# Steam Gaming Challenge

Steven Jordan

February 2020

# Background

Steam is one of the largest gaming networks in the world with over 100 million active gamers. The Steam dataset covers 109 million user accounts, 196 million friendships, 3 million groups, 384 million owned games, and a collective 1 million years of playtime. The details of each dataset can be found at https://steam.internet.byu.edu/

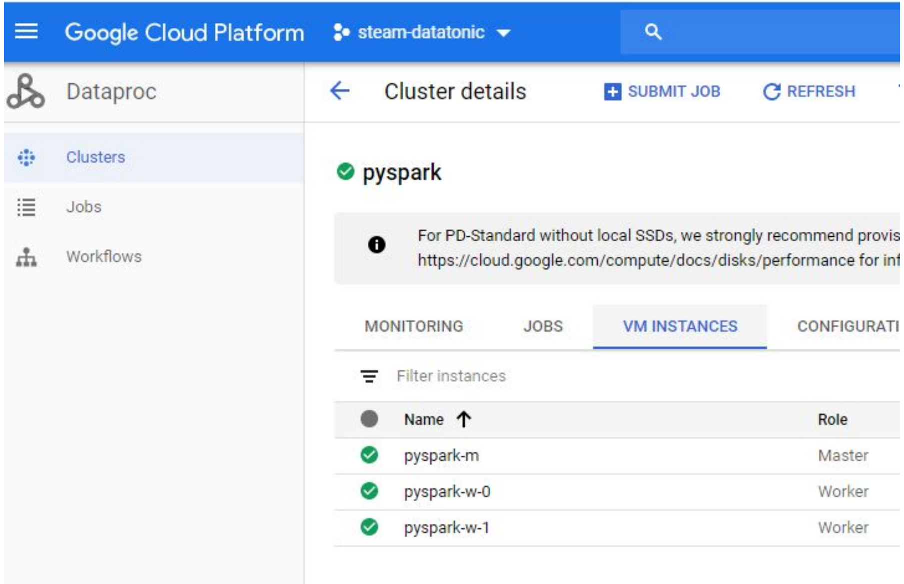
**This presentation is best viewed alongside my code. For Parts 1 and 2, I used the large data sets. For Part 3, I used the small data sets.**

# Exercise 1:
# Data Engineering

# Part 1: Data Engineering

1. **Install and run PySpark**

I decided to complete this exercise using DataProc on GCP. I set up Jupyter notebook for a cloud deployment of PySpark.



# Part 1: Data Engineering

## 2. Load .csv for Player_Summaries, Game_Publishers, Game_Genres, Game_Developers, Games_1 into PySpark dataframes

```
# Read the Player Summary csv files and combine into a single dataframe
path = "gs://dataproc-d5da8056-80df-436a-8ab5-db077106cb06-europe-west6/notebooks/"
player_summaries0_df = spark.read.csv(path + "Player_Summaries-000000000000.csv", header=True)
player_summaries1_df = spark.read.csv(path + "Player_Summaries-000000000001.csv", header=True)
player_summaries2_df = spark.read.csv(path + "Player_Summaries-000000000002.csv", header=True)
player_summaries3_df = spark.read.csv(path + "Player_Summaries-000000000003.csv", header=True)
player_summaries4_df = spark.read.csv(path + "Player_Summaries-000000000004.csv", header=True)
player_summaries5_df = spark.read.csv(path + "Player_Summaries-000000000005.csv", header=True)
player_summaries_df = player_summaries0_df.union(player_summaries1_df.union(player_summaries2_df.union(player_summaries3_c

# Read the Games_* csv files and combine the Games_1 files into single dataframes
games_publishers_df = spark.read.csv(path + "Games_Publishers.csv", header=True)
games_developers_df = spark.read.csv(path + "Games_Developers.csv", header=True)
games_genres_df = spark.read.csv(path + "Games_Genres.csv", header=True)

games_10_df = spark.read.csv(path + "Games_1-000000000000.csv", header=True)
games_11_df = spark.read.csv(path + "Games_1-000000000001.csv", header=True)
games_1_df = games_10_df.union(games_11_df)

## The instructions did not state to load the Games_2 files - but the following instruction says to include all Games_,
## so I assume it was ommitted by typo
games_20_df = spark.read.csv(path + "Games_2-000000000000.csv", header=True)
games_21_df = spark.read.csv(path + "Games_2-000000000001.csv", header=True)
games_22_df = spark.read.csv(path + "Games_2-000000000002.csv", header=True)
games_2_df = games_20_df.union(games_21_df.union(games_22_df))

games_full_df = games_1_df.union(games_2_df)
```

# Part 1: Data Engineering

**2. Load .csv for Player_Summaries, Games_Publishers, Games_Genres, Games_Developers, Games_1 into PySpark dataframes**

This was easily completed by moving the files into the GCP bucket and reading with PySpark.

