <- ggplot(data = my_pc, aes(x = PC1, y = PC2, color = cluster)) +
    geom_point()

# View the resulting plot
p1
```



In [26]:

```
my_pc_soln <- as.data.frame(prcomp(bustabit_standardized_soln[,-1])$x)
my_pc_soln$cluster <- bustabit_clus_soln$cluster

student_plot <- last_plot()
solution_plot <- ggplot(data = my_pc, aes(x = PC1, y = PC2, color = cluster)) +
    geom_point()

run_tests({
    test_that("min_max_standard is correct", {
        expect_is(p1, "ggplot",
            info = "Did you use ggplot to create the PCP and store it as p1?")
    })

    test_that("The plot is drawn correctly", {
        expect_s3_class(student_plot, "ggplot")
        expect_identical(
            student_plot$data,
            solution_plot$data,
            info = 'The plot data is incorrect. Did you use `my_pc` as the data argument?'
        )
        expect_identical(
            deparse(student_plot$mapping$x),
            deparse(solution_plot$mapping$x),
            info = 'The `x` variable is incorrect. Did you assign PC1 as the x variable?'
        )
        expect_identical(
            deparse(student_plot$mapping$y),
            deparse(solution_plot$mapping$y),
            info = 'The `y` variable is incorrect. Did you assign PC2 as the y variable?'
        )
        expect_identical(
            deparse(student_plot$mapping$color),
            deparse(solution_plot$mapping$color),
            info = 'The `color` variable is incorrect. Did you specify cluster as the color variable?'
        )
    })
})
```

```
2/2 tests passed
```

9. Analyzing the groups of gamblers our solution uncovered

Though most of the statistical and programmatical work has been completed, possibly the most important part of a cluster analysis is to interpret the resulting clusters. This often is the most desired aspect of the analysis by clients, who are hoping to use the results of your analysis to inform better business decision making and actionable items. As a final step, we'll use the parallel coordinate plot and cluster means table to interpret the Bustabit gambling user groups! Roughly speaking, we can breakdown the groups as follows:

Cautious Commoners:
This is the largest of the five clusters, and might be described as the more casual Bustabit players. They've played the fewest number of games overall, and tend to make more conservative bets in general.

Strategic Addicts:
These users play a lot of games on Bustabit, but tend to keep their bets under control. As a result, they've made on average a net positive earnings from the site, in spite of having the most games played. They seem to maintain a strategy (or an automated script/bot) that works to earn them money.

Risky Commoners:
These users seem to be a step above the Cautious Commoners in their Bustabit gambling habits, making larger average bets, and playing a larger number of games on the site. As a result, though they have about the same number of average games won as the Risk Takers, they have a significantly higher number of games lost.

Risk Takers:
These users have played only a couple games on average, but their average cashed out value is significantly higher than the other clusters, indicating that they tend to wait for the multiplier to increase to large values before cashing out.

High Rollers:
High bets are the name of the game for this group. They bet large sums of money in each game, although they tend to cash out at lower multipliers and thus play the game more conservatively, particularly compared to the Risk Takers. Interestingly, these users have also on average earned a net positive earnings from their games played.

In [27]:

```
# Assign names to clusters 1 through 5 in order
cluster_names <- c(
    "Risky Commoners",
    "High Rollers",
    "Risk Takers",
    "Cautious Commoners",
    "Strategic Addicts"
)

# Append the cluster names to the cluster means table
bustabit_clus_avg_named <- bustabit_clus_avg %>%
    cbind(Name = cluster_names)

# View the cluster means table with your appended cluster names
bustabit_clus_avg_named
```

A data.frame: 5 x 8

cluster	AverageCashedOut	AverageBet	TotalProfit	TotalLosses	GamesWon	GamesLost	Name
<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<fct>
1	1.915776	1633.2292	19362.9909	-19205.1165	27.0606796	21.036408	Risky Commoners
2	2.470024	298945.6618	1198191.1631	-1056062.1875	10.5625000	8.062500	High Rollers
3	27.448235	1278.2574	619.4041	-581.2941	0.7058824	1.529412	Risk Takers
4	1.699993	4024.1102	4272.6656	-4365.7788	2.9109211	2.128792	Cautious Commoners
5	1.758407	432.1163	18568.1141	-16724.0641	87.1794872	61.205128	Strategic Addicts

In [28]:

```
cluster_names_soln <- c(
    "Risky Commoners",
    "High Rollers",
    "Risk Takers",
    "Cautious Commoners",
    "Strategic Addicts"
)

bustabit_clus_avg_named_soln <- cluster_mean_solution %>%
    cbind(Name = cluster_names_soln)

run_tests({
    test_that("The cluster labels are correctly defined", {
        expect_identical(
            cluster_names,
            cluster_names_soln,
            info = 'Did you correctly identify clusters 1 through 5 with the appropriate name?'
        )
    })

    test_that("The labels were stored in the cluster means table correctly", {
        expect_identical(
            bustabit_clus_avg_named,
            bustabit_clus_avg_named_soln,
            info = 'Did you store the correct cluster names as a column in the cluster means table?'
        )
    })
})
```

```
2/2 tests passed
```

Name – Mohammed Safwan Aslam Kazi
NYIT ID – 1270795
Course – Data Mining
Project Title – Disney Movies and Box Office Success
Source – https://data.world/kgarrett/disney-character-success-00-16/disney-characters.csv

1. Exploration of the data set of Disney Movies

Walt Disney Studios is the foundation on which The Walt Disney Company was built. The Studios has produced more than 600 films since their debut film, Snow White and the Seven Dwarfs in 1937. While many of its films were big hits, some of them were not. In this notebook, we will explore a dataset of Disney movies and analyze what contributes to the success of Disney movies.
First, we will take a look at the Disney Data Set. The data contains 579 Disney movies with six attributes/features: movie title, release date, genre, MPAA rating, total gross, and inflation adjusted gross.

Let's load the data set (**source: https://data.world/kgarrett/disney-character-success-00-16/disney-characters.csv**) and explore the given data set: -

In [2]:
```
# Import pandas library as pd
import pandas as pd

# Read the csv file and store as a data frame into df_gross variable
df_gross = pd.read_csv("D:/NYIT (Fall2020-ACADEMICS)/INTRODUCTION TO DATA MINI
NG (CSCI657 or CSCI415) [PROFF Helen Gu]/PROJECTS/Disney Movies and Box Office
Success/datasets/disney_movies_total_gross.csv")

# Print the output of df_gross variable
df_gross
```

Out[2]:

	movie_title	release_date	genre	mpaa_rating	total_gross	inflation_adjusted_gross
0	Snow White and the Seven Dwarfs	1937-12-21	Musical	G	184925485	5228953251
1	Pinocchio	1940-02-09	Adventure	G	84300000	2188229052
2	Fantasia	1940-11-13	Musical	G	83320000	2187090808
3	Song of the South	1946-11-12	Adventure	G	65000000	1078510579
4	Cinderella	1950-02-15	Drama	G	85000000	920608730
...	...	...	...	...	...	...
574	The Light Between Oceans	2016-09-02	Drama	PG-13	12545979	12545979
575	Queen of Katwe	2016-09-23	Drama	PG	8874389	8874389
576	Doctor Strange	2016-11-04	Adventure	PG-13	232532923	232532923
577	Moana	2016-11-23	Adventure	PG	246082029	246082029
578	Rogue One: A Star Wars Story	2016-12-16	Adventure	PG-13	529483936	529483936

579 rows × 6 columns

In [37]:
```
# Calculates the length of records of DataFrame "df_gross"
print("Total number of movies present in data set:\n", len(df_gross))
```

```
Total number of movies present in data set:
 579
```

In [38]:
```
# Checks whether the DataFrame "df_gross" has any N/A values
df_gross.isna().sum()
```

Out[38]:
```
movie_title                 0
release_date                0
genre                      17
mpaa_rating                56
total_gross                 0
inflation_adjusted_gross    0
release_year                0
dtype: int64
```

In [33]:
```
# Concise summary of DataFrame "df_gross"
df_gross.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 579 entries, 0 to 578
Data columns (total 7 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   movie_title               579 non-null    object
 1   release_date              579 non-null    object
 2   genre                     562 non-null    object
 3   mpaa_rating               523 non-null    object
 4   total_gross               579 non-null    int64
 5   inflation_adjusted_gross  579 non-null    int64
 6   release_year              579 non-null    int64
dtypes: int64(3), object(4)
memory usage: 31.8+ KB
```

In [3]:
```
# Basic statistical details
df_gross.describe()
```

Out[3]:

	total_gross	inflation_adjusted_gross
count	5.790000e+02	5.790000e+02
mean	6.470179e+07	1.187625e+08
std	9.301301e+07	2.860853e+08
min	0.000000e+00	0.000000e+00
25%	1.278886e+07	2.274123e+07
50%	3.070245e+07	5.515978e+07
75%	7.570903e+07	1.192020e+08
max	9.366622e+08	5.228953e+09

2. Top ten movies at the box office

Let's start by exploring the data by ordering them in descending fashion as per the box office records (i.e., which movie has earned the most popularity at the box office). Here, the sorting of movies is done over their "inflation adjusted gross".

In [32]:
```
# Sort data by the adjusted gross attribute in descending order
inflation_ajusted_gross_dec = df_gross.sort_values(by="inflation_adjusted_gross", ascending=False)

# Display the top 10 movies
inflation_ajusted_gross_dec.head(10)
```

Out[32]:

	movie_title	release_date	genre	mpaa_rating	total_gross	inflation_adjusted_gross	rele
0	Snow White and the Seven Dwarfs	1937-12-21	Musical	G	184925485	5228953251	
1	Pinocchio	1940-02-09	Adventure	G	84300000	2188229052	
2	Fantasia	1940-11-13	Musical	G	83320000	2187090808	
8	101 Dalmatians	1961-01-25	Comedy	G	153000000	1362870985	
6	Lady and the Tramp	1955-06-22	Drama	G	93600000	1236035515	
3	Song of the South	1946-11-12	Adventure	G	65000000	1078510579	
564	Star Wars Ep. VII: The Force Awakens	2015-12-18	Adventure	PG-13	936662225	936662225	
4	Cinderella	1950-02-15	Drama	G	85000000	920608730	
13	The Jungle Book	1967-10-18	Musical	Not Rated	141843000	789612346	
179	The Lion King	1994-06-15	Adventure	G	422780140	761640898	

3. Movie genre trend

From the top 10 movies above, it seems that some genres are more popular than others. So, we will check which genres are growing stronger in popularity. To do this, we will group movies by genre and then by year to see the adjusted gross of each genre in each year.

In [22]:
```
# Extract year from release_date and store it in a new column
df_gross["release_year"] =pd.DatetimeIndex(df_gross["release_date"]).year

# Print the output of variable df_gross
df_gross
```

Out[22]:

	movie_title	release_date	genre	mpaa_rating	total_gross	inflation_adjusted_gross	rele
0	Snow White and the Seven Dwarfs	1937-12-21	Musical	G	184925485	5228953251	
1	Pinocchio	1940-02-09	Adventure	G	84300000	2188229052	
2	Fantasia	1940-11-13	Musical	G	83320000	2187090808	
3	Song of the South	1946-11-12	Adventure	G	65000000	1078510579	
4	Cinderella	1950-02-15	Drama	G	85000000	920608730	
...	...	...	...	...	...	...	
574	The Light Between Oceans	2016-09-02	Drama	PG-13	12545979	12545979	
575	Queen of Katwe	2016-09-23	Drama	PG	8874389	8874389	
576	Doctor Strange	2016-11-04	Adventure	PG-13	232532923	232532923	
577	Moana	2016-11-23	Adventure	PG	246082029	246082029	
578	Rogue One: A Star Wars Story	2016-12-16	Adventure	PG-13	529483936	529483936	

579 rows × 7 columns

In [23]:
```
# Compute mean of adjusted gross per genre and per year
df_group = df_gross.groupby(["genre", "release_year"]).mean()

# Conver the GroupBy object to a DataFrame
genre_yearly = df_group.reset_index()

# Inspect genre_yearly
genre_yearly.head(10)
```

Out[23]:

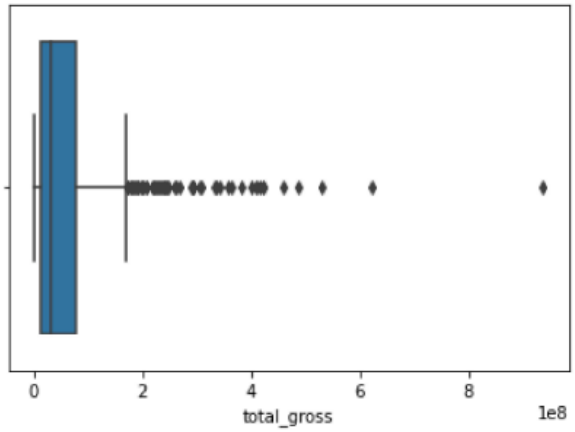
	genre	release_year	total_gross	inflation_adjusted_gross
0	Action	1981	0,0	0,0
1	Action	1982	26918576,0	77184895,0
2	Action	1988	17577696.0	36053517.0
3	Action	1990	59249588.5	118358772.0
4	Action	1991	28924936.5	57918572.5
5	Action	1992	29028000,0	58965304,0
6	Action	1993	21943553,5	44682157,0
7	Action	1994	19180582,0	39545796,0
8	Action	1995	63037553.5	122162426.5
9	Action	1996	135281096.0	257755262.5

4. Visualize the genre popularity trend

We will make some plots out of these means of groups to consider the outliers using box-plots, dist-plots (histograms), and line-plots to better see how the box office revenues have changed over time.

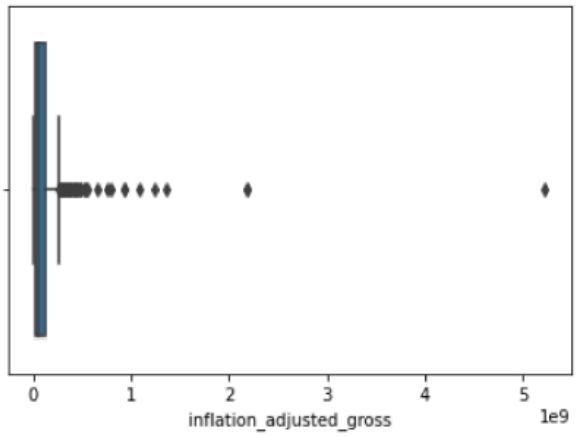
In [16]:
```
# Import seaborn library as sb
import seaborn as sb

# Plot the box-plot for attribute total_gross to depict groups through the qua
rtiles
# and represent the outliers in data set
sb.boxplot(x = df_gross["total_gross"])
```

Out[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x216ffdb4b50>
```



