Creating SVM binary classifier

Using Google Play apps data

Wiktor Kubera

Contents

1. Introduction

1.1. Definition of the Research Problem

The research problem concerns the classification of **applications as paid or free** based on their features.

1.2. Tools

1.2.1. Scraper and Data

Initially, I searched for a tool that would allow me to obtain specific data about applications available on the Google Play platform. I found a **scraper** that can be used with Python, namely `https://pypi.org/project/google-play-scraper/`.
Using the documentation, I conducted research on the types of data that can be retrieved using this tool. There is a large amount of data available, but many of them are textual. **Textual data are difficult to convert into numerical values, so I focused on numerical data.**
I found an online dataset at
`https://www.appbrain.com/stats/android-market-app-categories`, which allowed me to transform the textual parameter ”**Category**” into a numerical value. Specifically, I found a dataset that provides the number of applications per category along with the number of paid applications in each category. Thus, the ”Category” parameter was represented as the ratio of paid applications to the total number of applications.
After an initial analysis, the following features were selected to describe the application:

- **installs** - number of downloads
- **score** - application rating
- **ratings** - number of ratings
- **reviews** - number of reviews
- **categoryProb** - numerical value assigned to the category: $\frac{paidApps}{allApps}$

The predicted parameter is:

- **isFree** - whether the application is free or not

1.2.2. Machine Learning Tools

Since this is a binary problem, meaning an application can either be paid or free, it falls into the category of **binary classification**.
To build the model, I will use an algorithm that we studied in class – **SVM**.

2. Data Acquisition

2.1. Challenges

It turned out that the scraper available in Python is a wrapper for a scraper originally written in Node.js.
I initially thought it provided the same functionalities as the original scraper, so I started testing its capabilities.

2.1.1. No Category-Based Search

The Python scraper does not provide the ability to search by category, which is a significant problem. However, after experimenting, I discovered that if **I use a category as the query**, the scraper returns application data from that category.

2.1.2. Limited Number of Applications

It turned out that the scraper returns **a maximum of 30 applications** per query, which is quite a weak result. Additionally, it always returns the same applications, and there is a limit on the number of queries that can be made within a certain time frame.
I planned for the dataset to contain information about **25,000-50,000** applications.
With the Python scraper, this was not possible, so I decided to use the **original scraper written in Node.js** instead.

2.1.3. Category Mismatch

The Node.js scraper provides different categories than those I found online, along with the data needed to calculate **categoryProb**.
The scraper provides **54 categories**, out of which **23 categories do not have a corresponding categoryProb**.
I decided to exclude the categories for which categoryProb was unavailable.