I used the union() method to combine the csv files that were separated into batches.

```
# Read the Player Summary csv files and combine into a single dataframe
path = "gs://dataproc-d5da8056-80df-436a-8ab5-db077106cb06-europe-west6/notebooks/"
player_summaries0_df = spark.read.csv(path + "Player_Summaries-000000000000.csv", header=True)
player_summaries1_df = spark.read.csv(path + "Player_Summaries-000000000001.csv", header=True)
player_summaries2_df = spark.read.csv(path + "Player_Summaries-000000000002.csv", header=True)
player_summaries3_df = spark.read.csv(path + "Player_Summaries-000000000003.csv", header=True)
player_summaries4_df = spark.read.csv(path + "Player_Summaries-000000000004.csv", header=True)
player_summaries5_df = spark.read.csv(path + "Player_Summaries-000000000005.csv", header=True)
player_summaries_df = player_summaries0_df.union(player_summaries1_df.union(player_summaries2_df

# Read the Games_* csv files and combine the Games_1 files into single dataframes
games_publishers_df = spark.read.csv(path + "Games_Publishers.csv", header=True)
games_developers_df = spark.read.csv(path + "Games_Developers.csv", header=True)
games_genres_df = spark.read.csv(path + "Games_Genres.csv", header=True)

games_10_df = spark.read.csv(path + "Games_1-000000000000.csv", header=True)
games_11_df = spark.read.csv(path + "Games_1-000000000001.csv", header=True)
games_1_df = games_10_df.union(games_11_df)

## The instructions did not state to load the Games_2 files - but the following instruction says
## so I assume it was ommitted by typo
games_20_df = spark.read.csv(path + "Games_2-000000000000.csv", header=True)
games_21_df = spark.read.csv(path + "Games_2-000000000001.csv", header=True)
games_22_df = spark.read.csv(path + "Games_2-000000000002.csv", header=True)
games_2_df = games_20_df.union(games_21_df.union(games_22_df))

games_full_df = games_1_df.union(games_2_df)
```

# Part 1: Data Engineering

## 3. Join all ‘Games_’ tables into one dataframe

Since no other instructions were provided, I used ‘full’ joins so that no data was lost.

```
In [3]: games_join = games_full_df.join(games_publishers_df, on = ['appid'], how = 'full').join(games_developers_df, on = ['appid
```

```
In [4]: games_join.show(10)
```

```
+------+-----------------+---------------+-----------------+--------------------+---------------+--------------+--------+
| appid|          steamid|playtime_2weeks|playtime_forever|       dateretrieved|      Publisher|     Developer|   Genre|
+------+-----------------+---------------+-----------------+--------------------+---------------+--------------+--------+
|108231|             null|           null|             null|                null| Days of Wonder|Days of Wonder|   Indie|
|108231|             null|           null|             null|                null| Days of Wonder|Days of Wonder|  Casual|
|108231|             null|           null|             null|                null| Days of Wonder|Days of Wonder|Strategy|
|108800|76561197973718054|           null|             1348|2013-05-14 07:52:...|Electronic Arts|Crytek Studios|  Action|
|108800|76561198077027953|            213|              969|2013-09-27 11:45:...|Electronic Arts|Crytek Studios|  Action|
|108800|76561197970336122|           null|              998|2013-05-11 20:12:...|Electronic Arts|Crytek Studios|  Action|
|108800|76561198050351845|            102|              102|2013-08-28 07:52:...|Electronic Arts|Crytek Studios|  Action|
|108800|76561197983834231|           null|              942|2013-05-18 15:08:...|Electronic Arts|Crytek Studios|  Action|
|108800|76561198066712668|           null|              327|2013-10-01 08:01:...|Electronic Arts|Crytek Studios|  Action|
|108800|76561197993364277|           null|              300|2013-05-29 01:44:...|Electronic Arts|Crytek Studios|  Action|
+------+-----------------+---------------+-----------------+--------------------+---------------+--------------+--------+
only showing top 10 rows
```

(This game with no plays is Ticket to Ride: Switzerland! It is actually quite fun - I have it on mobile.)