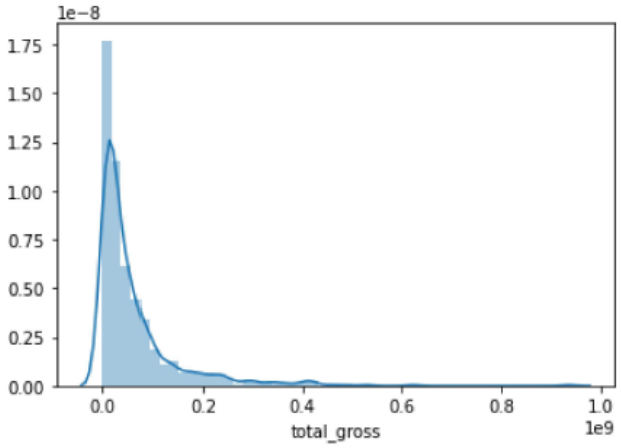
In [17]:
```
# Import seaborn library as sb
import seaborn as sb

# Plot the box-plot for attribute inflation_adjusted_gross to depict groups th
rough the quartiles
# and represent the outliers in data set
sb.boxplot(x = df_gross["inflation_adjusted_gross"])
```

Out[17]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x216fffbbf10>
```



In [18]:
```
# Import seaborn library as sb
import seaborn as sb

# Plot the dist-plot for attribute total_gross to depict histogram
sb.distplot(df_gross["total_gross"])
```

Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x216805847c0>



In [24]:
```
df_gross
```

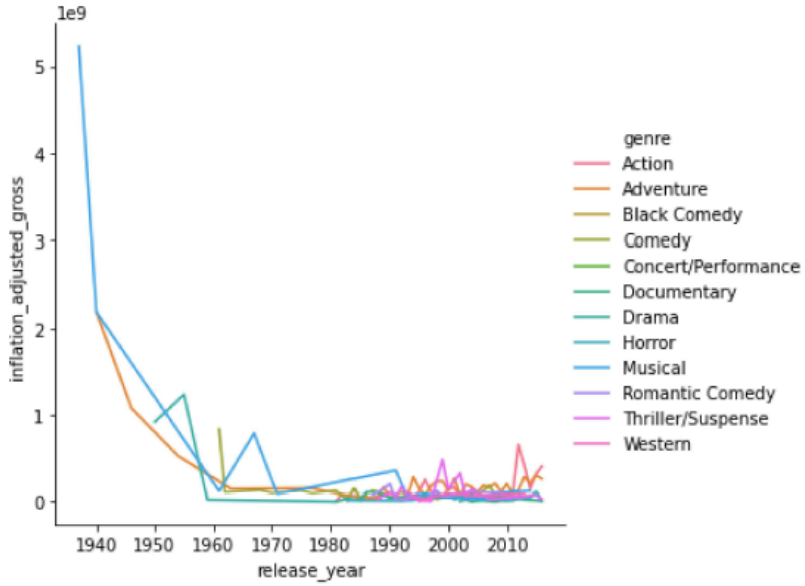
Out[24]:

	movie_title	release_date	genre	mpaa_rating	total_gross	inflation_adjusted_gross	rele
0	Snow White and the Seven Dwarfs	1937-12-21	Musical	G	184925485	5228953251	
1	Pinocchio	1940-02-09	Adventure	G	84300000	2188229052	
2	Fantasia	1940-11-13	Musical	G	83320000	2187090808	
3	Song of the South	1946-11-12	Adventure	G	65000000	1078510579	
4	Cinderella	1950-02-15	Drama	G	85000000	920608730	
...	...	...	...	...	...	...	
574	The Light Between Oceans	2016-09-02	Drama	PG-13	12545979	12545979	
575	Queen of Katwe	2016-09-23	Drama	PG	8874389	8874389	
576	Doctor Strange	2016-11-04	Adventure	PG-13	232532923	232532923	
577	Moana	2016-11-23	Adventure	PG	246082029	246082029	
578	Rogue One: A Star Wars Story	2016-12-16	Adventure	PG-13	529483936	529483936	

579 rows × 7 columns

In [31]:
```
# Import seaborn library as sb
import seaborn as sb

# Plot the line-plot for attribute total_gross to depict histogram
sb.relplot(x="release_year", y="inflation_adjusted_gross", kind="line", hue="genre", data=genre_yearly)
```

Out[31]: <seaborn.axisgrid.FacetGrid at 0x21680af0d90>



Name – Mohammed Safwan Aslam Kazi
NYIT ID – 1270795
Course – Data Mining
Project Title – The Android App Market on Google Play
Source – https://www.kaggle.com/lava18/google-play-store-apps/googleplaystore.csv
https://www.kaggle.com/lava18/google-play-store-apps/googleplaystore_user_reviews.csv

1. Exploration of the data set of Google Play Store apps and reviews

Mobile apps are everywhere. They are easy to create and can be lucrative. Because of these two factors, more and more apps are being developed. In this notebook, we will do a comprehensive analysis of the Android App Market by comparing over 10K apps in Google Play Store across different categories. We will look for insights in the data to devise strategies to drive growth and retention. The data contains 9658 Apps with thirteen attributes/features: app, category, rating, reviews, size, installs, type, price, content rating, genres, last updated, current version and android version.

Let's load the data set (**source: https://www.kaggle.com/lava18/google-play-store-apps/googleplaystore.csv**) and explore the given data set: -

In [9]:
```
# Import pandas library as pd
import pandas as pd

# Read the csv file and store as a data frame into df_apps variable
df_apps = pd.read_csv("D:/NYIT (Fall2020-ACADEMICS)/INTRODUCTION TO DATA MININ
G (CSCI657 or CSCI415) [PROFF Helen Gu]/PROJECTS/The Android App Market on Goo
gle Play/datasets/apps.csv")

# Print the output of df_apps variable
df_apps
```

Out[9]:

	Unnamed: 0	App	Category	Rating	Reviews	Size	Installs	Typ
0	0	Photo Editor & Candy Camera & Grid & ScrapBook	ART_AND_DESIGN	4.1	159	19.0	10,000+	Fre
1	1	Coloring book moana	ART_AND_DESIGN	3.9	967	14.0	500,000+	Fre
2	2	U Launcher Lite – FREE Live Cool Themes, Hide ...	ART_AND_DESIGN	4.7	87510	8.7	5,000,000+	Fre
3	3	Sketch - Draw & Paint	ART_AND_DESIGN	4.5	215644	25.0	50,000,000+	Fre
4	4	Pixel Draw - Number Art Coloring Book	ART_AND_DESIGN	4.3	967	2.8	100,000+	Fre
...	...	...	...	...	...	...	...	
9654	10836	Sya9a Maroc - FR	FAMILY	4.5	38	53.0	5,000+	Fre
9655	10837	Fr. Mike Schmitz Audio Teachings	FAMILY	5.0	4	3.6	100+	Fre
9656	10838	Parkinson Exercices FR	MEDICAL	NaN	3	9.5	1,000+	Fre
9657	10839	The SCP Foundation DB fr nn5n	BOOKS_AND_REFERENCE	4.5	114	NaN	1,000+	Fre
9658	10840	iHoroscope - 2018 Daily Horoscope & Astrology	LIFESTYLE	4.5	398307	19.0	10,000,000+	Fre

9659 rows × 14 columns

In [16]:
```
# Calculates the length of records of DataFrame "df_apps"
print("Total number of apps present in given data set:\n", len(df_apps))

# If-else condition to show whether there are any duplicate records in the dat
a set
if df_apps.duplicated().any():
    print("\nThere are" + len(df_apps_duplicates) + "duplicate apps in the giv
en data set")
else:
    print("\nSince the altered data set is equal to unaltered data set, theref
ore, there are no duplicate apps present in the actual data set")
```

```
Total number of apps present in given data set:
 9659

Since the altered data set is equal to unaltered data set, therefore, there a
re no duplicate apps present in the actual data set
```

In [62]:
```
# Concise summary of DataFrame "df_apps"
print(df_apps.info())
print("\n")

# Basic statistical details
print(df_apps.describe())
print("\n")

# Checks whether the DataFrame "df_apps" has any N/A values
df_apps.isna().sum()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9659 entries, 0 to 9658
Data columns (total 14 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Unnamed: 0      9659 non-null   int64
 1   App             9659 non-null   object
 2   Category        9659 non-null   object
 3   Rating          8196 non-null   float64
 4   Reviews         9659 non-null   int64
 5   Size            8432 non-null   float64
 6   Installs        9659 non-null   int64
 7   Type            9659 non-null   object
 8   Price           9659 non-null   float64
 9   Content Rating  9659 non-null   object
 10  Genres          9659 non-null   object
 11  Last Updated    9659 non-null   object
 12  Current Ver     9651 non-null   object
 13  Android Ver     9657 non-null   object
dtypes: float64(3), int64(3), object(8)
memory usage: 1.0+ MB
None
```

	Unnamed: 0	Rating	Reviews	Size	Installs
count	9659.000000	8196.000000	9.659000e+03	8432.000000	9.659000e+03
mean	5666.172896	4.173243	2.165926e+05	20.395327	7.777507e+06
std	3102.362863	0.536625	1.831320e+06	21.827509	5.375828e+07
min	0.000000	1.000000	0.000000e+00	0.000000	0.000000e+00
25%	3111.500000	4.000000	2.500000e+01	4.600000	1.000000e+03
50%	5814.000000	4.300000	9.670000e+02	12.000000	1.000000e+05
75%	8327.500000	4.500000	2.940100e+04	28.000000	1.000000e+06
max	10840.000000	5.000000	7.815831e+07	100.000000	1.000000e+09

	Price
count	9659.000000
mean	1.099299
std	16.852152
min	0.000000
25%	0.000000
50%	0.000000
75%	0.000000
max	400.000000

Out[62]:
```
Unnamed: 0          0
App                 0
Category            0
Rating           1463
Reviews             0
Size             1227
Installs            0
Type                0
Price               0
Content Rating      0
Genres              0
Last Updated        0
Current Ver         8
Android Ver         2
dtype: int64
```

In [29]:
```
# Have a look at a random sample of n rows
n = 5
df_apps.sample(n)
```

Out[29]:

	Unnamed: 0	App	Category	Rating	Reviews	Size	Installs	Type	Price
8158	9278	EF Parents	FAMILY	4.1	435	17.0	50,000+	Free	0
3491	4391	Traps n' Gemstones	GAME	4,5	413	9,1	1,000+	Paid	$4,99
6344	7391	Annonces,ci	SHOPPING	NaN	14	5,2	1,000+	Free	0
1447	1787	Hello Stars	GAME	4.6	101686	31.0	10,000,000+	Free	0
2777	3513	CM FILE MANAGER HD	PRODUCTIVITY	4,3	144879	NaN	10,000,000+	Free	0

2. Data Cleaning

The four features that we will be working with most frequently henceforth are installs, size, rating and price. The info() function (from the previous task) told us that installs and price columns are of type object and not int64 or float64 as we would expect. This is because the column contains some characters more than just [0,9] digits. Ideally, we would want these columns to be numeric as their name suggests. Hence, we now proceed to data cleaning and prepare our data to be consumed in our analysis later. Specifically, the presence of special characters (, $ +) in the installs and price columns make their conversion to a numerical data type difficult.

In [37]:
```
# List of characters to remove
chars_to_remove = ['+', ',', '$']

# List of column names to clean
cols_to_clean = ['Installs', 'Price']

# Loop for each column
for col in cols_to_clean:
    # Replace each character with an empty string
    for char in chars_to_remove:
        df_apps[col] = df_apps[col].astype(str).str.replace(char, '')
    # Convert col to numeric
    df_apps[col] = pd.to_numeric(df_apps[col])
```

3. Exploring app categories

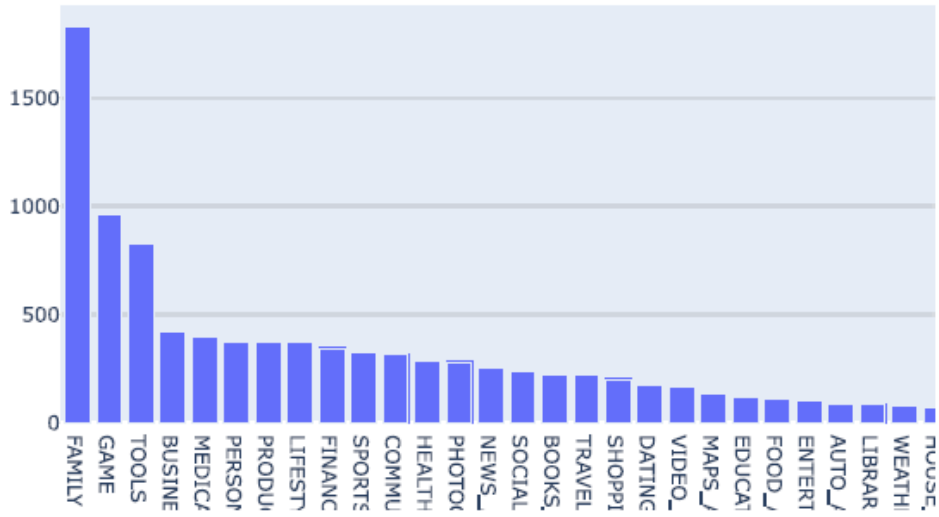
With more than 1 billion active users in 190 countries around the world, Google Play continues to be an important distribution platform
to build a global audience. For businesses to get their apps in front of users, it's important to make them more quickly and easily discoverable on Google Play. To improve the overall search experience, Google has introduced the concept of grouping apps into categories. This brings us to the following questions:

1) Which category has the highest share of (active) apps in the market?
2) Is any specific category dominating the market?
3) Which categories have the fewest number of apps?

We will see that there are 33 unique app categories present in our dataset, where, “Family” and “Game” apps have the highest market prevalence. Interestingly, “Tools”, “Business” and “Medical” apps are also at the top.

In [39]:
```
import plotly
plotly.offline.init_notebook_mode(connected=True)
import plotly.graph_objs as go

# Print the total number of unique categories
num_categories = len(df_apps['Category'].unique())
print('Number of categories = ', num_categories)

# Count the number of apps in each 'Category' and sort them in descending orde
r
num_apps_in_category = df_apps['Category'].value_counts().sort_values(ascendin
g = False)

data = [go.Bar(
        x = num_apps_in_category.index, # index = category name
        y = num_apps_in_category.values, # value = count
)]

plotly.offline.iplot(data)
```

```
Number of categories =  33
```



4. Distribution of app ratings

After having witnessed the market share for each category of apps, let's see how all these apps perform on an average. App ratings (on a scale of 1 to 5) impact the discoverability, conversion of apps as well as the company's overall brand image. Ratings are a key performance indicator of an app.
From our research, we found that the average volume of ratings across all app categories is 4.17. The histogram plot is skewed to the left indicating that the

majority of the apps are highly rated with only a few exceptions in the low-rated apps.

In [41]:
```
# Average rating of apps
avg_app_rating = df_apps['Rating'].mean()
print('Average app rating = ', avg_app_rating)

