**Android Application Market on Google Play store**

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 project, we have done a comprehensive analysis of the Android app market by comparing over ten thousand apps in Google Play across different categories. We have carried data analysis to devise strategies to drive growth and retention.

**AIM OF THE PROJECT:**

Our main aim is to help the app developers to look at the valuable insights to understand user satisfaction briefly with smart reviews. We mainly focussed to find the correlation between two or more variables from the data set to determine success factor of an application. Finding the success factor of the applications will help in discovering worldwide trends, outperform competitors with consolidated competitor insights and market leader strategies. Evaluating the success factor leads to increase the app visibility in app stores to improve conversion to installs.

**SOLUTION STEPS:**

At first, we stored the data in Neo4j by applying data modelling concepts. Later, we explored the data using PySpark to carry the analysis. Steps of implementation are: data cleaning, exploring the application categories, distribution of application ratings, size and price of the application, co-relation between application category and price, popularity of paid applications and free applications, and finally the sentiment analysis of the user reviews.

**DATA SET:**

Data set for this project is from the Kaggle website.

The data files are as follows:

**googleplaystore\_user\_reviews.csv**: contains more than 20 reviews for each app (64k+ data). The text in each review has been pre-processed and attributed with three new features: Sentiment (Positive, Negative or Neutral), Sentiment Polarity and Sentiment Subjectivity.

**googleplaystore.csv:** contains all the details of the applications on Google Play (10k+ data). There are 13 features that describe a given app.

**TOOLS FOR ANYALYSIS:**

* We have chosen NEO4J as NoSQL technology since it is a NoSQL database based on graph theory. It efficiently stores, handle, and query highly connected data in your data model. With a powerful and flexible data model, we can represent the real-world, variably structured information without a loss of richness. As the reviews of the application increases the speed of neo4j doesn't decrease and can be used in real time analysis.
* We have used PySpark to perform a Sentiment analysis of user reviews to determine the relation between different variables.

**Graph Database-NEO4J:**

The Neo4j database is a high-performance Nosql graph database. As a graph database, Neo4j has the following advantages:

* Faster database operations
* More intuitive data
* More flexible
* The speed of database operations does not decrease significantly as the database grows.
* Self-contained query language (called Cypher)
* The structure of the entity relationship is very natural and fits the intuitive feeling of human beings.

## Create Neo4j PLAYSTORE Database

If we take a close look at the CSV, below are the columns used in GooglePlayStore file-

App, Category, Rating, Reviews, Size, Installs, Type, Price, Content Rating, Genres, Last Updated, Current Ver and Android Ver.

As part of Data modelling, we have created three types of nodes and two relationships -

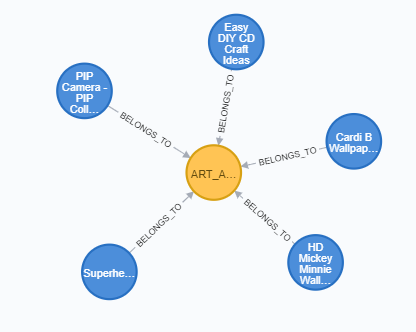
NODES => 1) APPLICATION II) CATEGORY III) APPLICATION\_USER\_REVIEW

RELATIONSHIPS => BELONGS\_TO, REVIEWED

1) For creation of node APPLICATION, we have used below CYPHER query-

|  |
| --- |
| **LOAD** CSV **WITH** HEADERS **FROM** '[file:///desktop-csv-import/googleplaystore.csv](file:///C:\desktop-csv-import\googleplaystore.csv)'  **AS** row  MERGE (cat:Category {**name**: row.Category})  **CREATE** (a:APPLICATION {Appid: row.Id, App: row.App,  Rating: toFloat(row.Rating), Reviews: toInteger(row.Reviews),  **Size**: row.**Size**, Installs: row.Installs, Type: row.Type, Price: row.Price, ContentRating: row.`Content Rating`})  **CREATE** (a)-[: BELONGS\_TO]->(cat) |

The above CYPHER query creates the “Application” node, “Category” node as well as the relationship “BELONGS\_TO” which connects the Application node with Category node.