2.1.4. Search Limitations

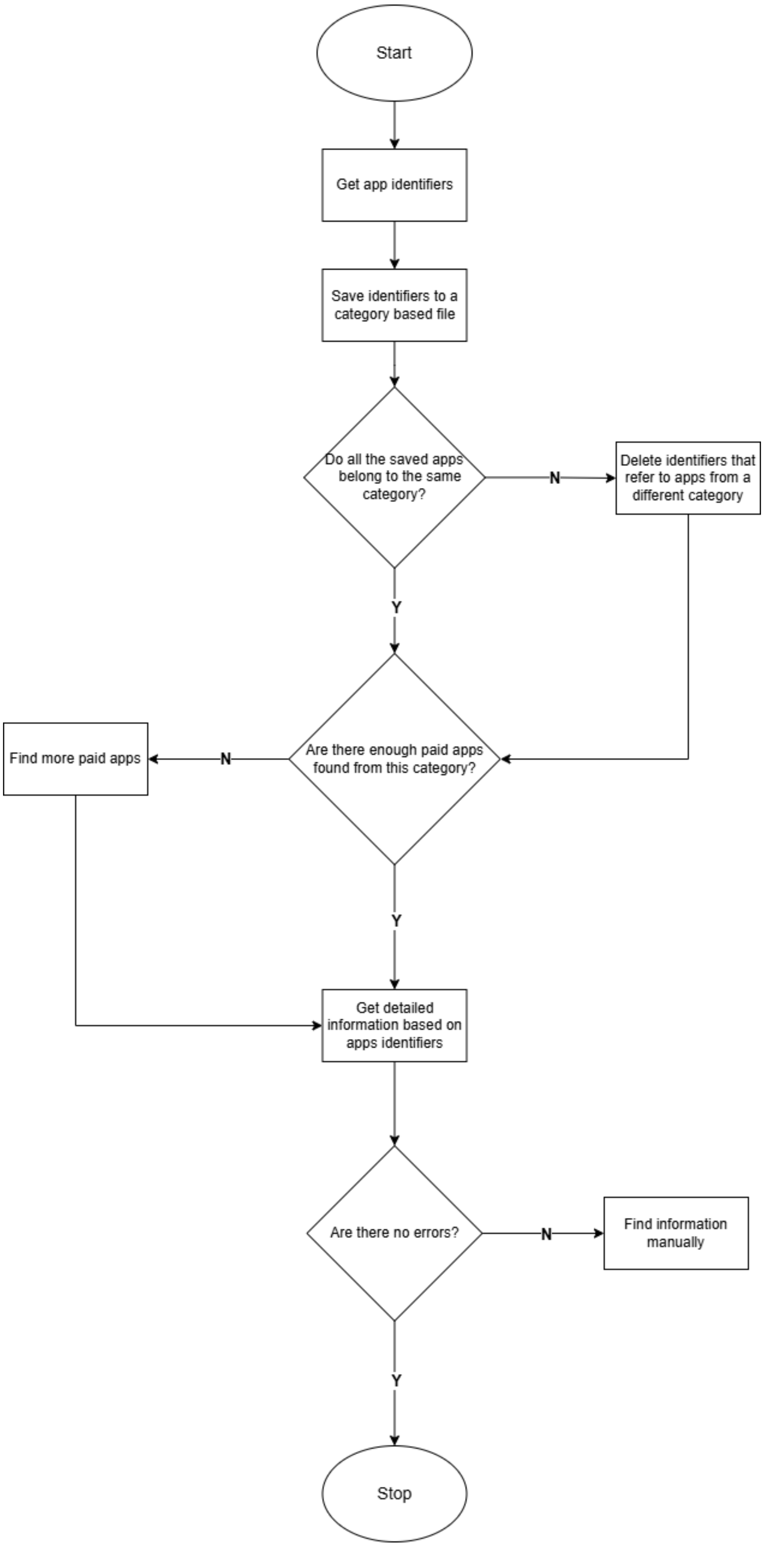
The Node.js scraper offers category-based search, which is a very useful tool, but it comes with three issues:

- A limit of 200 applications returned per query
- The same applications are always returned
- When searching by category, it is not possible to simultaneously search by keywords

After several experiments, I noticed that **I could simultaneously search for applications from a specific country along with the category**.
So, I modified the search process to attempt searching across all possible countries for a given category.
As a result, for the first category, I managed to increase the number of retrieved applications from 350 to **2,000+**.

2.2. Data Acquisition Process

Having developed the strategy, I first acquire the identifiers of applications from a given category, save them in a file, and then use search methods by identifier to retrieve the detailed parameters of the applications.
Detailed information about the applications is saved in a separate CSV file, and I will use it from within Python when building the model.
When attempting to retrieve detailed information about certain applications, there is an issue that is sometimes resolved by manually searching for the application (outside the loop).
After collecting a sufficient number of identifiers from a given category, I verify whether each application truly belongs to the specified category and check if there is a sufficient number of paid applications (since they are usually many fewer).
The algorithm illustrating the data acquisition process for a given category can be presented in the following block diagram.


Start
Get app identifiers
Save identifiers to a category based file
Do all the saved apps belong to the same category?
N
Delete identifiers that refer to apps from a different category
Y
Are there enough paid apps found from this category?
N
Find more paid apps
Y
Get detailed information based on apps identifiers
Are there no errors?
N
Find information manually
Y
Stop

3. Data Preparation and Cleaning

3.1. Data Cleaning

I managed to collect data on **143,310** applications. However, during the scraping process, a few applications contained missing data.
There are 120 applications that do not contain the complete set of required data, which is a very small number compared to the ”complete” applications. Therefore, I decided to remove them with the help of a script.

3.2. Data Preparation

I load the data from a CSV file using the pandas library.
The data is stored in a **DataFrame** structure.

4. Data Analysis

4.1. Contents

The prepared dataset contains **143,188** rows, where each row represents a single application and its parameters.
Each row is complete, meaning it does not contain any missing parameters.
The data concerning the individual parameters is highly diverse, and I will present it in the form of a table.