# Distribution of apps according to their ratings
data = [go.Histogram(
        x = df_apps['Rating']
)]

# Vertical dashed line to indicate the average app rating
layout = {'shapes': [{
              'type' :'line',
              'x0': avg_app_rating,
              'y0': 0,
              'x1': avg_app_rating,
              'y1': 1000,
              'line': { 'dash': 'dashdot'}
          }]
          }

plotly.offline.iplot({'data': data, 'layout': layout})
```

```
Average app rating =  4.173243045387998
```

5. Size and price of an app

Let's now examine app size and app price. For size, if the mobile app is too large, it may be difficult and/or expensive for users to download. Lengthy download times could turn users off before they even experience your mobile app. Plus,

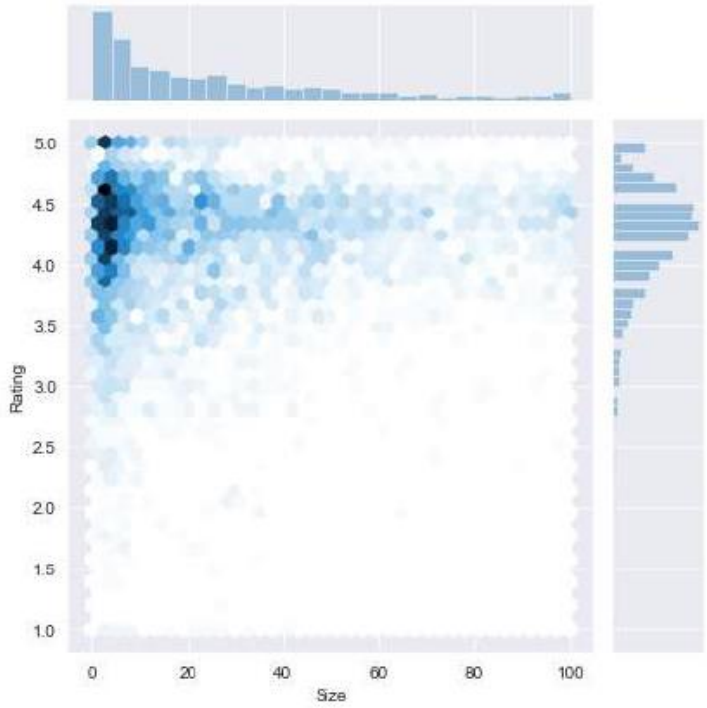
each user's device has a finite amount of disk space. For price, some users expect their apps to be free or inexpensive. These problems compound if the developing world is part of your target market; especially due to internet speeds, earning power and exchange rates.

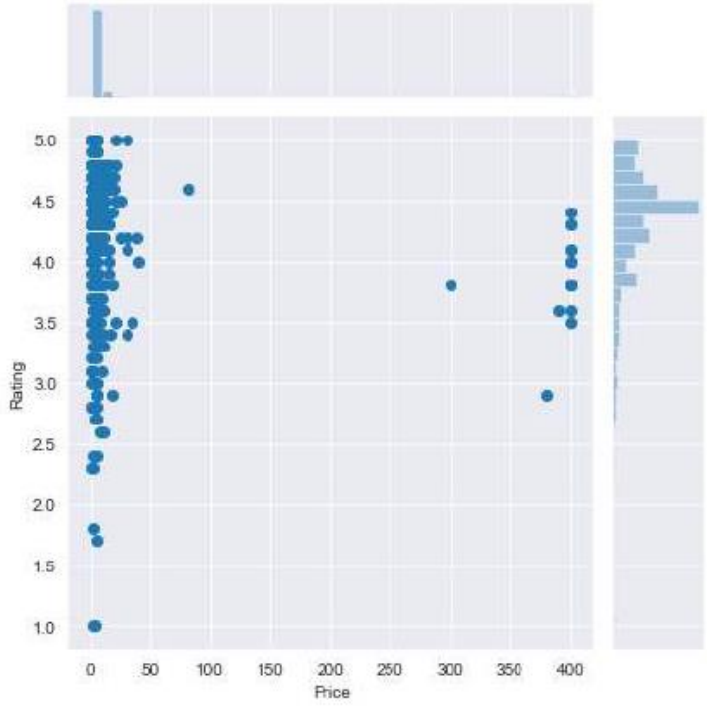
How can we effectively come up with strategies to size and price our app?

1) Does the size of an app affect its rating?
2) Do users really care about system-heavy apps or do they prefer light-weighted apps?
3) Does the price of an app affect its rating?
4) Do users always prefer free apps over paid apps?

We find that the majority of top-rated apps (rating over 4) range from 2 MB to 20 MB. We also find that the vast majority of apps price themselves under $10.

In [42]:
```
%matplotlib inline
import seaborn as sb
sb.set_style("darkgrid")
import warnings
warnings.filterwarnings("ignore")

# Filter rows where both Rating and Size values are not null
apps_with_size_and_rating_present = df_apps[(~df_apps['Rating'].isnull()) & (~
df_apps['Size'].isnull())]

# Subset for categories with at least 250 apps
large_categories = apps_with_size_and_rating_present.groupby(['Category']).fil
ter(lambda x: len(x) >= 250).reset_index()

# Plot size vs. rating
plt1 = sb.jointplot(x = large_categories['Size'], y = large_categories['Ratin
g'], kind = 'hex')

# Subset apps whose 'Type' is 'Paid'
paid_apps = apps_with_size_and_rating_present[apps_with_size_and_rating_presen
t['Type'] == 'Paid']

# Plot price vs. rating
plt2 = sb.jointplot(x = paid_apps['Price'], y = paid_apps['Rating'])
```





6. <u>Relation between app category and app price</u>

So now comes the hard part. And we must consider certain factors like,

1) How are companies and developers supposed to make ends meet?
2) What monetization strategies can companies use to maximize profit?

Usually, the costs of apps are largely based on features, complexity, and platform.

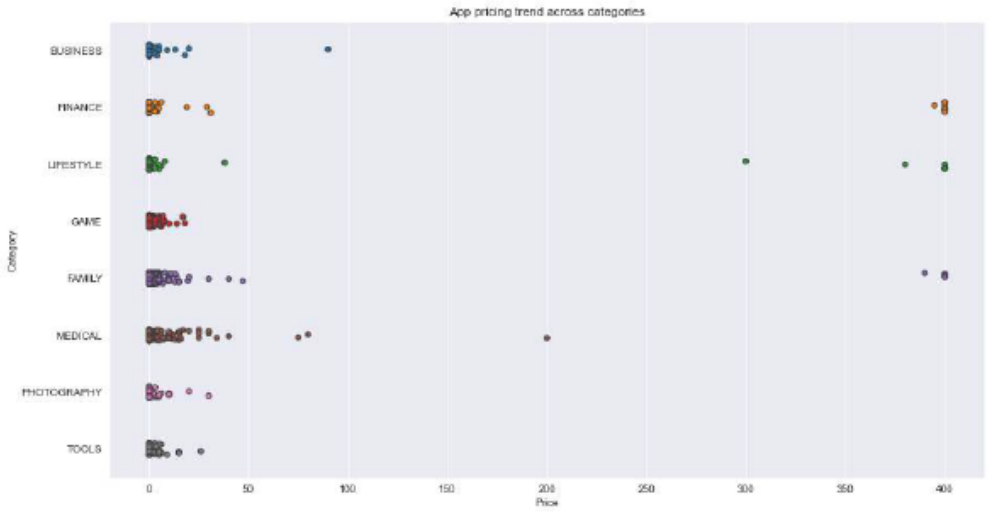
There are many more factors to consider when selecting the right pricing strategy for your mobile app. It is important to consider the willingness of your customer to pay for your app. A wrong price could break the deal before the download even happens. Potential customers could be turned off by what they perceive to be a shocking cost, or they might delete an app they've downloaded after receiving too many ads or simply not getting their money's worth.

Different categories demand different price ranges. Some apps that are simple and used daily, like the calculator app, should probably be kept free. However, it would make sense to charge for a highly-specialized medical app that diagnoses diabetic patients. Below, we see that *Medical and Family* apps are the most expensive. Some medical apps extend even up to $80! All game apps are reasonably priced below $20.

In [43]:
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
fig.set_size_inches(15, 8)

# Select a few popular app categories
popular_app_cats = df_apps[df_apps.Category.isin(['GAME', 'FAMILY', 'PHOTOGRAP
HY',
                                                  'MEDICAL', 'TOOLS', 'FINANCE',
                                                  'LIFESTYLE','BUSINESS'])]

# Examine the price trend by plotting Price vs Category
ax = sb.stripplot(x = popular_app_cats['Price'], y = popular_app_cats['Categor
y'], jitter=True, linewidth=1)
ax.set_title('App pricing trend across categories')

# Apps whose Price is greater than 200
apps_above_200 = popular_app_cats[['Category', 'App', 'Price']][popular_app_ca
ts['Price'] > 200]
apps_above_200
```

Out[43]:

	Category	App	Price
3327	FAMILY	most expensive app (H)	399,99
3465	LIFESTYLE	💎 I'm rich	399.99
3469	LIFESTYLE	I'm Rich - Trump Edition	400.00
4396	LIFESTYLE	I am rich	399,99
4398	FAMILY	I am Rich Plus	399.99
4399	LIFESTYLE	I am rich VIP	299.99
4400	FINANCE	I Am Rich Premium	399,99
4401	LIFESTYLE	I am extremely Rich	379.99
4402	FINANCE	I am Rich!	399.99
4403	FINANCE	I am rich(premium)	399,99
4406	FAMILY	I Am Rich Pro	399.99
4408	FINANCE	I am rich (Most expensive app)	399.99
4410	FAMILY	I Am Rich	389,99
4413	FINANCE	I am Rich	399.99
4417	FINANCE	I AM RICH PRO PLUS	399,99
8763	FINANCE	Eu Sou Rico	394.99
8780	LIFESTYLE	I'm Rich/Eu sou Rico/أنا غني/我很有錢	399.99

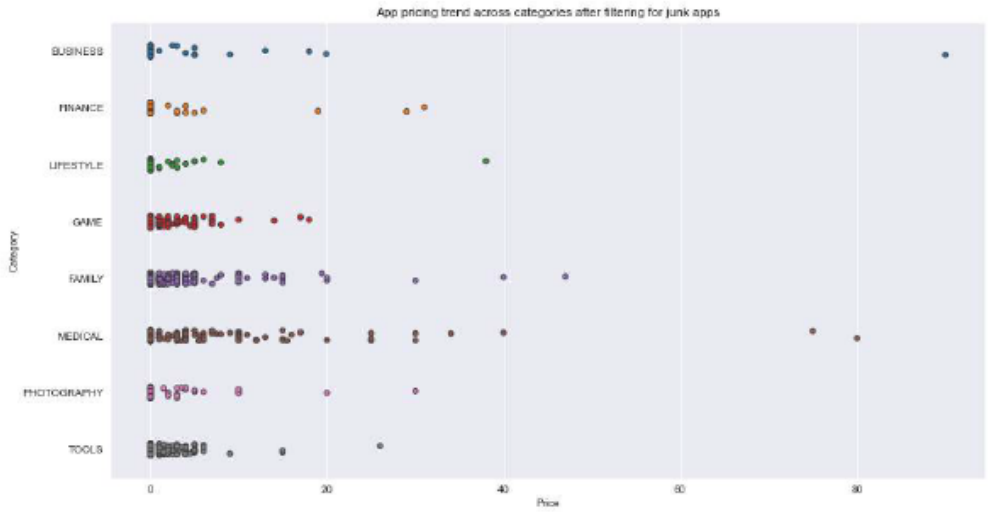


7. Filter out “junk” apps

It looks like a bunch of the really expensive apps are "junk" apps. That is, apps that don't really have a purpose. Some app developer may create an app called “I Am Rich Premium” or “most expensive” *app* (H) just for a joke or to test their app development skills. Some developers even do this with malicious intent and try to make money by hoping people accidentally click purchase on their app in the store. Let's filter out these junk apps and re-do our visualization.

In [49]:
```
# Select apps priced below $100
df_apps_under_100 = popular_app_cats[popular_app_cats['Price'] < 100]

fig, ax = plt.subplots()
fig.set_size_inches(15, 8)

# Examine price vs category with the authentic apps (apps_under_100)
ax = sb.stripplot(x='Price', y='Category', data=apps_under_100,
                  jitter=True, linewidth=1)
ax.set_title('App pricing trend across categories after filtering for junk app
s')
```

Out[49]:
```
Text(0.5, 1.0, 'App pricing trend across categories after filtering for junk
apps')
```



8. Popularity of paid apps vs free apps

For apps in the Play Store today, there are five types of pricing strategies: free, freemium, paid, paymium, and subscription. Let's focus on free and paid apps only. Some characteristics of free apps are:

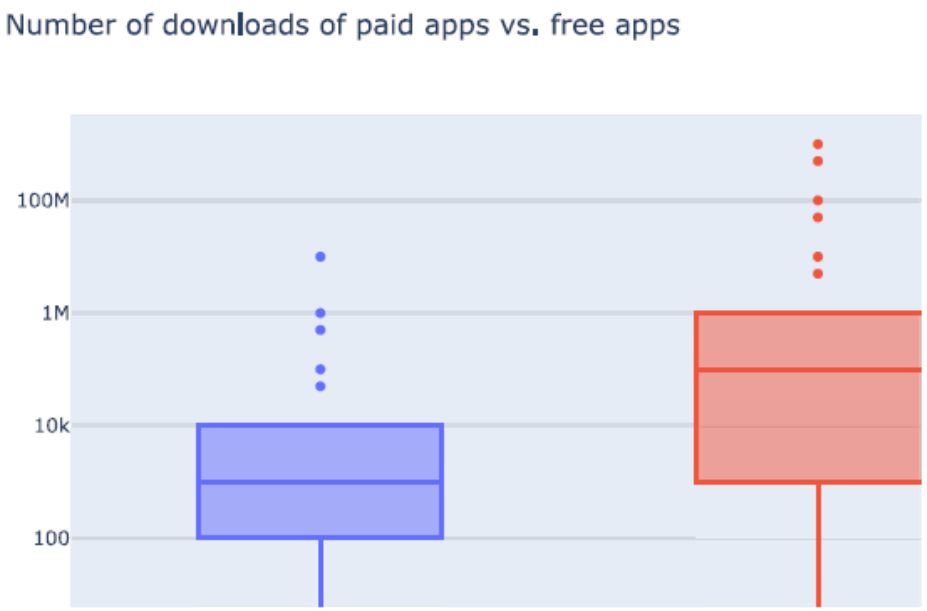
1) Free to download.
2) Main source of income often comes from advertisements.
3) Often created by companies that have other products and the app serves as an extension of those products.
4) Can serve as a tool for customer retention, communication, and customer service.

Some characteristics of paid apps are:

1) Users are asked to pay once for the app to download and use it.
2) The user can't really get a feel for the app before buying it.

Are paid apps installed as much as free apps? It turns out that paid apps have a relatively lower number of installs than free apps,
though the difference is not as stark as I would have expected!

In [50]:
```
trace0 = go.Box(
    # Data for paid apps
    y=df_apps[df_apps['Type'] == 'Paid']['Installs'],
    name = 'Paid'
)

trace1 = go.Box(
    # Data for free apps
    y=df_apps[df_apps['Type'] == 'Free']['Installs'],
    name = 'Free'
)

layout = go.Layout(
    title = "Number of downloads of paid apps vs. free apps",
    yaxis = dict(
        type = 'log',
        autorange = True
    )
)

# Add trace0 and trace1 to a list
data = [trace0, trace1]
plotly.offline.iplot({'data': data, 'layout': layout})
```



9. Sentiment analysis of user reviews

Mining user review data to determine how people feel about your product, brand, or service can be done using a technique called sentiment analysis. User reviews for apps can be analysed to identify if the mood is positive, negative or neutral about that app. For example, positive words in an app review might include words such as 'amazing', 'friendly', 'good', 'great', and 'love'. Negative words might be words like 'malware', 'hate', 'problem', 'refund', and 'incompetent'.