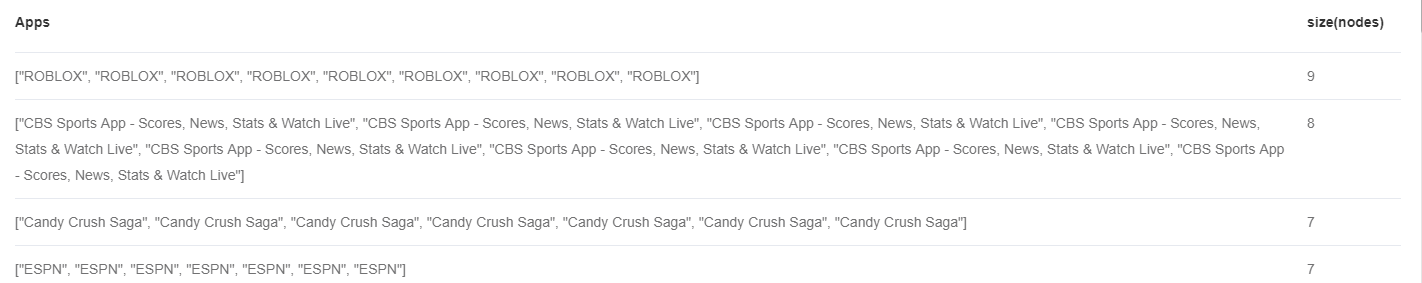


2) As second step, we have tried to perform some data cleaning steps. Below is the CYPHER query used for the identification of the duplicate rows containing same app names but different count for reviews.

A) To find duplicate rows-

|  |
| --- |
| **MATCH (a:APPLICATION)**  **WITH a.App as App, collect(a) AS nodes**  **WHERE size(nodes) > 1**  **RETURN [ n in nodes | n.App] AS Apps, size(nodes)**  **ORDER BY size(nodes) DESC**  **LIMIT 100** |

Result-



B) We observed that it's just the review count which is varying. Rest of the values for the rows sharing same app name are same. Hence, we decided to consider the rows with highest review count and delete the rest of the of duplicate rows.

|  |
| --- |
| **MATCH (n:APPLICATION) WHERE n.App = 'ROBLOX' RETURN n LIMIT 25**  **MATCH (n:APPLICATION) WHERE n.App = '8 Ball Pool' RETURN n LIMIT 25** |

C) We queried the rows from highest reviews to lowest for the same app.

|  |
| --- |
| MATCH (a:APPLICATION)  **WITH** a  **ORDER** **BY** a.App, a.Reviews **DESC**  **. . .**  **ORDER** **BY** **size**(nodes) **DESC**  LIMIT 100 |

D) Then, we executed delete query for the same-

|  |
| --- |
| MATCH (a:APPLICATION)  **WITH** a  **ORDER** **BY** a.App, a.Reviews **DESC**  **WITH** a.App **as** App, collect(a) **AS** nodes  **WHERE** **size**(nodes) > 1  UNWIND nodes[1..] **AS** n  DETACH **DELETE** n |

3) Created a constraint to make sure that even in future the App name is restricted to be unique. When we create a constraint, Neo4j will automatically create an index. Cypher will use that index for lookups just like other indexes.

|  |
| --- |
| **CREATE** **CONSTRAINT** **ON** (a:APPLICATION) ASSERT a.App **IS** **UNIQUE** |

4) To create APPLICATION\_USER\_REVIEW node, below CYPHER query is used-

|  |
| --- |
| **LOAD** CSV **WITH** HEADERS **FROM** '[file:///desktop-csv-import/googleplaystore\_user\_reviews.csv](file:///C:\desktop-csv-import\googleplaystore_user_reviews.csv)'  **AS** row  **CREATE** (r:APPLICATION\_USER\_REVIEW {Reviewid: row.Id, App: row.App,  . . . |

5) To delete the reviews with values as nan

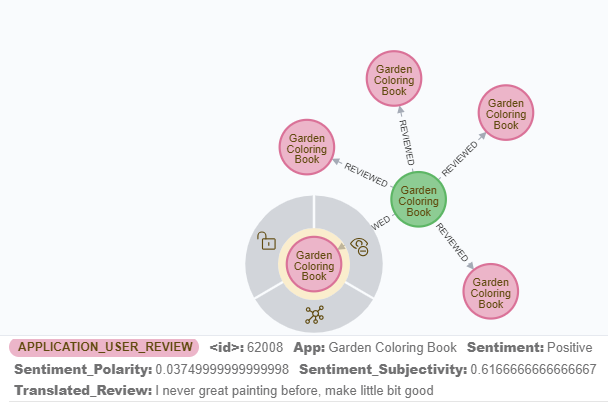
|  |
| --- |
| MATCH (r: APPLICATION\_USER\_REVIEW)  **WHERE** r.Translated\_Review = 'nan'  **DELETE** r |