# Part 1: Data Engineering

## 4. Count the number of games per ‘publisher’ and per ‘genre’

Leaving in the ‘null’ values for Publishers and Genres was a conscious choice. It could be removed using the .isNotNull() method.

```
games_publishers_df.groupBy('Publisher').count().or
```

| Publisher | count |
|---|---|
| null | 2627 |
| Ubisoft | 384 |
| SEGA | 349 |
| Dovetail Games - Trains | 279 |
| Paradox Interactive | 246 |
| Disney Interactive | 226 |
| Feral Interactive (Mac) | 221 |
| Activision | 221 |
| Degica | 190 |
| Nordic Games | 164 |
| Square Enix | 153 |
| KISS ltd | 145 |
| Feral Interactive (Linux) | 143 |
| Wizards of the Coast LLC | 135 |
| Strategy First | 134 |
| 2K Games | 120 |
| Capcom | 115 |
| Warner Bros. Interactive Entertainment | 108 |
| Kalypso Media Digital | 104 |
| Deep Silver | 98 |

only showing top 20 rows

```
games_genres_df.groupBy('Genre').count().orderBy('c
```

| Genre | count |
|---|---|
| Indie | 7982 |
| Action | 7126 |
| Adventure | 4517 |
| Strategy | 3953 |
| Casual | 3939 |
| Simulation | 3870 |
| RPG | 3147 |
| Free to Play | 1172 |
| Early Access | 901 |
| Massively Multiplayer | 748 |
| Racing | 619 |
| Sports | 604 |
| Design & Illustration | 240 |
| Utilities | 207 |
| Web Publishing | 146 |
| Animation & Modeling | 112 |
| Audio Production | 112 |
| Software Training | 82 |
| Education | 73 |
| Video Production | 69 |

only showing top 20 rows

# Part 1: Data Engineering

**5. Find the day and hour when most new accounts were created (based on Player_Summaries table) e.g. 8pm on 14th August 2005.**

To do this, I combined a groupBy() and the date_format function on the timestamp, but only included the year, month, day, and hour (so any accounts created before the next hour were rounded down). The time with the most accounts created was: **2nd of March 2013 at 10 AM.**

```
player_summaries_df.groupBy(date_format('timecreated', 'yyyy-MM-dd HH').alias('hour')).count().orderBy('count', ascending = 0).sh
```

```
+-------------+------+
|         hour| count|
+-------------+------+
|         null|252717|
|2013-03-02 10|  1782|
|2012-12-25 10|  1448|
|2012-12-25 08|  1355|
|2012-12-25 11|  1159|
+-------------+------+
only showing top 5 rows
```

# Part 2:
# Analytics

# Part 2: Analytics

**Business Case Instructions:** “Your client is a mental health expert from an NGO who is interested in understanding more about gaming and the potentially addictive effect it can have on some individuals. You are meeting the client in a few days and they would like you to extract and present insights from the Steam dataset to help them in their research.”

**My Approach:** I would ideally obtain my client’s criteria that they use to define an addiction - however, it was not provided, so I researched externally. There is no defined consensus (and the World Health Organization omits it purposefully), so I settled on using 50 hours of video gaming per week as an “addiction-level” of gaming for the purpose of this business case. I brainstormed questions the client might want answered, which are presented in the next slides.

For this exercise, I continued using PySpark for analysis and then wrote new dataframes into .csv files which were loaded into Tableau for visualizations.

# Part 2: Analytics

## Question 1 - How many and what percentage of active Steam users have an “addiction-level” of playtime?

I joined the App_ID dataframe with the previously created Games_ dataframe from Exercise 1. This dataframe was used as the base to answer several of the following questions.

From here, I counted the number of “active” users (accounts with any playtime in the last two weeks) and counted the number of users that are “addicts” (played more than 6000 minutes in the last two weeks).

**Incredibly, 135,493 players and 24.7% of all active Steam users have an “addiction-level” of playtime.**

```
addicts_df = games_played.groupBy('steamid').s
addicts_df.show(5)
```

```
+-----------------+--------------------+
|          steamid|sum(playtime_2weeks)|
+-----------------+--------------------+
|76561197979880077|               19835|
|76561198017077515|               45219|
|76561197996880035|               29113|
|76561198043166176|               10340|
|76561197971697538|               18111|
+-----------------+--------------------+
only showing top 5 rows
```

```
# Calculate addiction rate
addiction_rate = addicts_df.count()/active_players_df.count()
print(addiction_rate)
```