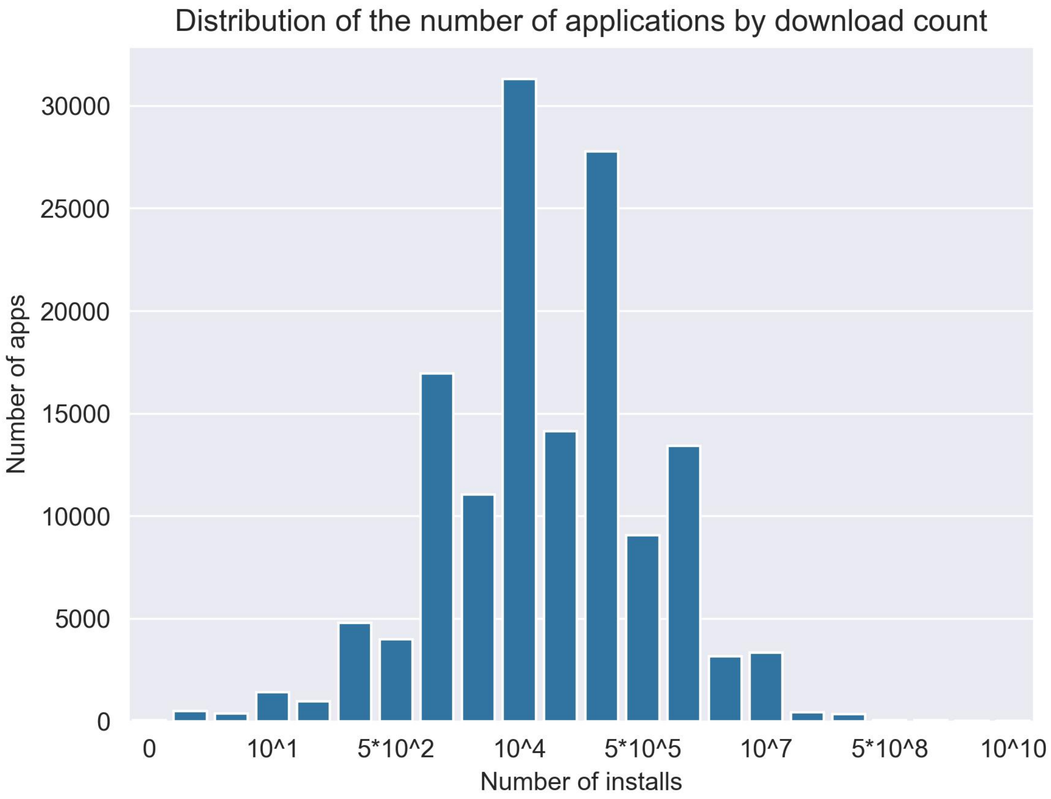
Parameter	Min	Max	Average	Std. Deviation
Installs	0	10000000000	2042198	69821755
Score	0	5	1.918	2.083
Ratings	0	176546332	30229	851576
Reviews	0	4469629	928	23845
isFree	0	1	0.9336	0.2488

Additionally, **there are 133,695 free applications** and **9,494 paid applications**. The breakdown of categories to which the applications belong is better illustrated with a chart.

