Final project – Scientific computing & Algorithms in Python

Oren Ben Eliyahu

On this Project, we were required to demonstrate our proficiency on Data Structures (Stack, Queue, Linked-List, Hash-Table &Graphs), Data Science, pre-processing to evaluation, and Machine Learning Algorithm
This report describes the details and conclusions from the project.

The project divided to three parts:

Part A: Data structures- we were required to implement the best data structures for:

1. Sorted Stack- which contains details of students, sorted by the values of each student Average.
2. Weighted Graph- which contains dictionary of cities on json file and develop functions on this undirected weighted graph.

Part B: Data Analysis & visualization- we were required to explore, analyze, and visualize the Dataset contains two tables that represent Apps and user behavior on Google Play.

Part C: Machine learning- we were required to predict by implementing KNN Algorithm user's Sentiment for a given App, based om the dataset.

In the following chapter, it will be detailed on the Algorithms, methods, and insights from each part of the project.

Part A: Data Structure

1. Sorted Stack: For a given student's details, the task is to save it in a sorted stack.

   First, I created Class of students, each student has an ID, first name, last name, age, gender and average.

   The attributes ID, age, gender, and average had constraint, and if passed the program raise an Error related to it.

   Secondly, to implement the stack I decided to use a linked list, due to its dynamic memory usage.

   I created a linked list Class that contains the following method:

   a. Is_empty- Returns True. Complexity time: O (1)
   b. Size- Return the size of the stack. In order to keep low complexity time, I set an attribute size to the stack, that saves the size of the stack. In case of popping student, it decries and the and vice versa. Complexity time: O (1).
   c. Push- In case the given details are valid, add an object of Student to its relevant place in the list by its rank of average. Complexity time: O(n).
   d. Pop- Return the first item in data structure. Complexity time: O (1).
   e. Top- Return the first item without pop it. Complexity time: O (1).

2. Weighted Graph: For a given dictionary of cities, paths and its weights, in json file format, the task is to load to file to a Graph object and apply methods on it. I decided to create the Graph based on Hash-tables and linked list data structure due to its dynamic memory usage and the fact the complexity time remains the same as other data structures in this case.

   I created a function to load the json file and transform it to a Graph object.

   The Graph Class include the following methods:

   a. load_graph_from_json- Gets a json file that contains a dictionary, call the deserialize method and return a Graph object. Complexity time: O (V+E)
   b. deserialize(dictionary): Gets an input a dictionary and returns a Graph object. I used it to load the json file in order to implement the other methods on it. Complexity time: O (V+E)
   c. get_vertices – Retruns a list of all the vertices in the Graph Complexity time: O (V).
   d. get_edges – Returns a list of sets and each set contains: source, target and weight. Complexity time: O (V+E).
   e. add_vertex- receives a vertex and a list of edges and weights and add it to the Graph. On this method I covered the following cases:

      If the vertex exists, it is required to update its list of edges by adding the new edge and its weight. Another case is, if one of the edges in the given edges not exists in the vertices list, I ignored it. Complexity time: O (V+E).
   f. delete_vertex- Delete the a given vertex and its connections from the Graph. Complexity time: O (V+E)
   g. BFS- Return all the paths from a source to target. Complexity time: O (V+E)
   h. get_the_shortest_path- Return the shortest paths from a source to target relaying on the smallest weights of the edges on the paths from source to target. Complexity time: O (V+E)
   i. serialize- Returns a dictionary of a given Graph object, I used the serialize method to save the Graph to a json file. Complexity time: O (V+E)

j. save_graph_to_json- save the given Graph object, call the serialize method and save it as a json file. Complexity time: O (V+E)

Part B: Data Analysis

This section of the project we were required to analyze a dataset and extract valuable insights from it.

About the data:

The dataset contains two tables of Apps and user reviews that scraped from Google play store.

Apps table:

(Name (PK), Category, Rating, Reviews, Size, Installs, Type, Price, Content Rating, Genres, Last Update, Current Version, Andriod Version)

User reviews:

contains 100 reviews for each App. not depending on the Apps table. (not have a PK)

(App's name (FK), Translated Review, Sentiment, Sentiment Polarity, Sentiment Subjectivity)

Exploring the data:

In order to extract valuable insights from the data I needed to clean it first. I deal with nan values, and other types of data types for example, in column Size on Apps table I required to convert the column form string to numeric values and replace M/k to 10^6/ 10^3 in order to calculate the 'heaviest' app.