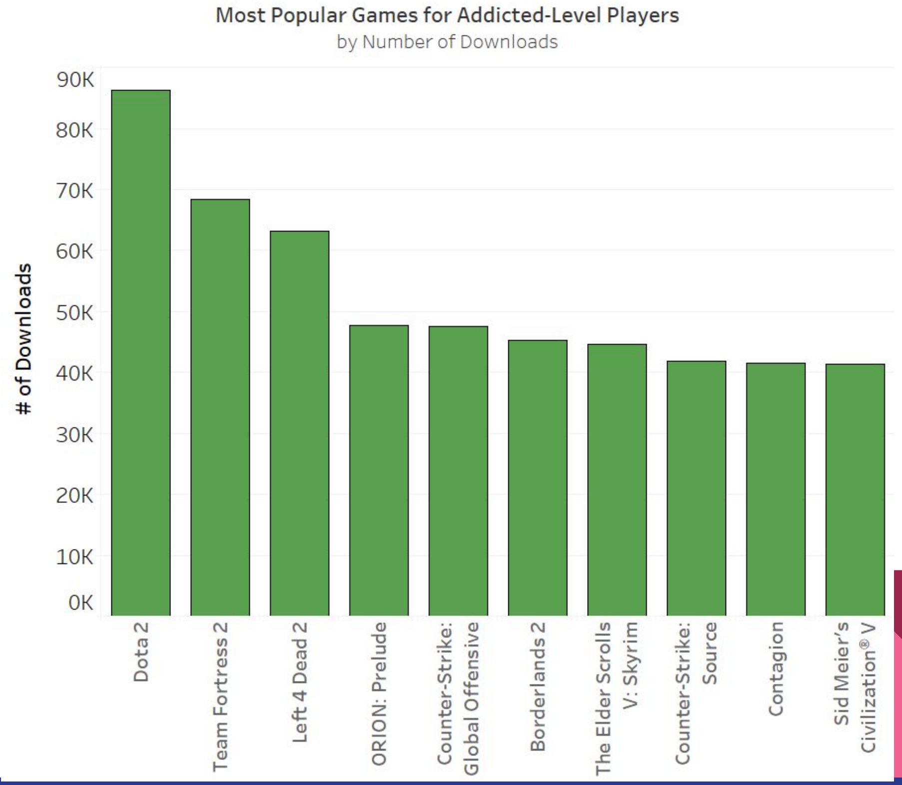
```
0.24729512684796495
```

# Part 2: Analytics

**Question 2 - What are the most popular games/apps downloaded for the 135k addicted-level players?**

| Game | Downloads |
|---|---:|
| Dota 2 | 86,214 |
| Team Fortress 2 | 68,351 |
| Left 4 Dead 2 | 63,188 |
| ORION: Prelude | 47,755 |
| Counter-Strike: Global Offensive | 47,541 |
| Borderlands 2 | 45,277 |
| The Elder Scrolls V: Skyrim | 44,572 |
| Counter-Strike: Source | 41,766 |
| Contagion | 41,597 |
| Sid Meier's Civilization V | 41,345 |

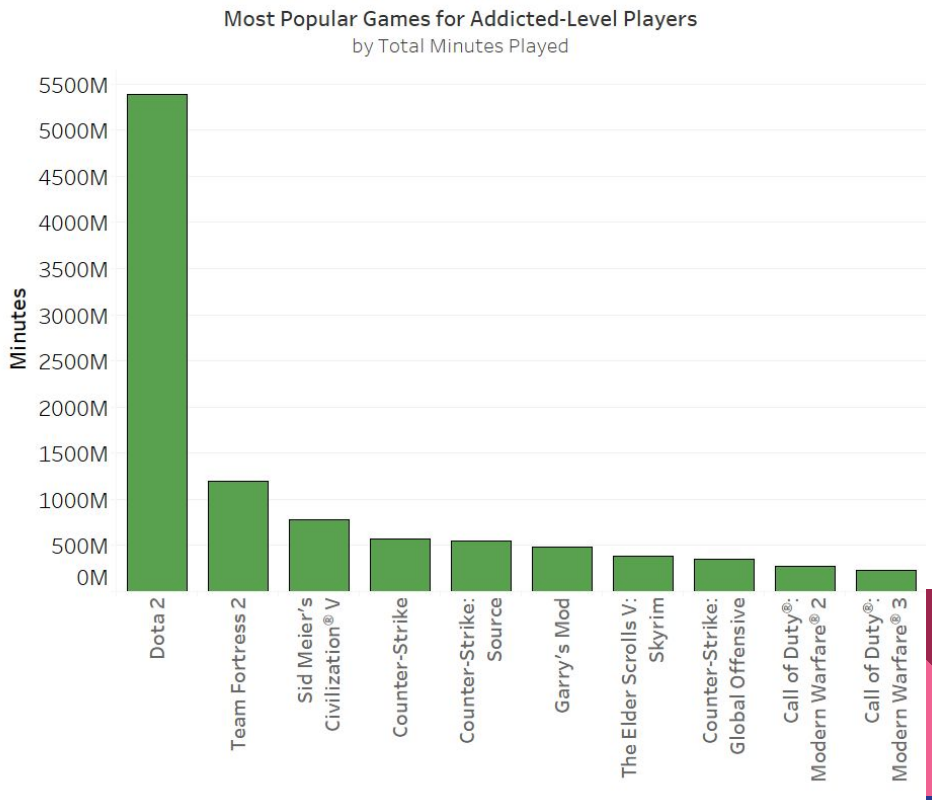
**63.6% of addicted-level players have downloaded Dota 2**