By plotting sentiment polarity scores of user reviews for paid and free apps, we observe that free apps receive a lot of harsh comments, as indicated by the outliers on the negative y-axis. Reviews for paid apps appear never to be extremely negative. This may indicate something about app quality, i.e., paid apps being of higher quality than free apps on average. The median polarity score for paid apps is a little higher than free apps, thereby syncing with our previous observation.

The data contains 64295 records of reviews given by users with five attributes/features: app, translated review, sentiment, sentiment polarity, and sentiment subjectivity.

Let's load the data set (**source: https://www.kaggle.com/lava18/google-play-store-apps/googleplaystore_user_reviews.csv**) and explore the given dataset.

In this notebook, we analysed over 64K apps from the Google Play Store. We can use our findings to inform our decisions should we ever wish to create an app ourselves.

In [52]:
```
# Import pandas library as pd
import pandas as pd

# Read the csv file and store as a data frame into df_user_reviews variable
df_user_reviews = pd.read_csv("D:/NYIT (Fall2020-ACADEMICS)/INTRODUCTION TO DATA MINING (CSCI657 or CSCI415) [PROFF Helen Gu]/PROJECTS/The Android App Market on Google Play/datasets/user_reviews.csv")

# Print the output of df_user_reviews variable
df_user_reviews
```

Out[52]:

	App	Translated_Review	Sentiment	Sentiment_Polarity	Sentiment_Subjectivity
0	10 Best Foods for You	I like eat delicious food. That's I'm cooking ...	Positive	1,00	0,533333
1	10 Best Foods for You	This help eating healthy exercise regular basis	Positive	0.25	0.288462
2	10 Best Foods for You	NaN	NaN	NaN	NaN
3	10 Best Foods for You	Works great especially going grocery store	Positive	0,40	0,875000
4	10 Best Foods for You	Best idea us	Positive	1.00	0.300000
...	...	...	...	...	...
64290	Houzz Interior Design Ideas	NaN	NaN	NaN	NaN
64291	Houzz Interior Design Ideas	NaN	NaN	NaN	NaN
64292	Houzz Interior Design Ideas	NaN	NaN	NaN	NaN
64293	Houzz Interior Design Ideas	NaN	NaN	NaN	NaN
64294	Houzz Interior Design Ideas	NaN	NaN	NaN	NaN

64295 rows × 5 columns

In [63]:
```
# Calculates the length of records of DataFrame "df_user_reviews"
print("Total number of reviews from the user are:\n", len(df_user_reviews))
print("\n")

# Concise summary of DataFrame "df_user_reviews"
print(df_user_reviews.info())
print("\n")

# Basic statistical details
print(df_user_reviews.describe())
print("\n")

# Checks whether the DataFrame "df_user_reviews" has any N/A values
df_user_reviews.isna().sum()
```

```
Total number of reviews from the user are:
 64295


<class 'pandas.core.frame.DataFrame'>
RangeIndex: 64295 entries, 0 to 64294
Data columns (total 5 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   App                     64295 non-null  object
 1   Translated_Review       37427 non-null  object
 2   Sentiment               37432 non-null  object
 3   Sentiment_Polarity      37432 non-null  float64
 4   Sentiment_Subjectivity  37432 non-null  float64
dtypes: float64(2), object(3)
memory usage: 2.5+ MB
None


       Sentiment_Polarity  Sentiment_Subjectivity
count        37432.000000            37432.000000
mean             0.182146                0.492704
std              0.351301                0.259949
min             -1.000000                0.000000
25%              0.000000                0.357143
50%              0.150000                0.514286
75%              0.400000                0.650000
max              1.000000                1.000000
```

Out[63]:
```
App                        0
Translated_Review      26868
Sentiment              26863
Sentiment_Polarity     26863
Sentiment_Subjectivity 26863
dtype: int64
```

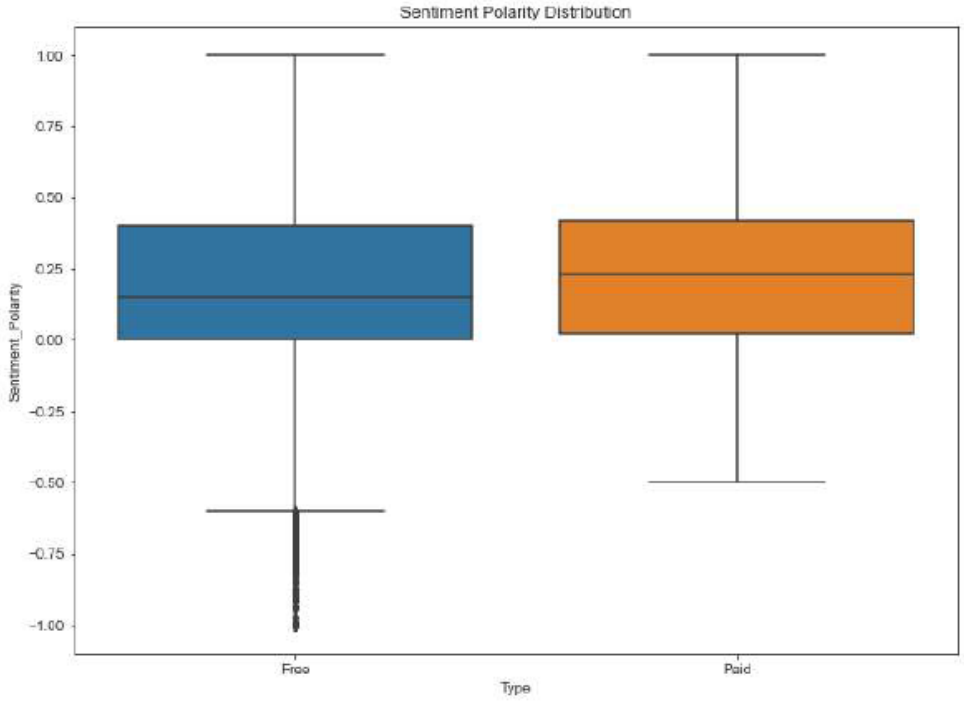
In [58]:
```
# Have a look at a random sample of n rows
n = 5
df_user_reviews.sample(n)
```

Out[58]:

	App	Translated_Review	Sentiment	Sentiment_Polarity	Sentiment_Subjectivity
21802	Calorie Counter - Macros	Update! 4-4-18 NOT work Samsung 8. Please fix!...	Positive	0.225000	0.537500
28032	Color by Number – New Coloring Book	Addicted! Would love ability create account I ...	Positive	0.160000	0.470000
23205	Candy Camera – selfie, beauty camera, photo ed...	Excellent app. It greatly improved Xgody Y19 c...	Positive	0.633333	0.711111
49734	Free live weather on screen	Such good	Positive	0.350000	0.550000
43140	Facetune - Ad Free	I think overall amazing app. You simple editin...	Positive	0.300000	0.491429

In [61]:
```
# Load user_reviews.csv
#reviews_df = pd.read_csv('D:/NYIT (Fall2020-ACADEMICS)/INTRODUCTION TO DATA M
INING (CSCI657 or CSCI415) [PROFF Helen Gu]/PROJECTS/The Android App Market on
Google Play/datasets/user_reviews.csv')

# Inner join and merge
merged_df = pd.merge(df_apps, df_user_reviews, on = "App", how = "inner")

# Drop NA values from Sentiment and Translated_Review columns
merged_df = merged_df.dropna(subset=['Sentiment', 'Translated_Review'])

sb.set_style('ticks')
fig, ax = plt.subplots()
fig.set_size_inches(11, 8)

# User review sentiment polarity for paid vs. free apps
ax = sb.boxplot(x = 'Type', y = 'Sentiment_Polarity', data=merged_df)
ax.set_title('Sentiment Polarity Distribution')
```

Out[61]:
```
Text(0.5, 1.0, 'Sentiment Polarity Distribution')
```



Rishabh Badjatya

NYIT ID – 1270662

Course – Data Mining

Project Title - A Visual History of Nobel Prize Winners

1. The most Nobel of Prizes

The Nobel Prize is perhaps the world's most well-known scientific award. Except for the honor, prestige, and substantial prize money the recipient also gets a gold medal showing Alfred Nobel (1833 - 1896) who established the prize. Every year it's given to scientists and scholars in the category's chemistry, literature, physics, physiology or medicine, economics, and peace. The first Nobel Prize was handed out in 1901, and at that time the Prize was very Eurocentric and male-focused, but nowadays it is not biased in any way whatsoever. Surely. Right?

Well, we're going to find out! The Nobel Foundation has made a dataset available of all prize winners from the start of the prize, in 1901, to 2016. Let us load it in and look.

In [13]:
```
# Loading in required libraries
import pandas as pd
import seaborn as sns
import numpy as np
```

In [18]:
```
# Reading in the Nobel Prize data
nobel = pd.read_csv('C:/Users/risha/OneDrive/Desktop/Projects/A Visual History
of Nobel Prize Winners/datasets/nobel.csv',na_values='n/a')
nobel.replace('NA', np.nan, inplace=True)
nobel.replace('Missing', np.nan, inplace=True)
nobel.dtypes
nobel.info()


# Taking a look at the first several winners
nobel.head(n=6)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 911 entries, 0 to 910
Data columns (total 18 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   year                  911 non-null    int64
 1   category              911 non-null    object
 2   prize                 911 non-null    object
 3   motivation            823 non-null    object
 4   prize_share           911 non-null    object
 5   laureate_id           911 non-null    int64
 6   laureate_type         911 non-null    object
 7   full_name             911 non-null    object
 8   birth_date            883 non-null    object
 9   birth_city            883 non-null    object
 10  birth_country         885 non-null    object
 11  sex                   885 non-null    object
 12  organization_name     665 non-null    object
 13  organization_city     667 non-null    object
 14  organization_country  667 non-null    object
 15  death_date            593 non-null    object
 16  death_city            576 non-null    object
 17  death_country         582 non-null    object
dtypes: int64(2), object(16)
memory usage: 128.2+ KB
```

Out[18]:

	year	category	prize	motivation	prize_share	laureate_id	laureate_type	full_name
0	1901	Chemistry	The Nobel Prize in Chemistry 1901	"in recognition of the extraordinary services ...	1/1	160	Individual	Jacobus Henricus van 't Hoff
1	1901	Literature	The Nobel Prize in Literature 1901	"in special recognition of his poetic composit...	1/1	569	Individual	Sully Prudhomme
2	1901	Medicine	The Nobel Prize in Physiology or Medicine 1901	"for his work on serum therapy, especially its...	1/1	293	Individual	Emil Adolf von Behring
3	1901	Peace	The Nobel Peace Prize 1901	NaN	1/2	462	Individual	Jean Henry Dunant
4	1901	Peace	The Nobel Peace Prize 1901	NaN	1/2	463	Individual	Frédéric Passy
5	1901	Physics	The Nobel Prize in Physics 1901	"in recognition of the extraordinary services ...	1/1	1	Individual	Wilhelm Conrad Röntgen

2. So, who gets the Nobel Prize?

Just looking at the first couple of prize winners, or Nobel laureates as they are also called, we already see a celebrity: Wilhelm Conrad Roentgen, the guy who discovered X-rays. And we see that all of the winners in 1901 were guys that came from Europe. But that was back in 1901, looking at all winners in the dataset, from 1901 to 2016, which sex and which country is the most represented?

(For *country*, we will use the birth country of the winner, as the organization country is Nan for all shared Nobel Prizes.)

In [19]:
```
# Display the number of (possibly shared) Nobel Prizes handed
# out between 1901 and 2016
display(len(nobel))

# Display the number of prizes won by male and female recipients.
display(nobel['sex'].value_counts())

# Display the number of prizes won by the top 10 nationalities.
nobel['birth_country'].value_counts().head(10)
```

```
911

Male      836
Female     49
Name: sex, dtype: int64
```

Out[19]:
```
United States of America    259
United Kingdom               85
Germany                      61
France                       51
Sweden                       29
Japan                        24
Netherlands                  18
Canada                       18
Italy                        17
Russia                       17
Name: birth_country, dtype: int64
```

3. USA dominance

Not so surprising perhaps: the most common Nobel laureate between 1901 and 2016 was a man born in the United States of America. But in 1901 all the winners were European. When did the USA start to dominate the Nobel Prize charts?

In [20]:
```
# Calculating the proportion of USA born winners per decade
nobel['usa_born_winner'] = nobel['birth_country'] == 'United States of Americ
a'
nobel['decade'] = (np.floor(nobel['year'] / 10) * 10).astype(int)
prop_usa_winners = nobel.groupby('decade', as_index=False)['usa_born_winner'].
mean()

# Display the proportions of USA born winners per decade
prop_usa_winners
```

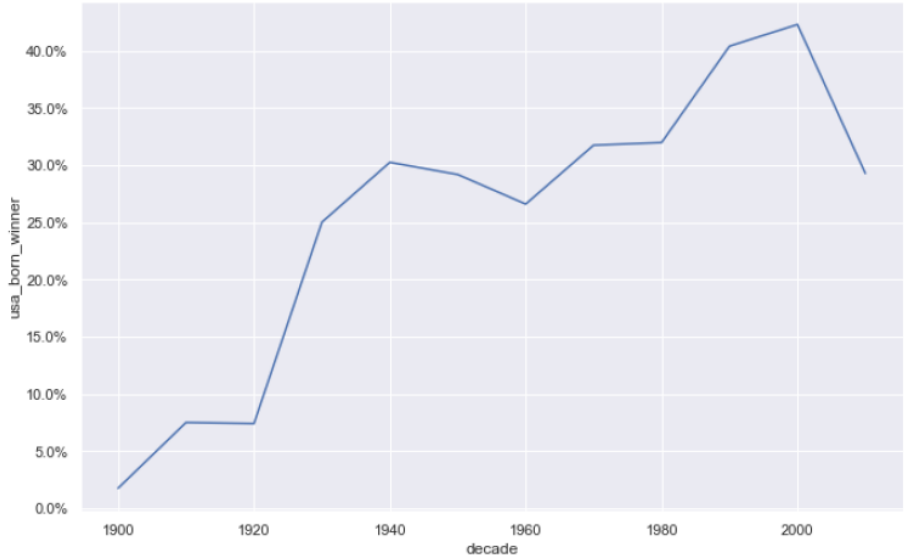
Out[20]:

	decade	usa_born_winner
0	1900	0.017544
1	1910	0.075000
2	1920	0.074074
3	1930	0.250000
4	1940	0.302326
5	1950	0.291667
6	1960	0.265823
7	1970	0.317308
8	1980	0.319588
9	1990	0.403846
10	2000	0.422764
11	2010	0.292683

4. USA dominance, visualized

A table is OK, but to *see* when the USA started to dominate the Nobel charts, we need a plot!