The same process done for Installs column, dropping '+', ' , ' from numeric values to calculate the most installed app.

On this part we implemented the following functions on the dataset:

1. get_app_details_by_letter(letter): that gets a letter and returns all the apps that starts with the given letter.
2. get_average_polarity(app_name): that return the average of the Sentiment polarity for a given app if exist on the data set.
3. get_sentiment(app_name): Returns the calculated sentiment of a given app if exist on the data set. If the calculation is greater that 0 it returns "positive" and vice versa.

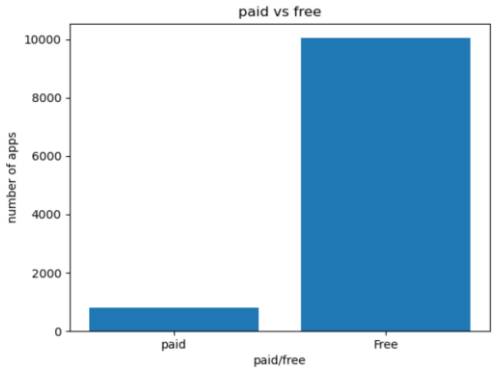
Analysis insights:

On this part I discovered that most of the apps on the data are free rather than paid 10,400 vs 800. And to calculate the popularity of the paid vs free

I multiplied it by the amount of installations for each Type, the result was that the gap increased dramatically:

Paid = 1.6^7

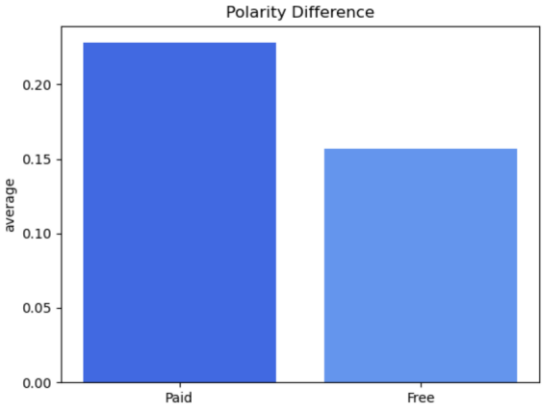
Free= 7.29^11 installations.



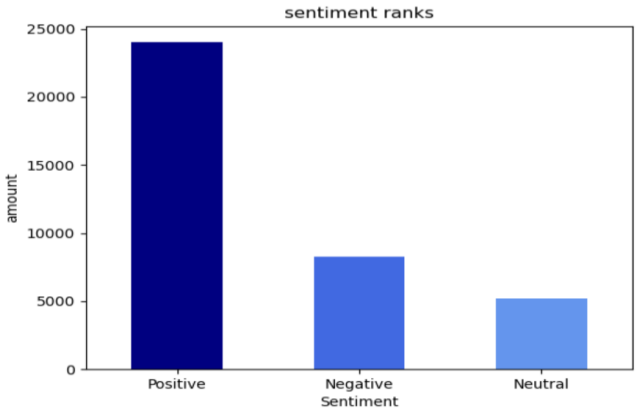
The polarity difference between Free vs paid apps can be seen in the following graph:

The diagram reveals that paid apps ranked with higher than free apps,

we can learn that paid apps apparently answer in a better way on the user needs than free apps.