# Part 2: Analytics

**Question 3 - What are the most popular games/apps played (in minutes) for the 135k addicted-level players?**

| Game | Minutes |
|---|---:|
| Dota 2 | 5,391,727,528 |
| Team Fortress 2 | 1,197,885,641 |
| Sid Meier's Civilization V | 777,257,322 |
| Counter-Strike | 569,676,280 |
| Counter-Strike: Source | 549,369,386 |
| Garry's Mod | 483,423,927 |
| The Elder Scrolls V: Skyrim | 386,992,154 |
| Counter-Strike: Global Offensive | 354,250,698 |
| Call of Duty: Modern Warfare 2 | 227,854,837 |
| Call of Duty: Modern Warfare 3 | 226,900,035 |

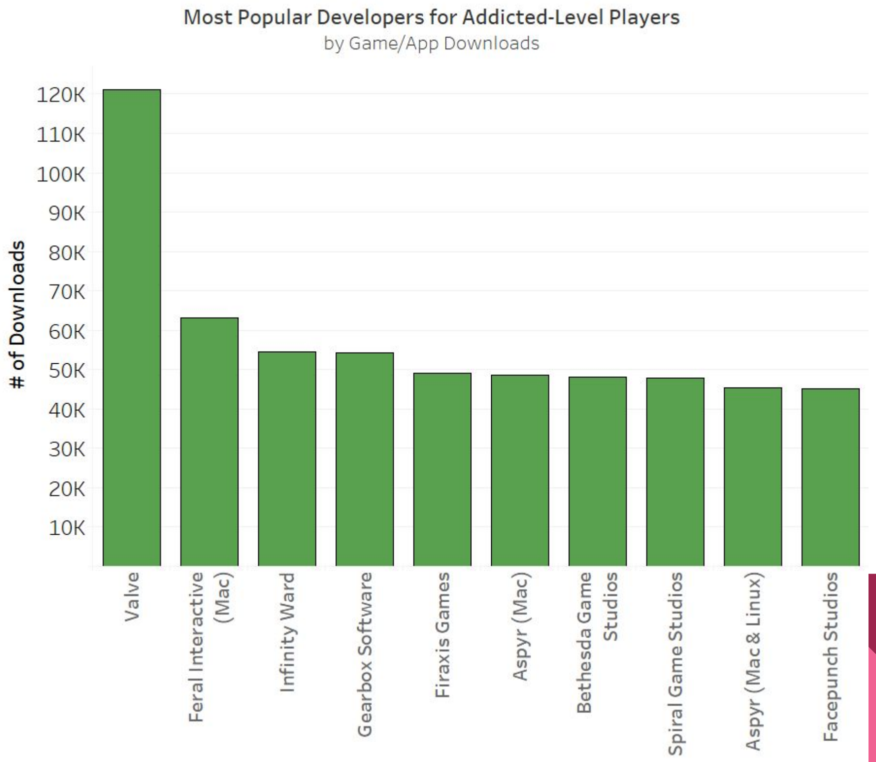



**The addicted-level players have played over a combined 10,258 years of Dota 2**

# Part 2: Analytics

## Question 4 - Which developers are the most popular for the addicted-level players?

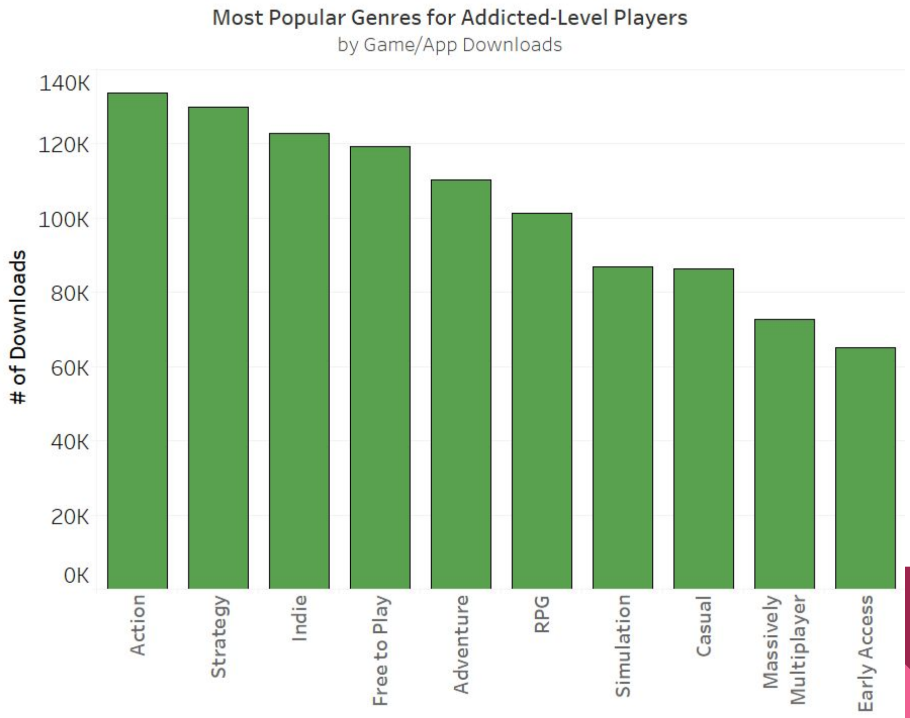
| Game | Downloads |
|---|---:|
| Valve | 121,205 |
| Feral Interactive (Mac) | 63,191 |