4.2. Visualizations

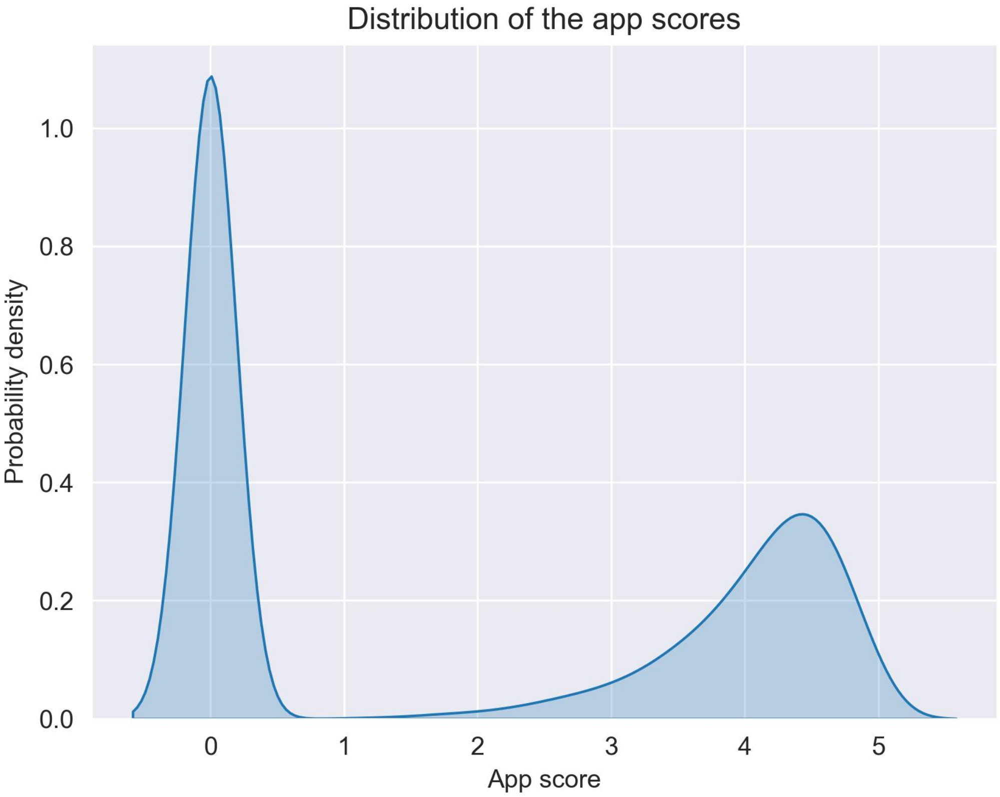
I have prepared several charts that will help illustrate the contents of the dataset and the types of data it contains.

4.2.1. Number of Applications with a Given Number of Downloads



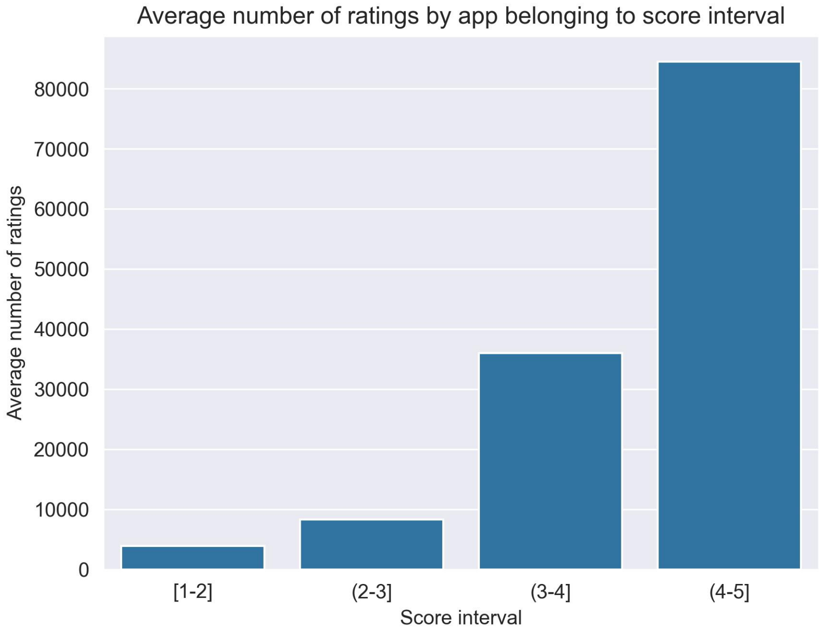
The chart presents the distribution of number of applications by the number of downloads. The largest number of applications has around 10^4 downloads.

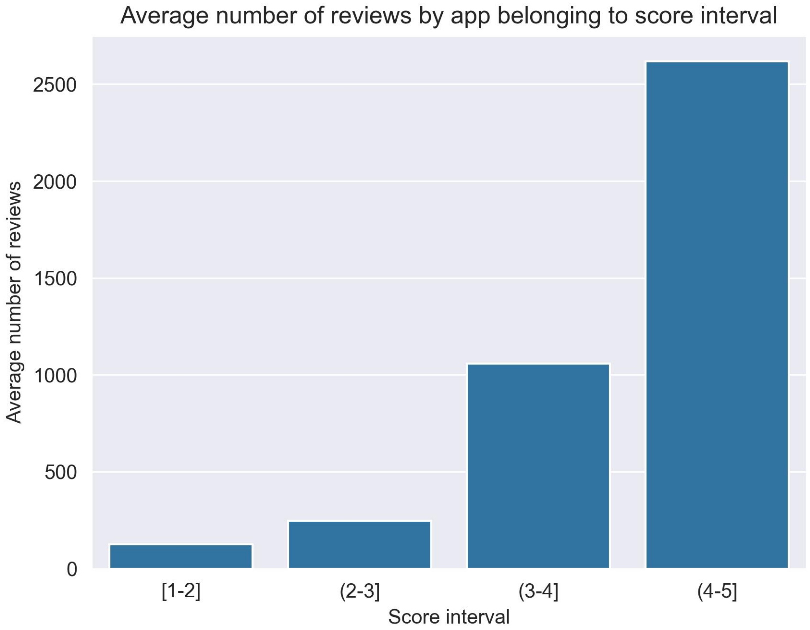
4.2.2. Distribution of Application Ratings