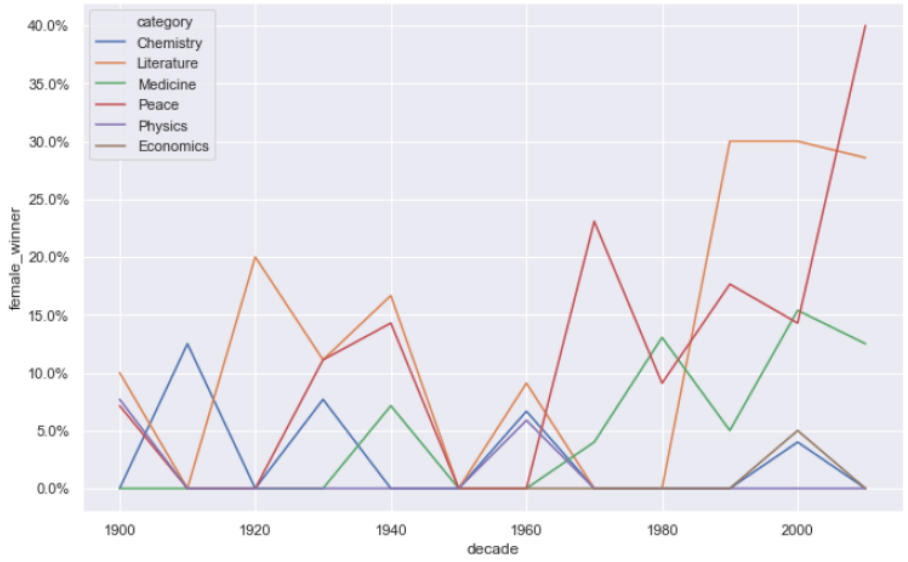
In [21]:
```
# Setting the plotting theme
sns.set()
# and setting the size of all plots.
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [11, 7]

# Plotting USA born winners
ax = sns.lineplot(x='decade', y='usa_born_winner', data=prop_usa_winners)

# Adding %-formatting to the y-axis
from matplotlib.ticker import PercentFormatter
ax.yaxis.set_major_formatter(PercentFormatter(1.0))
```



5. What is the gender of a typical Nobel Prize winner?

So, the USA became the dominating winner of the Nobel Prize first in the 1930s and had kept the leading position ever since. But one group that was in the lead from the start, and never seems to let go, are *men*. Maybe it shouldn't come as a shock that there is some imbalance between how many male and female prize winners there are, but how significant is this imbalance? And is it better or worse within specific prize categories like physics, medicine, literature, etc.?

In [22]:
```
# Calculating the proportion of female laureates per decade
nobel['female_winner'] = nobel['sex'] == 'Female'
prop_female_winners = nobel.groupby(['decade', 'category'], as_index=False)['f
emale_winner'].mean()

# Plotting female winners with % winners on the y-axis
ax = sns.lineplot(x='decade', y='female_winner', hue='category', data=prop_fem
ale_winners)
ax.yaxis.set_major_formatter(PercentFormatter(1.0))
```



In [ ]:

Data Set - https://www.kaggle.com/nobelfoundation/nobel-laureates.

Rishabh Badjatya
NYIT ID – 1270662

Course – Data Mining
Project Title - Analyze Your Runkeeper Fitness Data

1. Obtain and review raw data

One day, my old running friend and I were chatting about our running styles, training habits, and achievements, when I suddenly realized that I could take an in-depth analytical look at my training. I have been using a popular GPS fitness tracker called Runkeeper for years and decided it was time to analyze my running data to see how I was doing.

Since 2012, I've been using the Runkeeper app, and it's great. One key feature: its excellent data export. Anyone who has a smartphone can download the app and analyze their data like we will in this notebook.

After logging your run, the first step is to export the data from Runkeeper (which I have done already). Then import the data and start exploring to find potential problems. After that, create data cleaning strategies to fix the issues. Finally, analyze and visualize the clean time-series data.

I exported seven years' worth of my training data, from 2012 through 2018. The data is a CSV file where each row is a single training activity. Let us load and inspect it.

In [11]:
```
# Import pandas
import pandas as pd

# Define file containing dataset
runkeeper_file = 'C:/Users/risha/OneDrive/Desktop/Projects/Analyze Your Runkee
per Fitness Data/datasets/cardioActivities.csv'

# Create DataFrame with parse_dates and index_col parameters
df_activities = pd.read_csv(runkeeper_file, parse_dates=True, index_col='Date'
)

# First look at exported data: select sample of 3 random rows
display(df_activities.sample(n=3))

# Print DataFrame summary
df_activities.info()
```

Date	Activity Id	Type	Route Name	Distance (km)	Duration	Average Pace	Average Speed (km/h)	Calories Burned	Climb (m)
2018-06-13 18:50:49	2f10b609-c2ef-4d8d-90ad-bd49cec2c41b	Running	NaN	12.75	1:07:30	5:18	11.34	890.0	173
2014-10-09 18:10:00	df510a68-0447-4ca6-ade8-d558cd700e76	Running	NaN	9.61	52:17	5:26	11.03	677.0	54
2013-10-12 09:36:00	2c71d785-7846-47f1-81f0-55ea80e6e22c	Running	NaN	6.36	34:21	5:24	11.11	455.0	35

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 508 entries, 2018-11-11 14:05:12 to 2012-08-22 18:53:54
Data columns (total 13 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Activity Id              508 non-null    object
 1   Type                     508 non-null    object
 2   Route Name               1 non-null      object
 3   Distance (km)            508 non-null    float64
 4   Duration                 508 non-null    object
 5   Average Pace             508 non-null    object
 6   Average Speed (km/h)     508 non-null    float64
 7   Calories Burned          508 non-null    float64
 8   Climb (m)                508 non-null    int64
 9   Average Heart Rate (bpm) 294 non-null    float64
 10  Friend's Tagged          0 non-null      float64
 11  Notes                    231 non-null    object
 12  GPX File                 504 non-null    object
dtypes: float64(5), int64(1), object(7)
memory usage: 55.6+ KB
```

2. Data preprocessing

Lucky for us, the column names Runkeeper provides are informative, and we don't need to rename any columns.

But we do notice missing values using the info () method. What are the reasons for these missing values? It depends. Some heart rate information is missing because I did not always use a cardio sensor. In the case of the Notes column, it is an optional field that I sometimes left blank. Also, I only used the Route Name column once, and never used the Friend's Tagged column.

We will fill in missing values in the heart rate column to avoid misleading results later, but right now, our first data preprocessing steps will be to:

- Remove columns not useful for our analysis.
- Replace the "Other" activity type to "Unicycling" because that was always the "Other" activity.
- Count missing values.

In [12]:
```
# Define list of columns to be deleted
cols_to_drop = ['Friend\'s Tagged','Route Name','GPX File','Activity Id','Calo
ries Burned', 'Notes']

# Delete unnecessary columns
df_activities.drop(columns=cols_to_drop, inplace=True)

# Count types of training activities
display(df_activities['Type'].value_counts())

# Rename 'Other' type to 'Unicycling'
df_activities['Type'] = df_activities['Type'].str.replace('Other', 'Unicyclin
g')

# Count missing values for each column
df_activities.isnull().sum()
```

```
Running    459
Cycling     29
Walking     18
Other        2
Name: Type, dtype: int64
```

Out[12]:
```
Type                        0
Distance (km)               0
Duration                    0
Average Pace                0
Average Speed (km/h)        0
Climb (m)                   0
Average Heart Rate (bpm)    214
dtype: int64
```

3. Dealing with missing values

As we can see from the last output, there are 214 missing entries for my average heart rate.

We can't go back in time to get those data, but we can fill in the missing values with an average value. This process is called *mean imputation*. When imputing the mean to fill in missing data, we need to consider that the average heart rate varies for different activities (e.g., walking vs. running). We'll filter the Data Frames by activity type (`Type`) and calculate each activity's mean heart rate, then fill in the missing values with those means.

In [13]:
```
# Calculate sample means for heart rate for each training activity type
avg_hr_run = df_activities[df_activities['Type'] == 'Running']['Average Heart
 Rate (bpm)'].mean()
avg_hr_cycle = df_activities[df_activities['Type'] == 'Cycling']['Average Hear
t Rate (bpm)'].mean()

# Split whole DataFrame into several, specific for different activities
df_run = df_activities[df_activities['Type'] == 'Running'].copy()
df_walk = df_activities[df_activities['Type'] == 'Walking'].copy()
df_cycle = df_activities[df_activities['Type'] == 'Cycling'].copy()

# Filling missing values with counted means
df_walk['Average Heart Rate (bpm)'].fillna(110, inplace=True)
df_run['Average Heart Rate (bpm)'].fillna(int(avg_hr_run), inplace=True)
df_cycle['Average Heart Rate (bpm)'].fillna(int(avg_hr_cycle), inplace=True)

# Count missing values for each column in running data
df_run.isnull().sum()
```

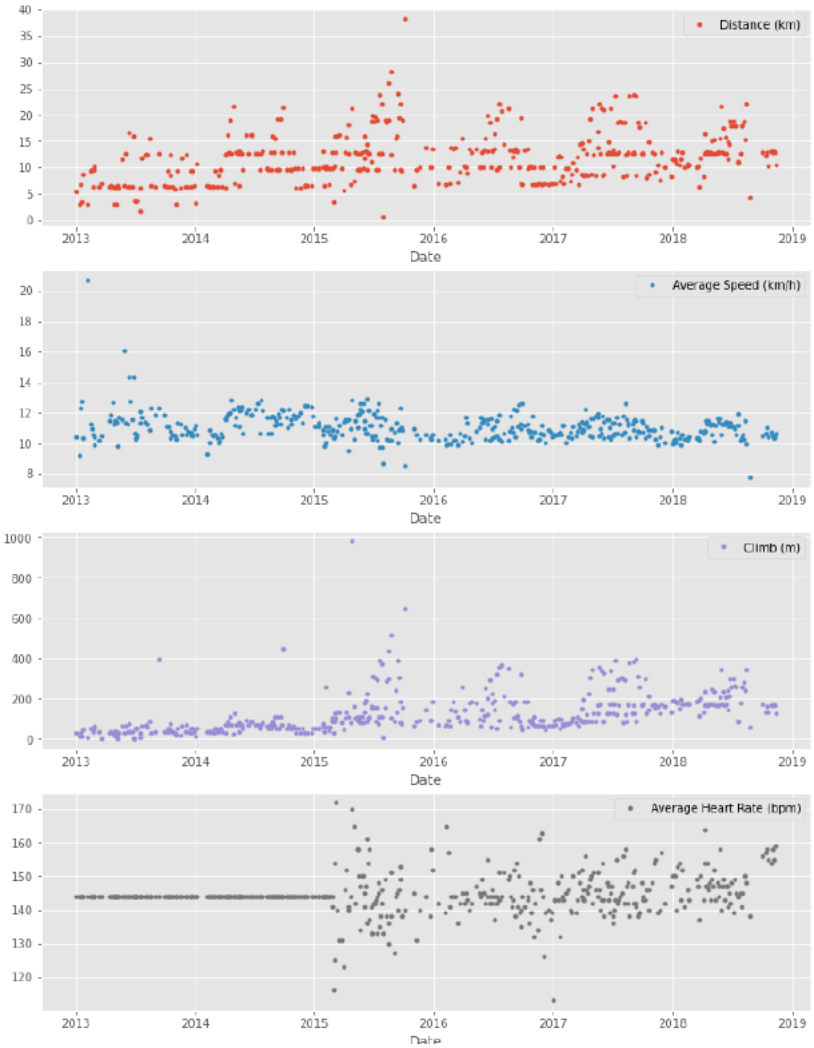
Out[13]:
```
Type                        0
Distance (km)               0
Duration                    0
Average Pace                0
Average Speed (km/h)        0
Climb (m)                   0
Average Heart Rate (bpm)    0
dtype: int64
```

4. Plot running data

Now we can create our first plot! As we found earlier, most of the activities in my data were running (459 of them to be exact). There are only 29, 18, and two instances for cycling, walking, and unicycling, respectively. So, for now, let us focus on plotting the different running metrics.

An excellent first visualization is a figure with four subplots, one for each running metric (each numerical column). Each subplot will have a different y-axis, which is explained in each legend. The x-axis, `Date`, is shared among all subplots.

In [14]:
```
%matplotlib inline

# Import matplotlib, set style and ignore warning
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
plt.style.use('ggplot')
warnings.filterwarnings(
    action='ignore', module='matplotlib.figure', category=UserWarning,
    message=('This figure includes Axes that are not compatible with tight_lay
out, so results might be incorrect.')
)

# Prepare data subsetting period from 2013 till 2018
runs_subset_2013_2018 = df_run['2018':'2013']

# Create, plot and customize in one step
runs_subset_2013_2018.plot(subplots=True,
                           sharex=False,
                           figsize=(12,16),
                           linestyle='none',
                           marker='o',
                           markersize=3,
                          )

# Show plot
plt.show()
```



5. Running statistics

No doubt, running helps people stay mentally and physically healthy and productive at any age. And it is great fun! When runners talk to each other about their hobby, we not only discuss our results, but we also discuss different training strategies.

You will know you are with a group of runners if you commonly hear questions like:

- What is your average distance?
- How fast do you run?
- Do you measure your heart rate?
- How often do you train?

Let us find the answers to these questions in my data. If you look back at plots in Task 4, you can see the answer to, *do you measure your heart rate?* Before 2015: no. To look at the averages, let's only use the data from 2015 through 2018.

In pandas, the resample () method is similar to the group by () method - with resample() you group by a specific time span. We will use resample () to group the time series data by a sampling period and apply several methods to each sampling period. In our case, we will resample annually and weekly.

In [15]:
```
# Prepare running data for the last 4 years
runs_subset_2015_2018 = df_run['2018':'2015']

# Calculate annual statistics
print('How my average run looks in last 4 years:')
display(runs_subset_2015_2018.resample('A').mean())

# Calculate weekly statistics
print('Weekly averages of last 4 years:')
display(runs_subset_2015_2018.resample('W').mean().mean())

# Mean weekly counts
weekly_counts_average = runs_subset_2015_2018['Distance (km)'].resample('W').c
ount().mean()
print('How many trainings per week I had on average:', weekly_counts_average)
```

```
How my average run looks in last 4 years:
```

Date	Distance (km)	Average Speed (km/h)	Climb (m)	Average Heart Rate (bpm)
2015-12-31	13.602805	10.998902	160.170732	143.353659
2016-12-31	11.411667	10.837778	133.194444	143.388889
2017-12-31	12.935176	10.959059	169.376471	145.247059
2018-12-31	13.339063	10.777969	191.218750	148.125000

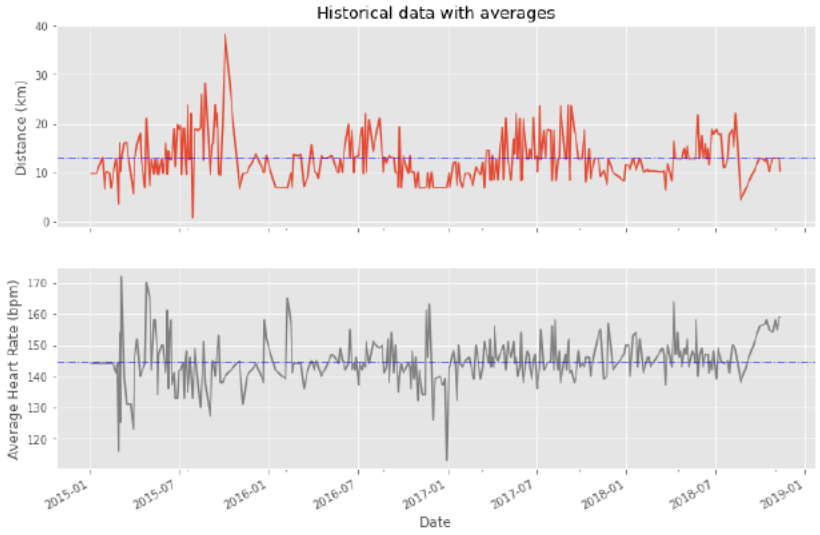
```
Weekly averages of last 4 years:

Distance (km)               12.518176
Average Speed (km/h)        10.835473
Climb (m)                  158.325444
Average Heart Rate (bpm)   144.801775
dtype: float64

How many trainings per week I had on average: 1.5
```

6. Visualization with averages

Let us plot the long-term averages of my distance run and my heart rate with their raw data to visually compare the averages to each training session. Again, we'll use the data from 2015 through 2018.

In this task, we will use `matplotlib` functionality for plot creation and customization.

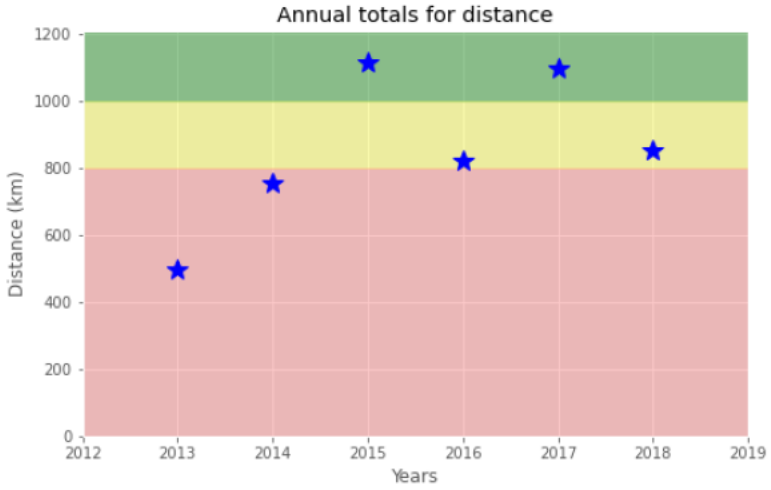
In [16]:
```
# Prepare data
runs_subset_2015_2018 = df_run['2018':'2015']
runs_distance = runs_subset_2015_2018['Distance (km)']
runs_hr = runs_subset_2015_2018['Average Heart Rate (bpm)']

# Create plot
fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))

# Plot and customize first subplot
runs_distance.plot(ax=ax1)
ax1.set(ylabel='Distance (km)', title='Historical data with averages')
ax1.axhline(runs_distance.mean(), color='blue', linewidth=1, linestyle='-.')

# Plot and customize second subplot
runs_hr.plot(ax=ax2, color='gray')
ax2.set(xlabel='Date', ylabel='Average Heart Rate (bpm)')
ax2.axhline(runs_hr.mean(), color='blue', linewidth=1, linestyle='-.')

# Show plot
plt.show()
```



7. Did I reach my goals?

To motivate myself to run regularly, I set a target goal of running 1000 km per year. Let's visualize my annual running distance (km) from 2013 through 2018 to see if I reached my goal each year. Only stars in the green region indicate success.

In [17]:
```
# Prepare data
df_run_dist_annual = df_run['2018':'2013']['Distance (km)'].resample('A').sum
()

# Create plot
fig = plt.figure(figsize=(8, 5))

# Plot and customize
ax = df_run_dist_annual.plot(marker='*', markersize=14, linewidth=0, color='bl
ue')
ax.set(ylim=[0, 1210],
       xlim=['2012','2019'],
       ylabel='Distance (km)',
       xlabel='Years',
       title='Annual totals for distance')

ax.axhspan(1000, 1210, color='green', alpha=0.4)
ax.axhspan(800, 1000, color='yellow', alpha=0.3)
ax.axhspan(0, 800, color='red', alpha=0.2)

# Show plot
plt.show()
```



Dataset **- https://github.com/AilingLiu/Analyze-Your-Runkeeper-Fitness-Data/tree/master/datasets**

NAME: Maitry Amishkumar Jariwala

NYIT ID: 1275288

COURSE: Data Mining

PROJECT TITLE: Level Difficulty in Candy Crush Saga

URL: https://github.com/ryanschaub/Level-Difficulty-in-Candy-Crush-Saga/blob/master/candy_crush.csv

1. Candy Crush Saga

Candy Crush Saga is a hit mobile game developed by King (part of Activision Blizzard) that is played by millions of people all around the world. The game is structured as a series of levels where players need to match similar candy together to (hopefully) clear the level and keep progressing on the level map.

Candy Crush has more than 3000 levels, and new ones are added every week. That is a lot of levels! And with that many levels, it's important to get *level difficulty* just right. Too easy and the game gets boring, too hard and players become frustrated and quit playing.

In this project, we will see how we can use data collected from players to estimate level difficulty. Let's start by loading in the packages we're going to need.

In [1]:
```
# This sets the size of plots to a good default.
options(repr.plot.width = 5, repr.plot.height = 4)

# Loading in packages
library(dplyr)
library(datasets)
library(ggplot2)
library(readr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

2. The data set

The dataset we will use contains one week of data from a sample of players who played Candy Crush back in 2014. The data is also from a single *episode*, that is, a set of 15 levels. It has the following columns:

- **player_id**: a unique player id
- **dt**: the date
- **level**: the level number within the episode, from 1 to 15.
- **num_attempts**: number of level attempts for the player on that level and date.
- **num_success**: number of level attempts that resulted in a success/win for the player on that level and date.

The granularity of the dataset is player, date, and level. That is, there is a row for every player, day, and level recording the total number of attempts and how many of those resulted in a win.

Now, let's load in the dataset and take a look at the first couple of rows.

```
# Reading in the data
data <- read_csv("datasets/candy_crush.csv")

# Printing out the first six rows
head(data)
```

```
Parsed with column specification:
cols(
  player_id = col_character(),
  dt = col_date(format = ""),
  level = col_double(),
  num_attempts = col_double(),
  num_success = col_double()
)
```

player_id	dt	level	num_attempts	num_success
<chr>	<date>	<dbl>	<dbl>	<dbl>
6dd5af4c7228fa353d505767143f5815	2014-01-04	4	3	1
c7ec97c39349ab7e4d39b4f74062ec13	2014-01-01	8	4	1
c7ec97c39349ab7e4d39b4f74062ec13	2014-01-05	12	6	0
a32c5e9700ed356dc8dd5bb3230c5227	2014-01-03	11	1	1
a32c5e9700ed356dc8dd5bb3230c5227	2014-01-07	15	6	0
b94d403ac4edf639442f93eeffdc7d92	2014-01-01	8	8	1

3.Checking the data set

Now that we have loaded the dataset let's count how many players we have in the sample and how many days' worth of data we have.

In [25]:
```
# Count and display the number of unique players
print("Number of players:")
length(unique(data$player_id))

# Display the date range of the data
print("Period for which we have data:")
range(data$dt)
```

```
[1] "Number of players:"
6814
[1] "Period for which we have data:"
2014-01-01  2014-01-07
```

4. Computing level difficulty

Within each Candy Crush episode, there is a mix of easier and tougher levels. Luck and individual skill make the number of attempts required to pass a level different from player to player. The assumption is that difficult levels require more attempts on average than easier ones. That is, *the harder* a level is, *the lower* the probability to pass that level in a single attempt is.

A simple approach to model this probability is as a Bernoulli process; as a binary outcome (you either win or lose) characterized by a single parameter p_{win}: the probability of winning the level in a single attempt. This probability can be estimated for each level as:

$$p_{win} = \frac{\sum wins}{\sum attempts}$$

For example, let's say a level has been played 10 times and 2 of those attempts ended up in a victory. Then the probability of winning in a single attempt would be p_{win} = 2 / 10 = 20%.

Now, let's compute the difficulty p_{win} separately for each of the 15 levels.

In [3]:

```
# Calculating level difficulty
difficulty <- data %>%
  group_by(level) %>%
  summarise(attempts = sum(num_attempts), wins = sum(num_success)) %>%
  mutate(p_win = wins / attempts)

# Printing out the calculated difficulty
difficulty
```

A tibble: 15 x 4

level	attempts	wins	p_win
<dbl>	<dbl>	<dbl>	<dbl>
1	1322	818	0.61875946
2	1285	666	0.51828794
3	1546	662	0.42820181
4	1893	705	0.37242472
5	6937	634	0.09139397
6	1591	668	0.41986172
7	4526	614	0.13566063
8	15816	641	0.04052858
9	8241	670	0.08130081
10	3282	617	0.18799512
11	5575	603	0.10816143
12	6868	659	0.09595224
13	1327	686	0.51695554
14	2772	777	0.28030303
15	30374	1157	0.03809179

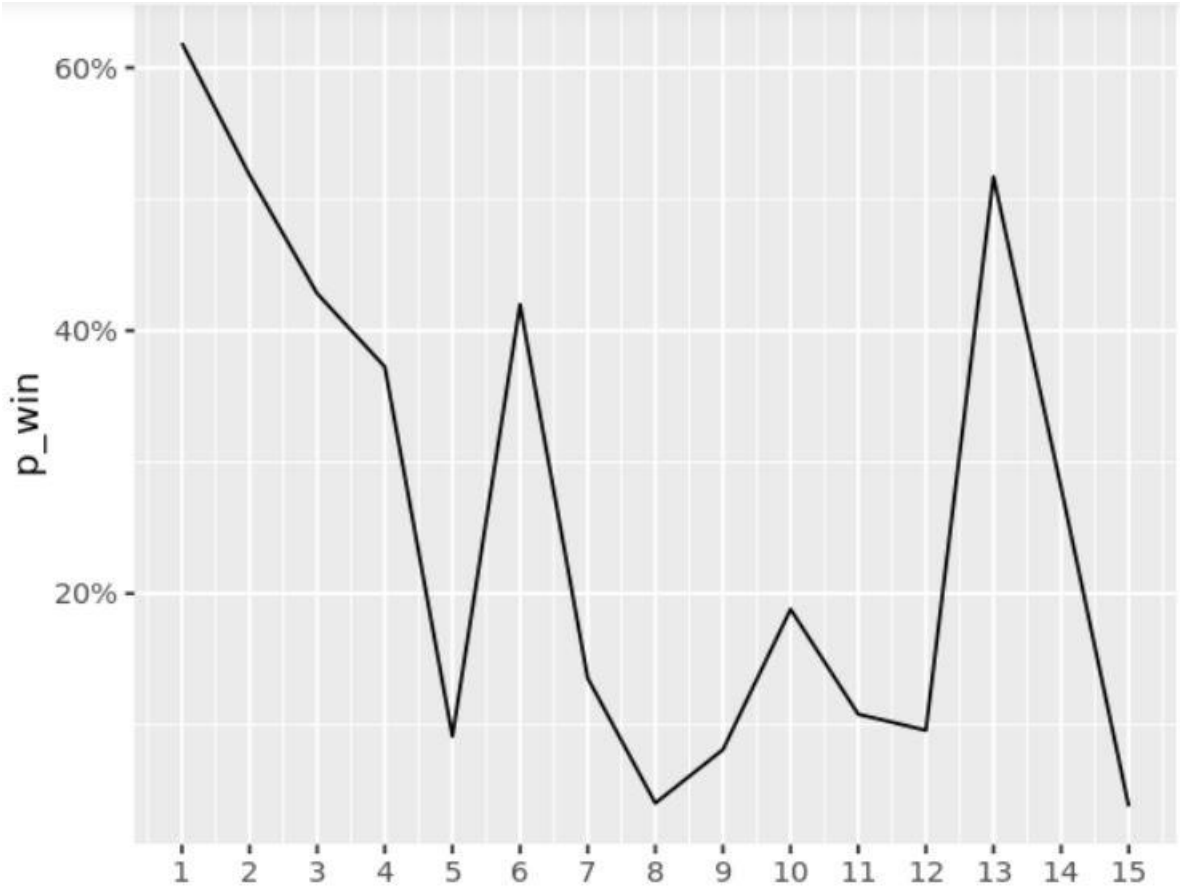
5. Plotting difficulty profile

Great! We now have the difficulty for all the 15 levels in the episode. Keep in mind that, as we measure difficulty as the probability to pass a level in a single attempt, a *lower* value (a smaller probability of winning the level) implies a *higher* level difficulty.

Now that we have the difficulty of the episode we should plot it. Let's plot a line graph with the levels on the X-axis and the difficulty (p_{win}) on the Y-axis. We call this plot the *difficulty profile* of the episode.

In [29]:

```
# Plotting the level difficulty profile
difficulty %>%
  ggplot(aes(x = level, y = p_win)) +
    geom_line() +
    scale_x_continuous(breaks = 1:15) +
    scale_y_continuous(label = scales::percent)
```

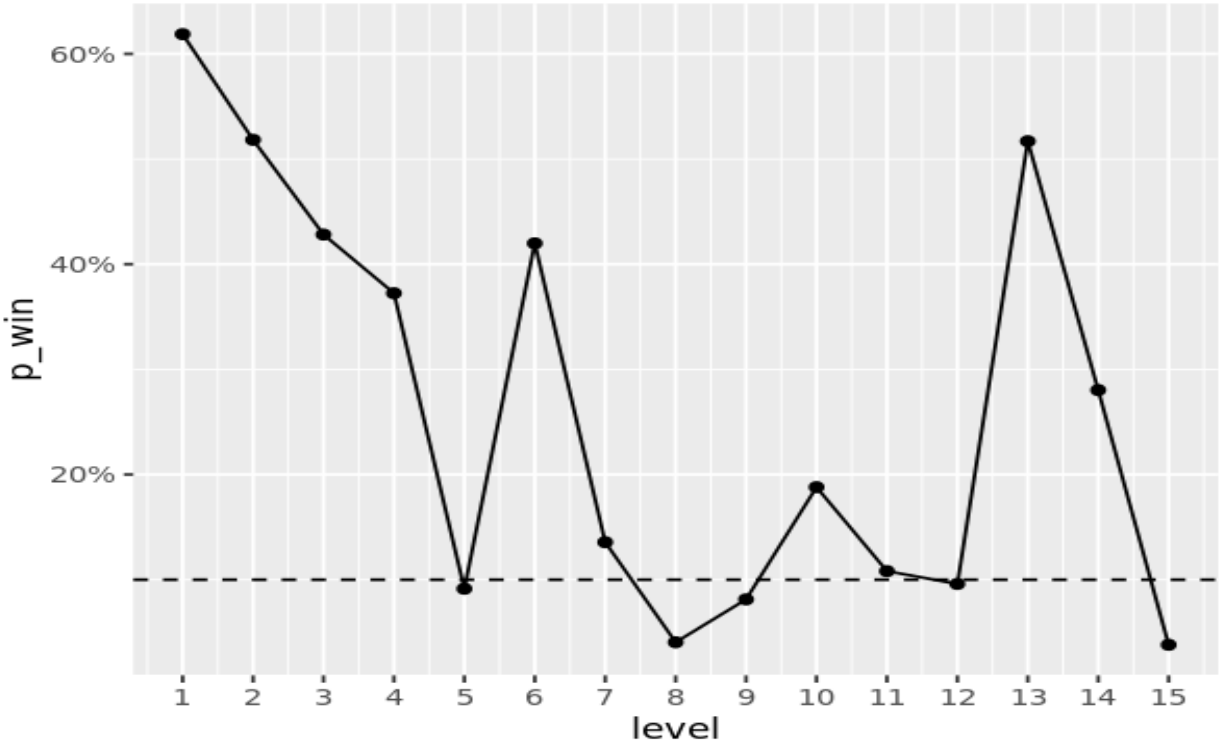


6. Spotting hard levels

What constitutes a *hard* level is subjective. However, to keep things simple, we could define a threshold of difficulty, say 10%, and label levels with p_{win} < 10% as *hard*. It's relatively easy to spot these hard levels on the plot, but we can make the plot more friendly by explicitly highlighting the hard levels.