| Infinity Ward | 54,492 |
| Gearbox Software | 54,291 |
| Firaxis Games | 49,169 |
| Aspyr (Mac) | 48,560 |
| Bethesda Game Studios | 48,014 |
| Spiral Game Studios | 47,755 |
| Aspyr (Mac & Linux) | 45,277 |
| Facepunch Studios | 45,147 |



# Part 2: Analytics

## Question 5 - Which genres are the most popular for the addicted-level players?

| Game | Downloads |
|---|---:|
| Action | 133,740 |
| Strategy | 129,858 |
| Indie | 122,848 |
| Free to Play | 119,091 |
| Adventure | 110,148 |
| RPG | 101,159 |
| Simulation | 86,835 |
| Casual | 86,315 |
| Massively Multiplayer | 72,717 |
| Early Access | 65,014 |

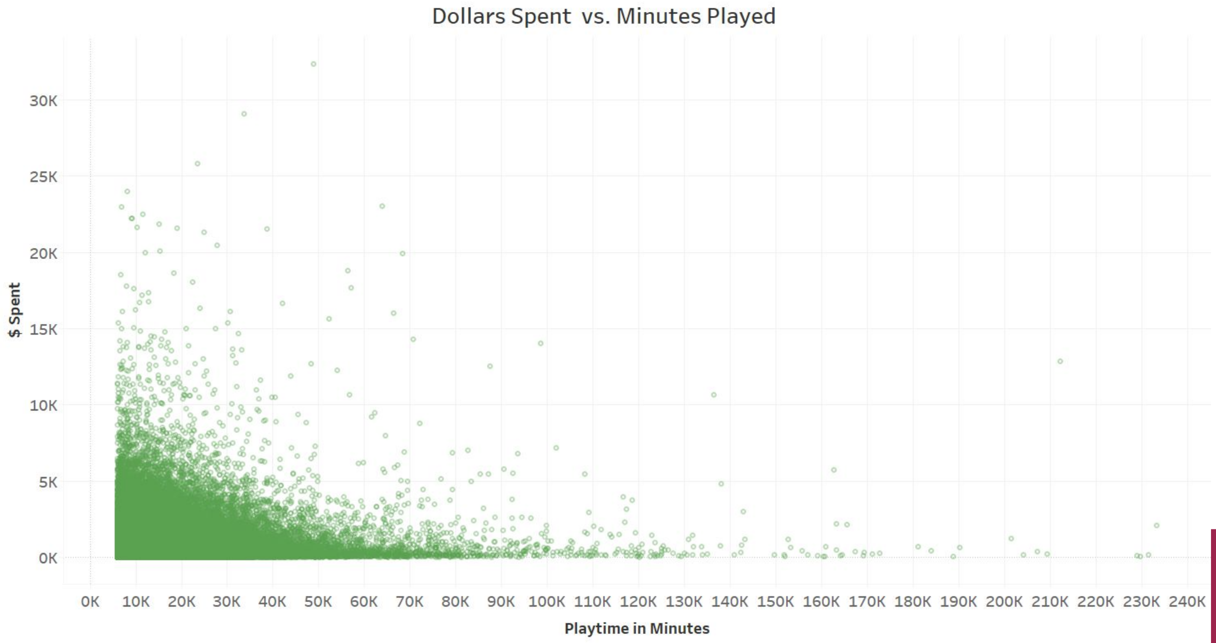


# Part 2: Analytics

## Question 6 - What’s the relationship between the amount of time played and dollars spent?

While several addicted-level users have spent over $25,000 on the Steam platform, the average addicted-level user spent just $692.