6) Create Relationship REVIEWED to relate “Application” with “Application\_user\_review”

|  |
| --- |
| MATCH (a:APPLICATION),(r:APPLICATION\_USER\_REVIEW)  **WHERE** a.App = r.App  **CREATE** (a)-[rel:REVIEWED]->(r) |

6) To display the relationship, we have used below query-

|  |
| --- |
| MATCH p=()-[r:REVIEWED]->() **RETURN** p LIMIT 5 |



Below is the final graph, to display all 3 nodes and 2 relationships is-



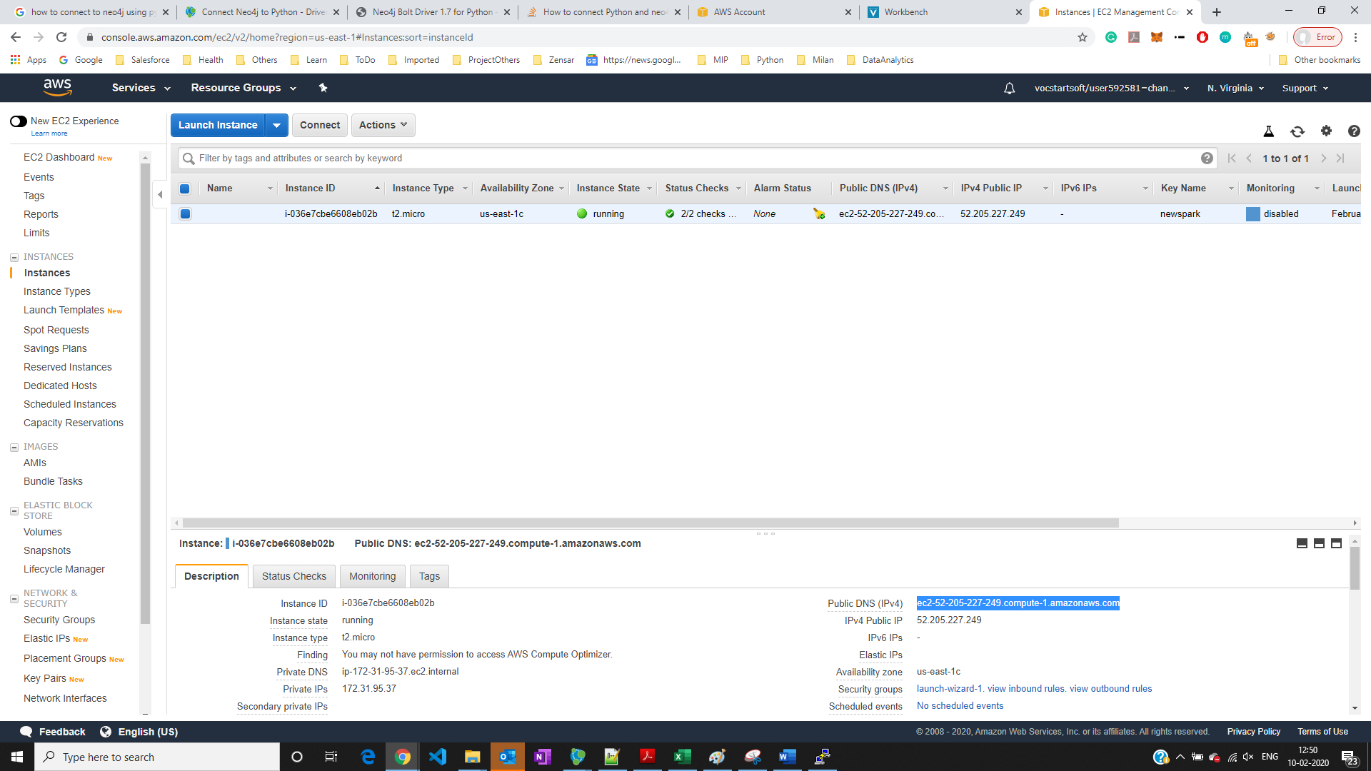
**PYSPARK**

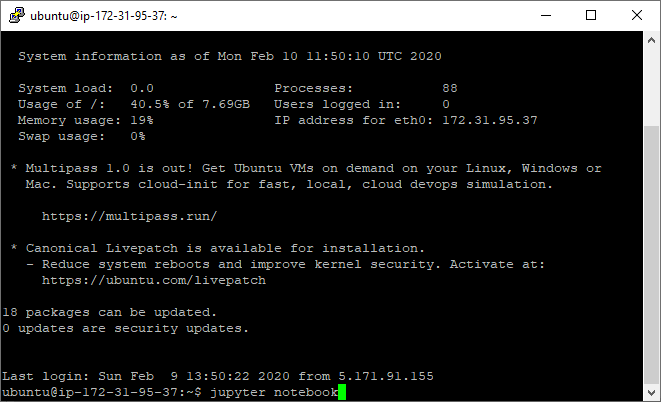
Using PySpark, we have performed data cleaning steps. We have then performed DataType conversions and transformations since transformed data will be utilized to carry the data analysis operations. Below are the steps explained in detail-