In [31]:

```
# Adding points and a dashed line
difficulty %>%
  ggplot(aes(x = level, y = p_win)) +
    geom_line() + geom_point() +
    scale_x_continuous(breaks = 1:15) +
    scale_y_continuous(label = scales::percent) +
    geom_hline(yintercept = 0.1, linetype = 'dashed')
```



7. Computing uncertainty

As Data Scientists we should always report some measure of the uncertainty of any provided numbers. Maybe tomorrow, another sample will give us slightly different values for the difficulties? Here we will simply use the *Standard error* as a measure of uncertainty:

$$\sigma_{error} \approx \frac{\sigma_{sample}}{\sqrt{n}}$$

Here n is the number of datapoints and σ_{sample} is the sample standard deviation. For a Bernoulli process, the sample standard deviation is:

$$\sigma_{sample} = \sqrt{p_{win}(1 - p_{win})}$$

Therefore, we can calculate the standard error like this:

$$\sigma_{error} \approx \sqrt{\frac{p_{win}(1 - p_{win})}{n}}$$

We already have all we need in the difficulty data frame! Every level has been played n number of times and we have their difficulty p_{win}. Now, let's calculate the standard error for each level.

In [1]:

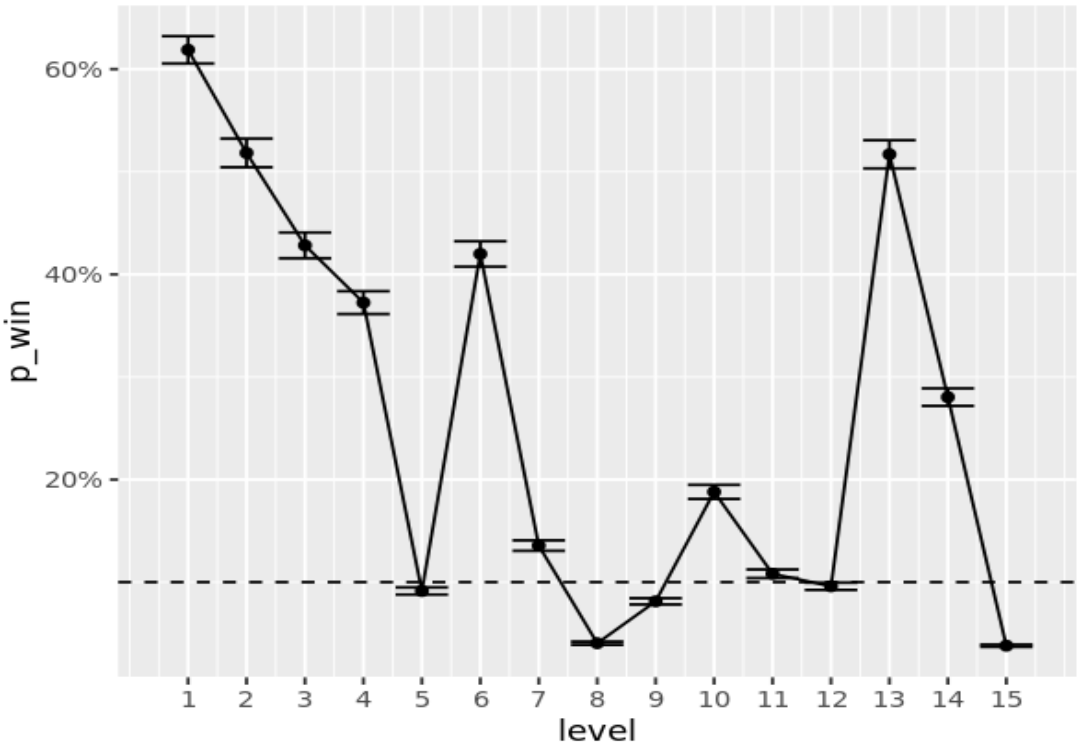
```
# Computing the standard error of p_win for each level
difficulty <- difficulty %>%
  mutate(error = sqrt(p_win * (1 - p_win) / attempts))
```

8. Showing uncertainty

Now that we have a measure of uncertainty for each levels' difficulty estimate let's use *error bars* to show this uncertainty in the plot. We will set the length of the error bars to one standard error. The upper limit and the lower limit of each error bar should then be $p_{win} + \sigma_{error}$ and $p_{win} - \sigma_{error}$, respectively.

In [35]:

```
# Adding standard error bars
difficulty %>%
  ggplot(aes(x = level, y = p_win)) +
    geom_line() + geom_point() +
    scale_x_continuous(breaks = 1:15) +
    scale_y_continuous(label = scales::percent) +
    geom_hline(yintercept = 0.1, linetype = 'dashed') +
    geom_errorbar(aes(ymin = p_win - error, ymax = p_win + error))
```



9. A final metric

It looks like our difficulty estimates are pretty precise! Using this plot, a level designer can quickly spot where the hard levels are and also see if there seems to be too many hard levels in the episode.

One question a level designer might ask is: "How likely is it that a player will complete the episode without losing a single time?" Let's calculate this using the estimated level difficulties!

In [37]:

```
# The probability of completing the episode without losing a single time
p <- prod(difficulty$p_win)

# Printing it out
p
```

9.44714093448606e-12

NAME: Maitry Amishkumar Jariwala

NYIT ID: 1275288

COURSE: Data Mining

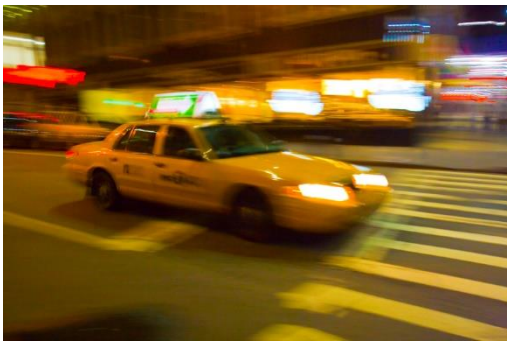
PROJECT TITLE: Predict Taxi Fares with Random Forests

URL: https://github.com/noahweber1/datacamp-project---PREDICT-TAXI-FARES-WITH-RANDOM-FORESTS

Data Attributes Are:

medallion= It is a transferable permit in the United States allowing a taxicab driver to operate
pickup_datetime = Date and time to pickup the passenger
pickup_longitude = Longitude of pickup address
pickup_latitude = Latitude of pickup address
trip_time_in_secs = time taken for the trip
fare_amount = amount of fare taken by passenger
tip_amount = tip amount if given by passenger

1. 49999 New York taxi trips



To drive a yellow New York taxi, you have to hold a "medallion" from the city's *Taxi and Limousine Commission*. Recently, one of those changed hands for over one million dollars, which shows how lucrative the job can be.

But this is the age of business intelligence and analytics! Even taxi drivers can stand to benefit from some careful investigation of the data, guiding them to maximize their profits. In this project, we will analyze a random sample of 49999 New York journeys made in 2013. We will also use regression trees and random forests to build a model that can predict the locations and times when the biggest fares can be earned.

Let us start by taking a look at the data!

In [40]:
```
# Loading the tidyverse
library(tidyverse)

# Reading in the taxi data
taxi <- read_csv('datasets/taxi.csv')

# Taking a look at the first few rows in taxi
head(taxi)
```

```
Parsed with column specification:
cols(
  medallion = col_character(),
  pickup_datetime = col_datetime(format = ""),
  pickup_longitude = col_double(),
  pickup_latitude = col_double(),
  trip_time_in_secs = col_double(),
  fare_amount = col_double(),
  tip_amount = col_double()
```

medallion	pickup_datetime	pickup_longitude	pickup_latitude	trip_time_in_secs	fare_amount	tip_amount
4D24F4D8EF35878595044A52B098DFD2	2013-01-13 10:23:00	-73.94646	40.77273	600	8.0	2.5
A49C37EB966E7B05E69523D1CB7BE303	2013-01-13 04:52:00	-73.99827	40.74041	840	18.0	0.0
1E4B72A8E623888F53A9693C364AC05A	2013-01-13 10:47:00	-73.95346	40.77586	60	3.5	0.7
F7E4E9439C46B8AD5B16AB9F1B3279D7	2013-01-13 11:14:00	-73.98137	40.72473	720	11.5	2.3
A9DC75D59E0EA27E1ED328E8BE8CD828	2013-01-13 11:24:00	-73.96800	40.76000	240	6.5	0.0
19BF1BB516C4E992EA3FBAEDA73D6262	2013-01-13 10:51:00	-73.98502	40.76341	540	8.5	1.7

2. Cleaning the taxi data

As you can see above, the taxi dataset contains the times and price of a large number of taxi trips. Importantly we also get to know the location, the longitude and latitude, where the trip was started.

Cleaning data is a large part of any data scientist's daily work. It may not seem glamorous, but it makes the difference between a successful model and a failure. The taxi dataset needs a bit of polishing before we are ready to use it.

In [42]:
```
# Renaming the location variables,
# dropping any journeys with zero fares and zero tips,
# and creating the total variable as the log sum of fare and tip
taxi <- taxi %>%
   rename(long = pickup_longitude, lat = pickup_latitude)  %>%
   filter(fare_amount > 0 | tip_amount > 0) %>%
   mutate(total = log(fare_amount + tip_amount) )
```

3. Zooming in on Manhattan

While the dataset contains taxi trips from all over New York City, the bulk of the trips are to and from Manhattan, so let's focus only on trips initiated there.

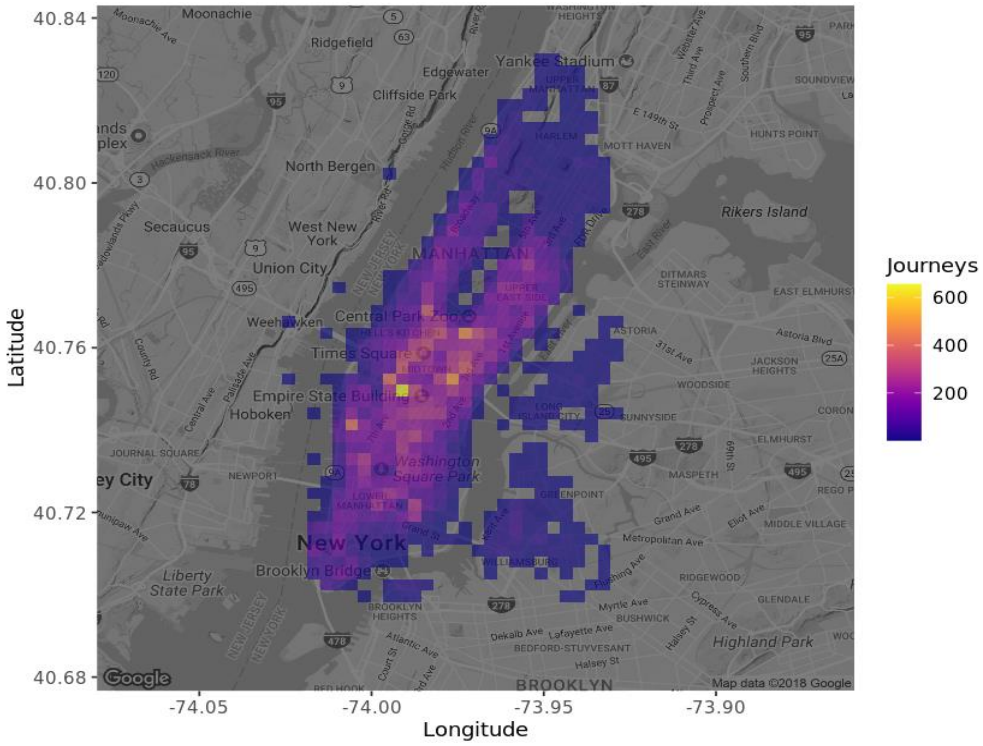
In [3]:
```
# Reducing the data to taxi trips starting in Manhattan
# Manhattan is bounded by the rectangle with
# latitude from 40.70 to 40.83 and
# longitude from -74.025 to -73.93
taxi <- taxi  %>%
    filter(between(lat,  40.70, 40.83) &
           between(long, -74.025, -73.93))
```

4. Where does the journey begin?

It's time to draw a map! We're going to use the excellent ggmap package together with ggplot2 to visualize where in Manhattan people tend to start their taxi journeys.