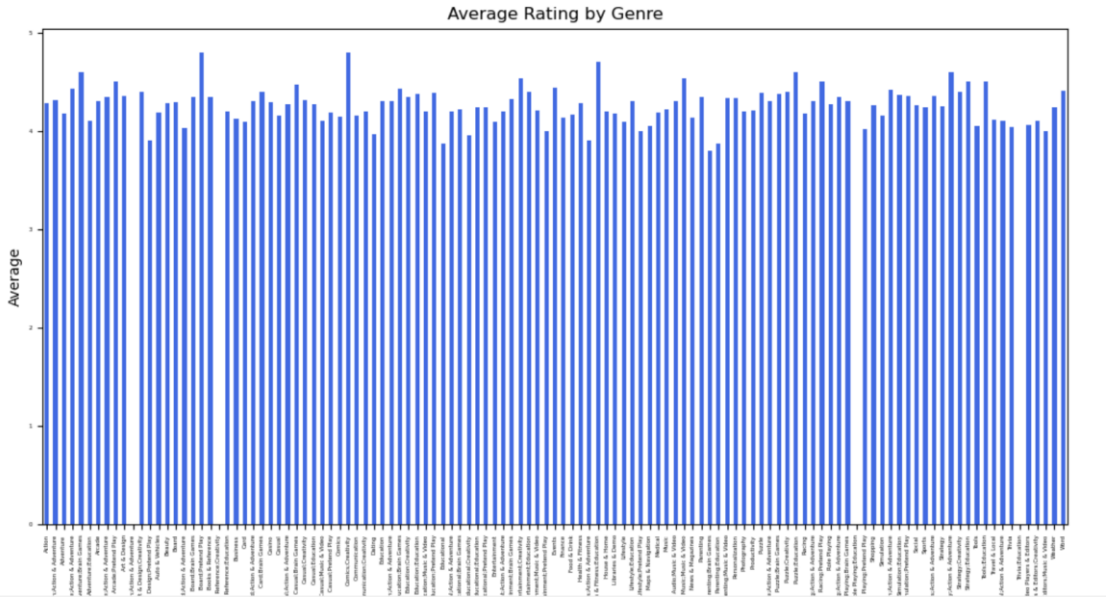


I wanted to explore the tendency of the users' Sentiments for the app on the dataset and we can learn from the following diagram that user tend to rate the apps in a positive way.



For each app genre I calculated the average rank:

From the diagram we can learn that most of the app in in each ranked between 4-5.



I wanted to get a better picture about apps' ratings. Therefore

I calculated the Mean, Mode, Median and STD, For each app category.

From the diagram we can learn the apps' rating distribute in a normal shape. the 'photo frames' category is the most stable with its ratings, we can see it by the value of its STD (0).

In addition, the rating of this category lower than other categories (2), while the rest categories' ratings values are between 4-5.

Part C: Machine learning

On this part of the project we were required to build a KNN classifier model, in order to predict the Sentiment for a given app parameters.

KNN is one of the basic supervised machine learning Algorithm that can be implemented in an easy way by using python libraries. Most of our work was to pre-process the data for the algorithm.

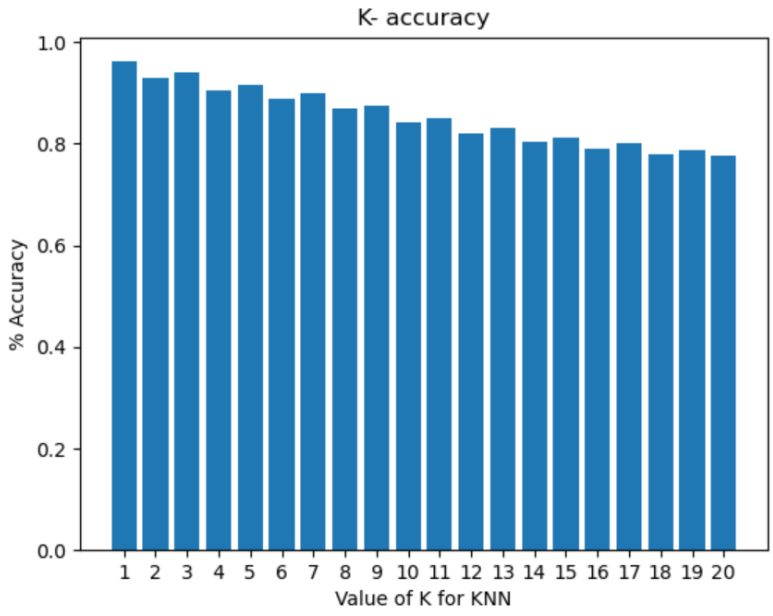
I merged the apps data and the user reviews and divided the tables to X- that represent the data, any y- that represent the labels (Sentiment).

Since KNN is an algorithm that calculate Euclidean distance, it was necessary to convert categorical variables to dummy variables, normalize the numeric variables by dividing all the column in the max value, and fill the missing values the data.

On the labels column I decided to drop the rows that the Sentiment was missing. In order to maximize the accuracy of the mode.

Now, when the data is ready, I split the data to 80% training and 20% test.

I ran the algorithm over k- between 1-20 and the accuracy results can be seen in the following diagram. The best k-is 1.



There is no need to use k=the data-length, in this case the accuracy must be lower than other than other lower k due to the calculations of distance from neighbors. The diagram in the previous page give it a proof.