As one can see from the scatter plot, there isn’t an obvious relationship between minutes played and dollars spent for addicted-level users



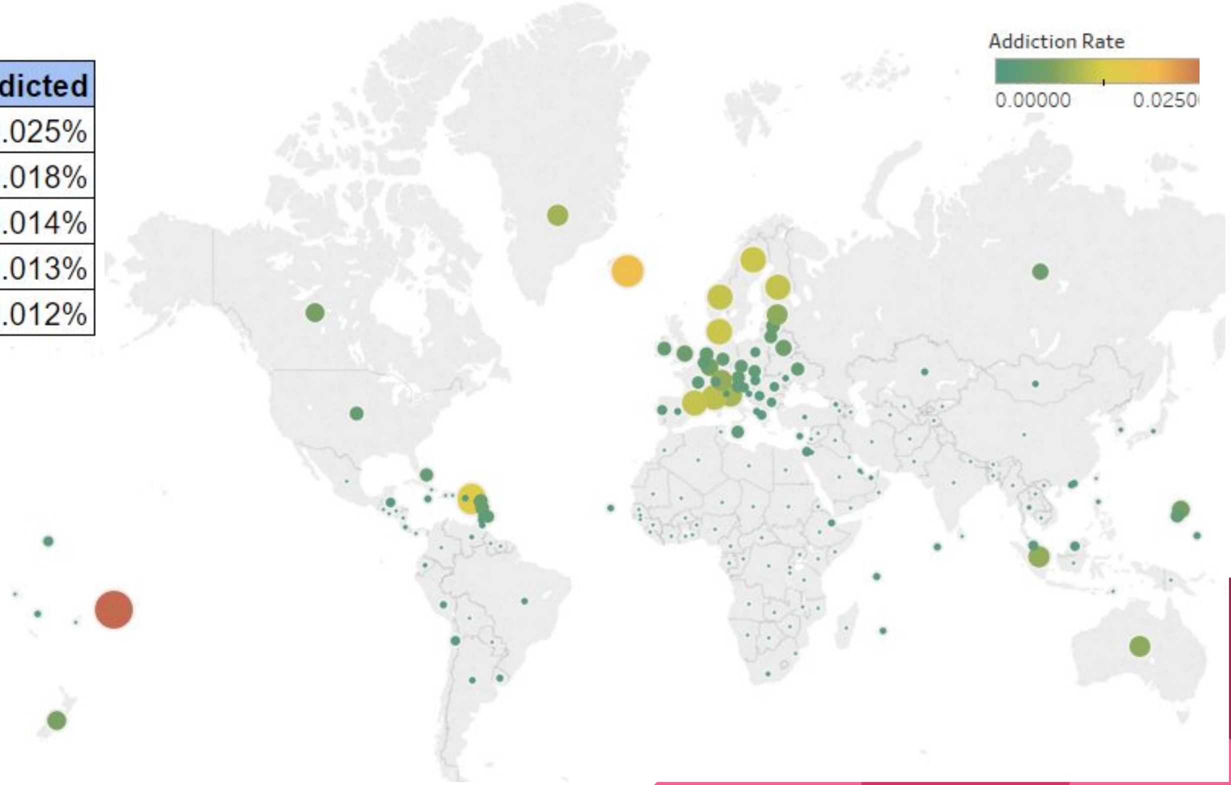
# Part 2: Analytics

**Question 7 - What countries have the highest rates of addicted-level players per capita?**

| Country/Territory | % Addicted |
|---|---|
| American Samoa | 0.025% |
| Iceland | 0.018% |
| US Virgin Islands | 0.014% |
| British Virgin Islands | 0.013% |
| Denmark | 0.012% |

The highest rates of addiction per capita appear to be on islands, in Scandinavia, and in micro-states.

External population data was obtained here.



# Exercise 3: Advanced

# Part 3: Recommender System

**Instructions:** Be creative and impress us with something novel, do an analytics deep-dive or show off some machine learning. For instance, use supervised methods to make predictions or recommendations, or use network analysis on the friends table.

**My Approach:** For this exercise, I built a collaborative-based recommender system for Steam users. Because there are no user ratings in this data set, I used a user’s playtime on any particular game as a proxy for the rating (people spend more time playing games they like).

I began the project in PySpark on GCP to do some dataframe manipulations and obtain the “base” dataframe for the recommender system. I then wrote the dataframe to a .csv file and constructed the recommender system using Pandas using my own machine as it computes more quickly.