PySpark setup:

Amazon EC2,

Jupyter Notebook





<https://ec2-52-205-227-249.compute-1.amazonaws.com:8888/tree?token=4d1277fd5fd46dc4923bc090dec14073e8d8e7ae34e0d1bf>

## **I) IMPORTING LIBRARIES**

|  |
| --- |
| **from** **pyspark** **import** StorageLevel  **from** **pyspark.sql** **import** SparkSession  **from** **pyspark.sql.functions** **import** lit, col, explode, initcap, regexp\_replace, split, concat, substring, to\_date  **from** **pyspark.sql.types** **import** StructType, StructField, StringType, IntegerType, FloatType, DoubleType |

## **II) LOADING DATASET**

Since some of the app names have a double quotation mark, we should indicate to spark.read.load that this character must be escaped. Otherwise, we would have extra columns in the Dataframe.

|  |
| --- |
| spark = SparkSession.builder.appName('PySparkGooglePlayApps').getOrCreate()  apps = spark.read.load('/home/ubuntu/googleplaystore.csv'),  format='csv',  sep = ',',  header='true',  escape='"',  inferSchema='true') |

|  |
| --- |
| apps.select("App").where("App like '%**\"**%'").show(truncate=**False**)  apps.count() |

**III) Transformations & Datatype conversions**

|  |
| --- |
| apps.printSchema() |

root

|-- App: string (nullable = true)

|-- Category: string (nullable = true)

|-- Rating: double (nullable = true)

|-- Reviews: string (nullable = true)

|-- Size: string (nullable = true)

|-- Installs: string (nullable = true)

|-- Type: string (nullable = true)

|-- Price: string (nullable = true)

|-- Content Rating: string (nullable = true)

|-- Genres: string (nullable = true)

|-- Last Updated: string (nullable = true)

|-- Current Ver: string (nullable = true)

|-- Android Ver: string (nullable = true)

Below are the transformations we are going to focus-

|  |
| --- |
| apps = apps \  .withColumnRenamed("Android Ver", "android\_ver") \  .withColumnRenamed("Current Ver", "current\_ver") \  .withColumnRenamed("Last Updated", "last\_updated") \  .withColumn("Rating", col("Rating").cast(DoubleType())) \  .withColumn("Reviews", col("Reviews").cast(IntegerType())) \  .withColumn("Installs", regexp\_replace(col("Installs"), "[^0-9]", "")) \  .withColumn("Installs", col("Installs").cast(IntegerType())) \  .withColumn("Price", regexp\_replace(col("Price"), "[$]", "")) \  .withColumn("Price", col("Price").cast(DoubleType())) \  .withColumn("last\_updated", to\_date('last\_updated', 'MMM d, yyyy'))  *# Fill NULL values with 0*  apps = apps.fillna(0, "price") |

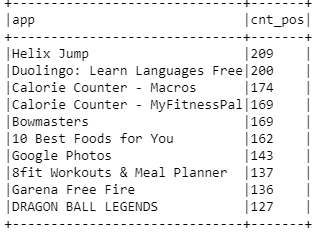
We have repeated same steps for GooglePlayStore\_User\_Reviews as well. After review load, we created temporary view of Review

|  |
| --- |
| reviews.createOrReplaceTempView("reviews") |

Using PySpark, we tried to answer below Business Questions-

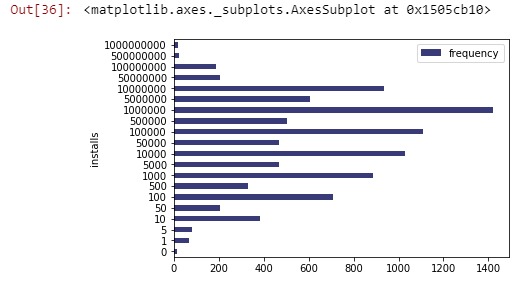
### Top 10 applications regarding the highest number of **positive sentiments**.

|  |
| --- |
| spark.sql("""  SELECT app, count(\*) cnt\_pos  FROM reviews  WHERE sentiment = "Positive"  GROUP BY app  ORDER BY 2 DESC  LIMIT 10  """).show(truncate=**False**) |