In [24]:
```
# Loading in ggmap and viridis for nice colors
library(ggmap)
library(viridis)

# Retrieving a stored map object which originally was created by
# nycmap <- get_map("manhattan", zoom = 12, color = "bw")
manhattan <- readRDS("datasets/manhattan.rds")

# Drawing a density map with the number of journey start locations
ggmap(manhattan, darken=0.5)+
    scale_fill_viridis(option='plasma') +
    geom_bin2d(data = taxi, aes(x = pickup_longitude, y = pickup_latitude), bins = 60, alpha = 0.6) +
    labs(x = 'Longitude', y = 'Latitude', fill = 'Journeys')
```

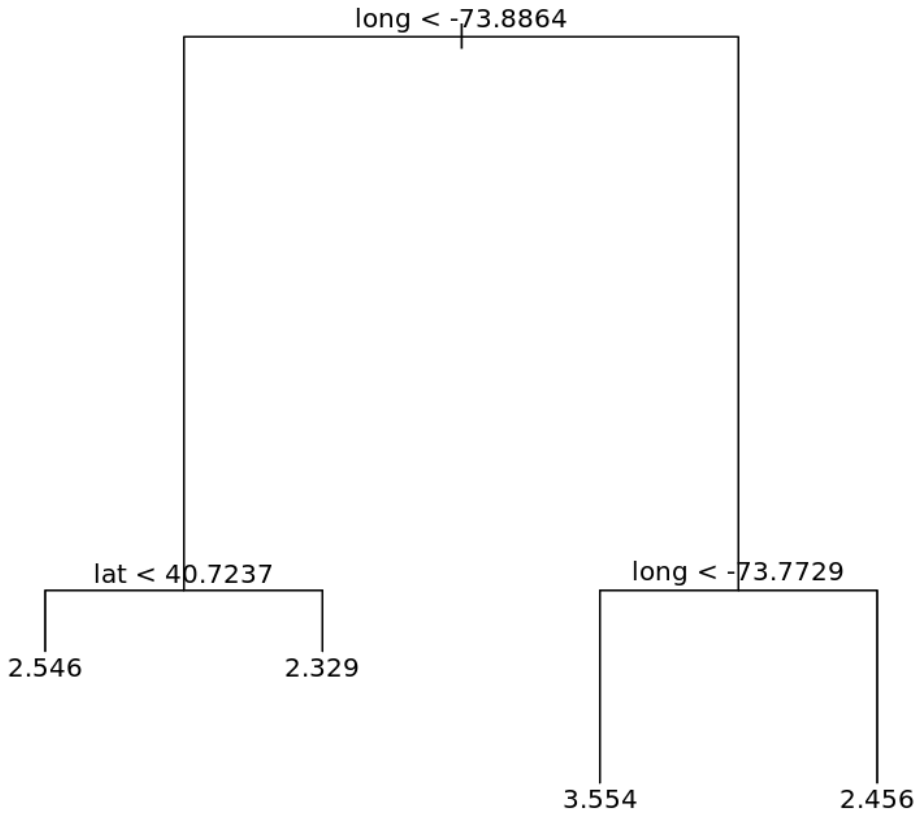
```
Google's Terms of Service: https://cloud.google.com/maps-platform/terms/.
Please cite ggmap if you use it! See citation("ggmap") for details.
Loading required package: viridisLite
```



5. Predicting taxi fares using a tree

The map from the previous task showed that the journeys are highly concentrated in the business and tourist areas. We also see that some taxi trips originating in Brooklyn slipped through, but that's fine. We're now going to use a regression tree to predict the total fare with lat and long being the predictors. The tree algorithm will try to find cut points in those predictors that results in the decision tree with the best predictive capability.

```
In [8]: # Loading in the tree package
        library(tree)

        # Fitting a tree to lat and long
        fitted_tree <- tree(total ~ lat + long, data = taxi)

        # draw a diagram of the tree structure
        plot(fitted_tree)
        text(fitted_tree)
```



6. It's time. More predictors.

The tree above looks a bit frugal, it only includes one split: It predicts that trips where lat < 40.7237 are more expensive, which makes sense as it is downtown Manhattan. But that's it. It didn't even include long as tree deemed that it didn't improve the predictions. Taxi drivers will need more information than this and any driver paying for your data-driven insights would be disappointed with that. As we know from Robert de Niro, it's best not to upset New York taxi drivers. Let's start by adding some more predictors related to the *time* the taxi trip was made.

In [12]:
```
# Loading in the lubridate package
library(lubridate)

# Generate the three new time variables
taxi <- taxi %>%
    mutate(hour = hour(pickup_datetime),
           wday = wday(pickup_datetime, label = TRUE),
           month = month(pickup_datetime, label = TRUE))
```

```
Attaching package: 'lubridate'

The following object is masked from 'package:base':

    date
```

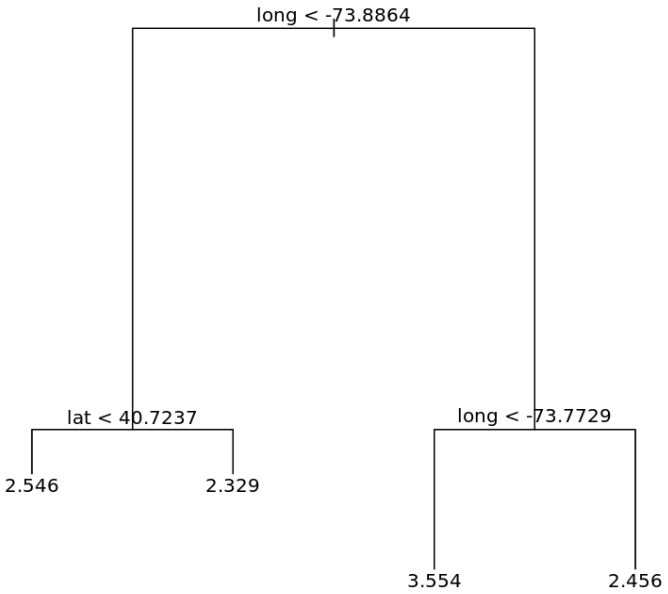
7. One more tree!

Let's try fitting a new regression tree where we include the new time variables.

In [11]:
```
# Fitting a tree to lat and long
fitted_tree <- tree(total ~ lat + long + hour + wday + month, data = taxi)

# draw a diagram of the tree structure
plot(fitted_tree)
text(fitted_tree)

# Summarizing the performance of the tree
summary(fitted_tree)
```

Regression tree:
tree(formula = total ~ lat + long + hour + wday + month, data = taxi)
Variables actually used in tree construction:
[1] "long" "lat"
Number of terminal nodes: 4
Residual mean deviance: 0.3165 = 15820 / 49990
Distribution of residuals:

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-2.63800	-0.38270	-0.02603	0.00000	0.34550	5.17900



8. One tree is not enough

The regression tree has not changed after including the three time variables. This is likely because latitude is still the most promising first variable to split the data on, and after that split, the other variables are not informative enough to be included. A random forest model, where many different trees are fitted to subsets of the data, may well include the other variables in some of the trees that make it up.

In [13]:
```
# Loading in the randomForest package
library(randomForest)

# Fitting a random forest
fitted_forest <- randomForest(total ~ lat + long + hour + wday + month,
    data=taxi, ntree=80, sampsize=10000)

# Printing the fitted_forest object
fitted_forest
```

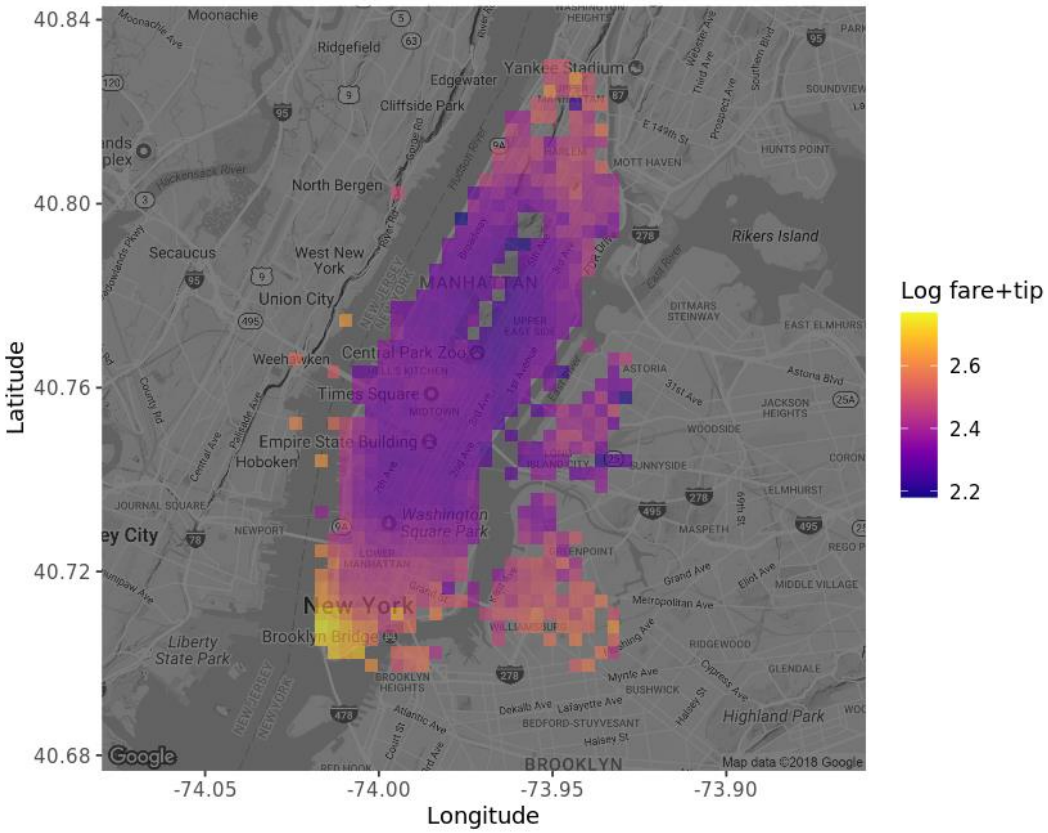
```
randomForest 4.6-14
Type rfNews() to see new features/changes/bug fixes.

Attaching package: 'randomForest'

The following object is masked from 'package:dplyr':

    combine

The following object is masked from 'package:ggplot2':

    margin
```

Call:
randomForest(formula = total ~ lat + long + hour + wday + month, data = taxi, ntree = 80, sampsize = 10000)
Type of random forest: regression
Number of trees: 80
No. of variables tried at each split: 1
Mean of squared residuals: 0.2997314
% Var explained: 2.79

9. Plotting the predicted fare

In the output of fitted_forest you should see the Mean of squared residuals, that is, the average of the squared errors the model makes. If you scroll up and check the summary of fitted_tree you'll find Residual mean deviance which is the same number. If you compare these numbers, you'll see that fitted_forest has a slightly lower error. Neither predictive model is *that* good, in statistical terms, they explain only about 3% of the variance. Now, let's take a look at the predictions of fitted_forest projected back onto Manhattan.

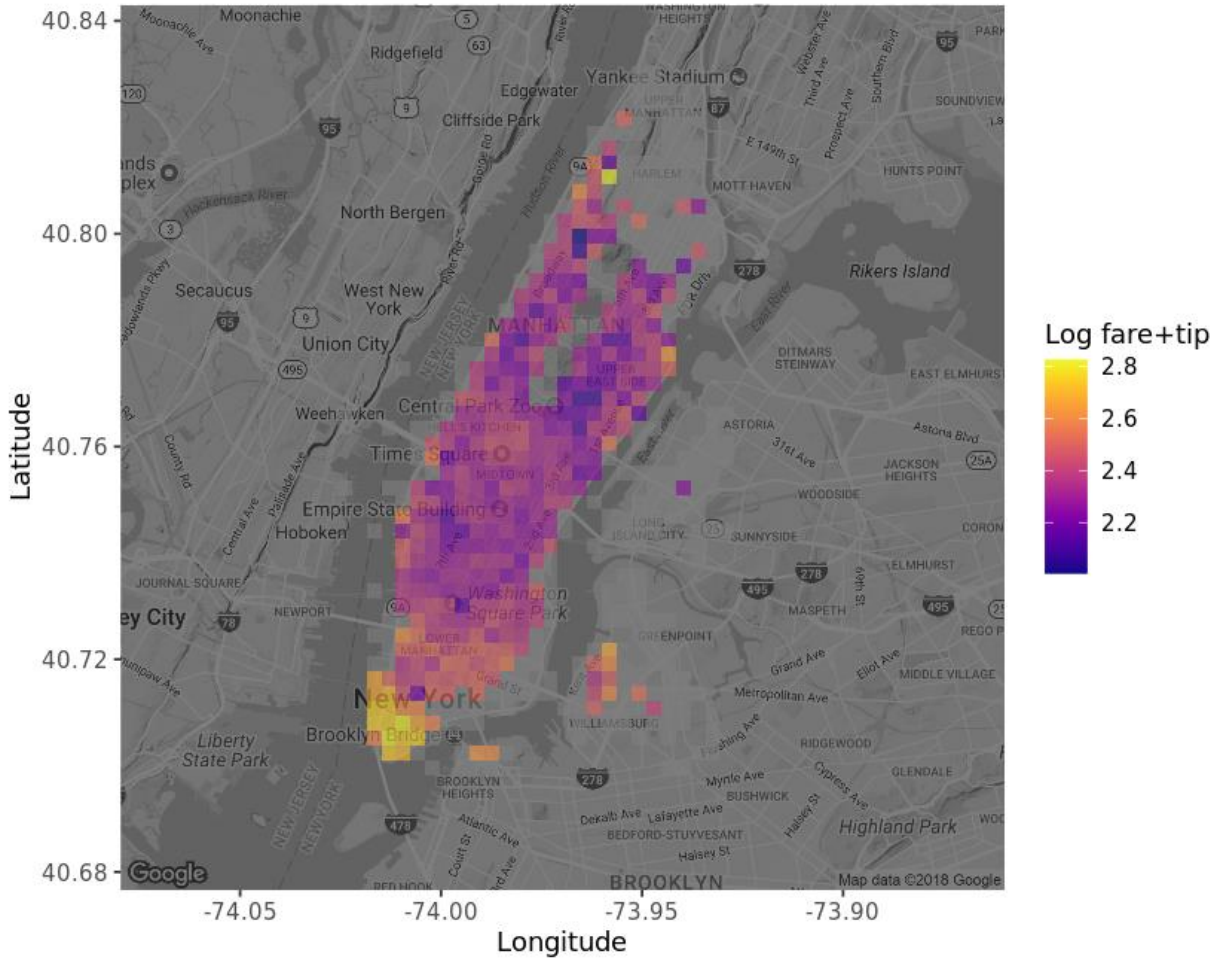
In [16]:
```
# Extracting the prediction from forest_fit
taxi$pred_total <- fitted_forest$predicted

# Plotting the predicted mean trip prices from according to the random forest
ggmap(manhattan, darken=0.5) +
    scale_fill_viridis(option = 'plasma') +
    stat_summary_2d(data=taxi, aes(x = long, y = lat, z = pred_total),
                    fun = mean, alpha = 0.6, bins = 60) +
    labs(x = 'Longitude', y = 'Latitude', fill = 'Log fare+tip')
```



10. Plotting the actual fare

Looking at the map with the predicted fares we see that fares in downtown Manhattan are predicted to be high, while midtown is lower. This map only shows the prediction as a function of lat and long, but we could also plot the predictions over time, or a combination of time and space, but we'll leave that for another time. For now, let's compare the map with the predicted fares with a new map showing the mean fares according to the data.

In [17]:
```
# Function that returns the mean *if* there are 15 or more datapoints
mean_if_enough_data <- function(x) {
    ifelse( length(x) >= 15, mean(x), NA)
}
# Plotting the mean trip prices from the data
ggmap(manhattan, darken=0.5) +
    stat_summary_2d(data=taxi, aes(x = long, y = lat, z = total),
                    fun = mean_if_enough_data,
                    alpha = 0.6, bins = 60) +
  scale_fill_viridis(option = 'plasma') +
  labs(x = 'Longitude', y = 'Latitude', fill = 'Log fare+tip')
```


Latitude
40.84
40.80
40.76
40.72
40.68
-74.05
-74.00
-73.95
-73.90
Longitude
Log fare+tip
2.8
2.6
2.4
2.2
Moonachie
Ridgefield
Edgewater
Yankee Stadium
Cliffside Park
North Bergen
Secaucus
West New York
Union City
Weehawken
Hoboken
Rikers Island
MANHATTAN
Central Park Zoo
Times Square
Empire State Building
Washington Square Park
New York
Brooklyn Bridge
Liberty State Park
BROOKLYN
Highland Park
Google
Map data ©2018 Google