# Part 3: Recommender System

I used PySpark on GCP to adjust the dataframe obtained from games_1.csv so that each row contains a steamID and the total playtime of every game/app (as indicated by the appid). I then read it onto my own machine using Pandas.

| | steamid | 10 | 100 | 10000 | 1002 | 10040 | 100400 | 100410 | 10080 | 10090 | ... | 99400 | 99410 | 9960 | 9970 | 997 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 76561197973784324 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |
| 1 | 76561198027864348 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |
| 2 | 76561197972510274 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |
| 3 | 76561198001205398 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |
| 4 | 76561197972024750 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |
| 5 | 76561198077212088 | 1431 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |

# Part 3: Recommender System

The recommender system currently is designed to accept one SteamID as input. In the code, one can modify the selected SteamID, or use the commented code to create a user input.

The recommender system will create recommendations for this user.

```
# One can replace the provided steamid with any other steamid,
# or could be obtained through user input

#userid = int(input("Please enter your Steam ID: "))
userid = 76561198077212088
```

# Part 3: Recommender System

The recommender system currently is designed to accept one SteamID as input. In the code, one can modify the selected SteamID, or use the commented code to create a user input.

The recommender system will create recommendations for this user.

```
# One can replace the provided steamid with any other steamid,
# or could be obtained through user input

#userid = int(input("Please enter your Steam ID: "))
userid = 76561198077212088
```

# Part 3: Recommender System

If I were to develop this further, I would enable the ability to create recommendations for batches of users (marketing events, sales, etc.).

Using the playtime of each game as a proxy for a rating, I included code that would find centroid of a group of users' gameplay. However, as it is currently built, this code just returns the same data as is input - the playtime of each game of the one user.

```
# The below code would find the centroid for a group of users. However, since there is only a
# single SteamID used for this demonstration, the centroid is identical to the single user record

user_data = [df.loc[userid]]
kmeans = KMeans(n_clusters=1).fit(user_data)
user_centroid = kmeans.cluster_centers_[0]
```

# Part 3: Recommender System

The recommender calculates and ranks a similarity score for every steamID in the dataframe, relative to the user's steamID..

```
sim_list = []
for user in data.iterrows():
    sim = skmp.cosine_similarity([user_centroid], [user[1]])
    sim_list.append(sim[0][0])
```

```
# Zips the similarity scores to their associated SteamIDs, sorting them by similarity
tup = zip(sim_list, sim_users.steamid)
tups_sorted = sorted(tup, key=lambda tup:tup[0], reverse = True)

# Prints the top 10 most similar Steam IDs and their similarity scores
tups_sorted[1:11]
```

```
[(0.9658393953021863, 76561197973970142),
 (0.9657963690076256, 76561198065204329),
 (0.965660212329164, 76561197989059688),
 (0.9652960219623384, 76561198029515752),
 (0.9651958231014126, 76561197972540006),
 (0.9650707031043491, 76561198001419384)
```

# Part 3: Recommender System

The recommender system finds the 100 steamIDs with the greatest similarity scores for the basis of the recommendations. It then sums of the playtime of every one of those users for every game they have played - the higher the total, playtime of a game, the higher its ranking as a recommendation.

```
# Pulls out the most similar user SteamIDs. I am using 100 for the recommender system, but this
# can be adjusted based on testing

most_similar_users = []
for y in tups_sorted[1:101]:
    most_similar_users.append(y[1])
```

```
# Iterates through the similar users, summing up their total playtime for each game/app
total_playtime = df.loc[userid]
for u in most_similar_users:
    total_playtime += df.loc[u]

# Sorts the potentially recommended games by total playtime, resetting it as Pandas dataframe
total_playtime_sorted = total_playtime.sort_values(ascending=False)
sim_df = pd.DataFrame(total_playtime_sorted)
sim_df.head()
```

# Part 3: Recommender System

After iterating through the top recommendations, the system prints only those which the input steamID has had not played.

```
For User with Steam ID: 76561198077212088
...we recommend the following games or applications:

Counter-Strike: Global Offensive          appid: 730
Call of Duty®: Modern Warfare® 2          appid: 10190
Counter-Strike: Source                    appid: 240
Call of Duty®: Modern Warfare® 3          appid: 42690
Counter-Strike: Condition Zero            appid: 80
Left 4 Dead 2                             appid: 550
Total War: SHOGUN 2                       appid: 34330
Killing Floor                             appid: 1250
Call of Duty®: Modern Warfare® 3          appid: 42680
PAYDAY™ The Heist                         appid: 24240
```