### Histogram for the number of installations.

|  |
| --- |
| histogram = spark.sql("""  SELECT NVL(installs, 0) installs  , count(\*) frequency  FROM apps  GROUP BY NVL(installs, 0)  ORDER BY 1  """)  *# Programmatic way to replace NULL with 0* histogram = histogram.fillna(0, "installs")  *# Conversion: Spark DF to Pandas DF*  plotdf = histogram.toPandas()  *# Pandas internal plot library*  plotdf.plot(kind='barh', x='installs', y='frequency', colormap='tab20b') |



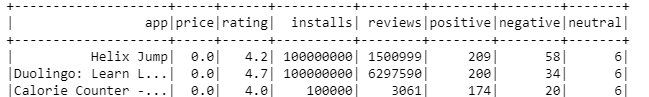
### Trying to find correlation between some of the attributes.

1. ***Correlation between sentiment and rating***

We found that rating is mostly similar in both very positive and negative sentiments. We had expected much worse rating for those with negative sentiment, but the data shows that it's only slightly lower.

**Positive sentiment**

|  |
| --- |
| spark.sql("""  WITH apps\_vw AS  (SELECT app, rating, reviews, installs, price, genres  FROM apps  ),  reviews\_vw AS  (SELECT app,  SUM(CASE sentiment WHEN "Positive" THEN 1 ELSE 0 END) positive,  SUM(CASE sentiment WHEN "Negative" THEN 1 ELSE 0 END) negative,  SUM(CASE sentiment WHEN "Neutral" THEN 1 ELSE 0 END) neutral  FROM reviews r  WHERE r.sentiment != "nan"  GROUP BY app  )  SELECT a.app, price, rating, installs, reviews, positive, negative, neutral  FROM apps\_vw a, reviews\_vw r  WHERE a.app = r.app  AND positive > negative + neutral  ORDER BY positive DESC  """).show() |



We have used same query to check negative sentiment except for last two statements-

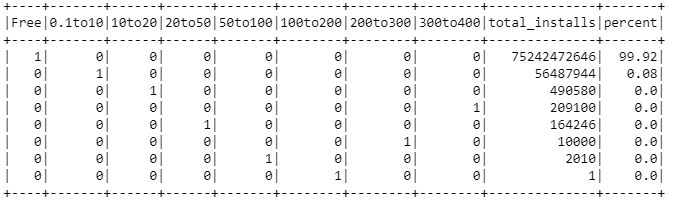
AND negative > positive + neutral

ORDER BY negative DESC

1. ***Correlation between price and number of installations***

We found that the most expensive apps are not frequently installed.

|  |
| --- |
| spark.sql("""  WITH app\_vw AS  (select CASE WHEN price = 0 THEN 1 ELSE 0 END Free  , CASE WHEN price > 0 AND price <= 10 THEN 1 ELSE 0 END `0.1to10`  , CASE WHEN price > 10 AND price <= 20 THEN 1 ELSE 0 END `10to20`  , CASE WHEN price > 20 AND price <= 50 THEN 1 ELSE 0 END `20to50`  , CASE WHEN price > 50 AND price <= 100 THEN 1 ELSE 0 END `50to100`  , CASE WHEN price > 100 AND price <= 200 THEN 1 ELSE 0 END `100to200`  , CASE WHEN price > 200 AND price <= 300 THEN 1 ELSE 0 END `200to300`  , CASE WHEN price > 300 AND price <= 400 THEN 1 ELSE 0 END `300to400`  , SUM(installs) total\_installs  from apps  GROUP BY 1, 2, 3, 4, 5, 6, 7, 8)  SELECT \*,  ROUND(total\_installs / SUM(total\_installs) OVER () \* 100, 2) percent  FROM app\_vw  ORDER BY 9 DESC  """).show() |



1. ***Correlation between rating and number of installations***

It's clear here that higher rated apps are more frequently installed. (*last row are apps with NULL rating*)

|  |
| --- |
| spark.sql("""  SELECT CASE WHEN rating <= 2 THEN 1 ELSE 0 END `0-2`  , CASE WHEN rating > 2 AND rating <= 3 THEN 1 ELSE 0 END `2-3`  , CASE WHEN rating > 3 AND rating <= 4 THEN 1 ELSE 0 END `3-4`  , CASE WHEN rating > 4 AND rating <= 5 THEN 1 ELSE 0 END `4-5`  , SUM(installs)  FROM apps  GROUP BY 1, 2, 3, 4  ORDER BY 5 DESC  """).show() |