The chart shows the density distribution of application ratings.
From the chart, it is evident that the dataset contains **the most applications with a rating of 0**.
A rating of 0 is associated with a lack of ratings, meaning no one has rated the application yet.
As for applications that have been rated, the majority of them have a rating between 4.3 and 4.6.

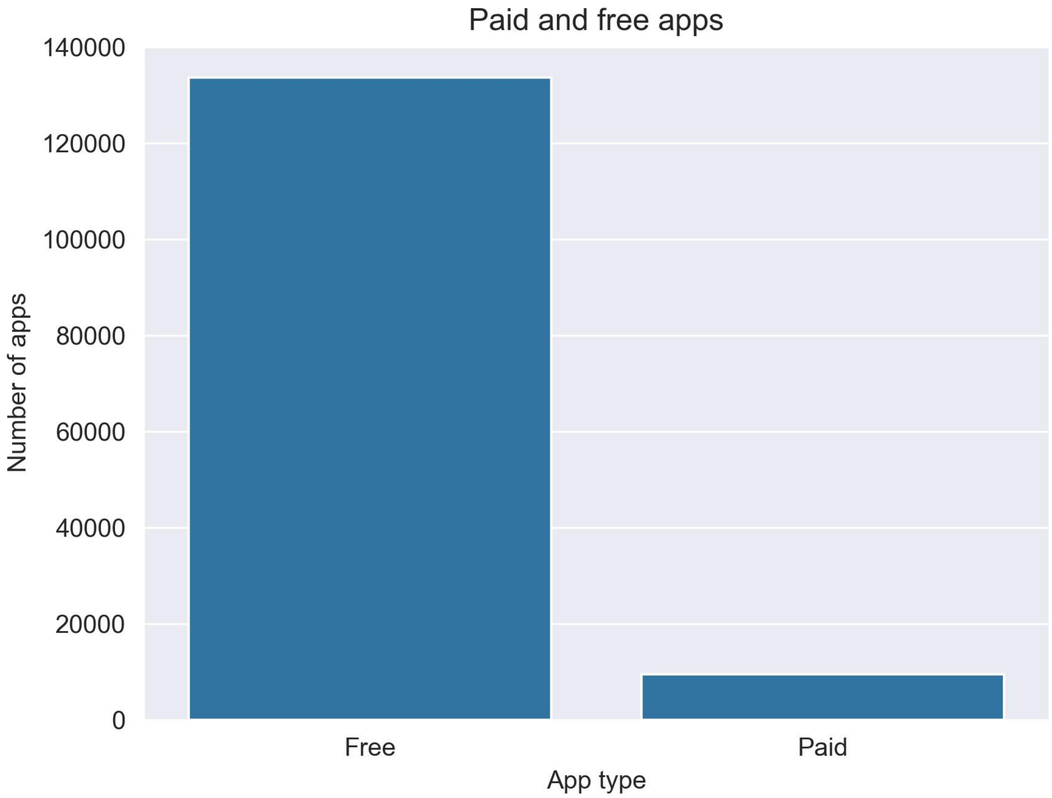
4.2.3. Relationship Between Number of Ratings, Number of Reviews, and Application Rating





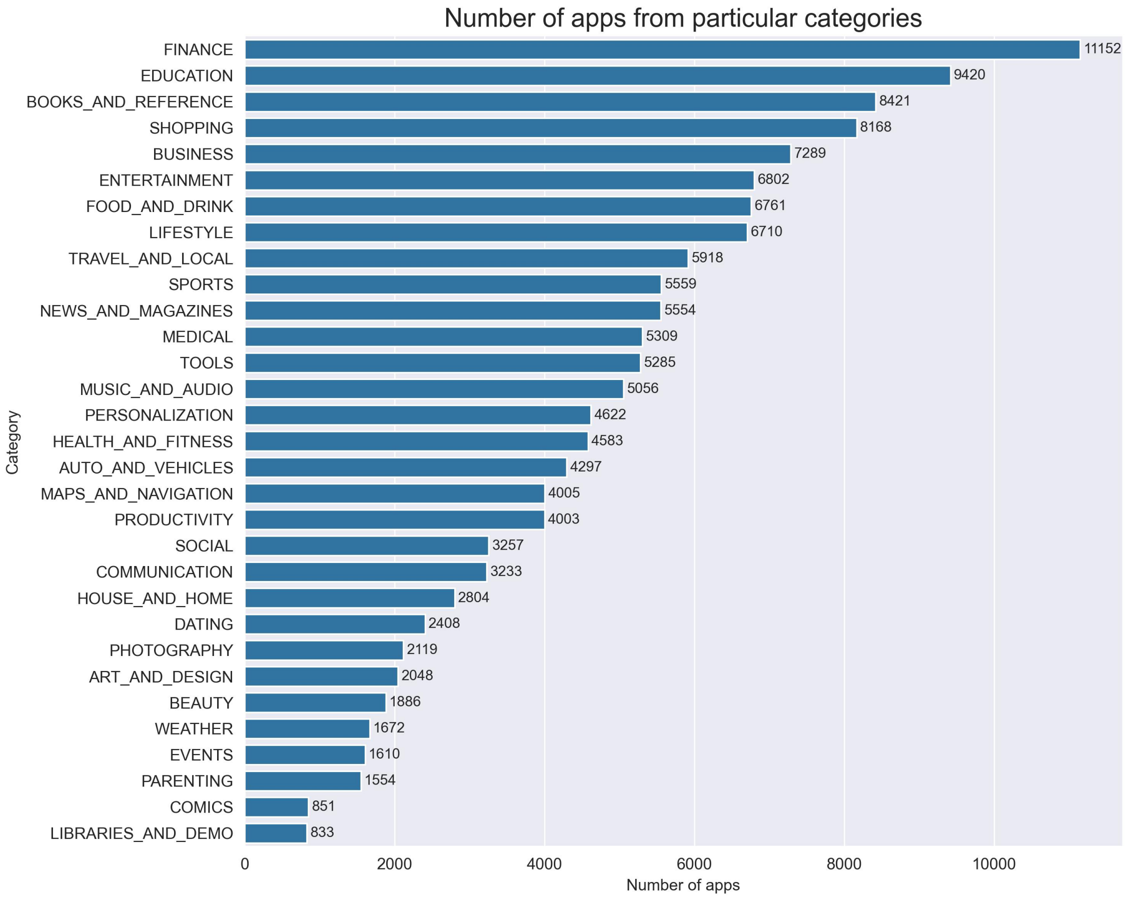
The charts show the average number of ratings and the average number of reviews for applications whose rating falls within the specified range.
The charts are very similar, so it can be concluded that the average number of reviews is directly proportional to the average number of ratings.

4.2.4. Comparison of Free and Paid Applications



The chart shows the number of paid and free applications within the dataset.
There are significantly more free applications than paid ones, which is natural, as statistically, when browsing the Google Play platform, users see many more free applications than paid ones.

4.2.5. Analysis of Application Category Membership



The chart shows the number of applications from each category. From the chart, it is evident that the largest number of applications belongs to the ”FINANCE” category, while the fewest come from the ”LIBRARIES_AND_DEMO” category.

5. Model Preparation

5.1. Division of Data into Input and Output Data

The data loaded from the csv file includes all parameters, including the output data. The first step will be to create a list of feature vectors and the corresponding list of output data 0, 1, which indicate whether an application is free or not.

5.2. Splitting Data into Training and Test Sets, Scaling

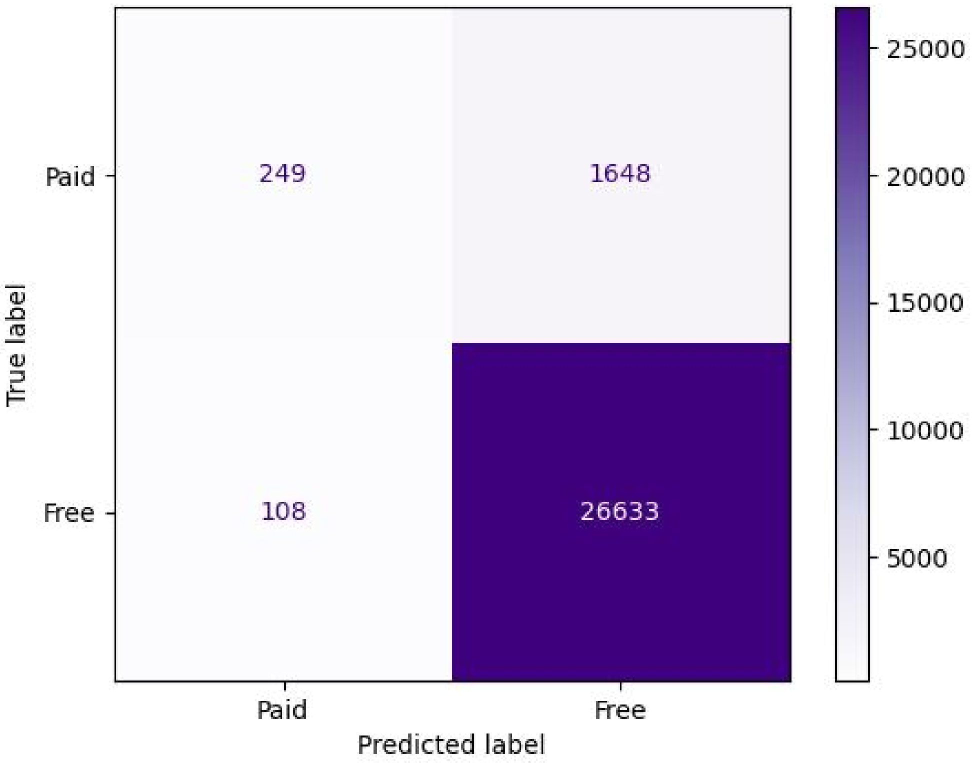
For data splitting, I use the `train_test_split` function from the `sklearn` library. I decided to split the data in an 80:20 ratio.
The parameters of the sets after splitting, compared to the original set, are presented in the table.

Set type	Quantity	Number of paid apps	Number of free apps
Training	114551	7597 (6.632 %)	106954 (93.368 %)
Test	28638	1897 (6.6241 %)	26741 (93.3759 %)
Original	143189	9494 (6.6304 %)	133695 (93.3696 %)

As can be seen, the dataset has been split correctly, i.e., the percentage of paid and free applications is very similar to the original dataset.
The input data must be scaled because the feature vectors contain parameters such as the number of downloads, which range from $[0; 10^{10}]$, and the number corresponding to the category, which lies in the range $(0; 1)$. Therefore, scaling of the data is necessary.
I considered two methods for scaling: **normalization** or **min-max**.
I decided on normalization, as min-max scaling can be significantly affected by outliers, while normalization is less sensitive to this.

5.3. Model and First Predictions

As the first instance of the model, I created an SVC model that uses a radial basis function (rbf) kernel. I made this decision because, based on the plots, I believe that apart from the relationship between the average number of ratings and the average number of reviews with respect to the rating, there are no other dependencies that resemble linear ones.
For the first predictions, I decided to use the default parameters for C and gamma, i.e., C=1.0 and gamma=”scale”.
The time it takes for the model to make predictions is quite long, around 4 or 5 minutes, which is caused by the fact that the computational complexity of SVM is quadratic in relation to the amount of data.
This is the main issue with the model, which limits the possibilities for optimization, a topic I will address later.
I will present the results of the first predictions using the confusion matrix.



5.4. Prediction Analysis and Quality Metric Selection

The model predicts free apps well, but this can be misleading since **there are over 10 times more free apps than paid ones**.
Thus, when evaluating the quality of the model, more focus should be placed on the errors related to predicting paid apps. Keeping this in mind, I will define the meaning of the parameters used in the quality calculation.

- **Positive class** - paid app (0)
- **Negative class** - free app (1)
- **TP** (True positive) - if the model correctly predicts that the app is paid
- **TN** (True negative) - if the model correctly predicts that the app is free
- **FP** (False positive) - if the model predicts that the app is paid, but it is free
- **FN** (False negative) - if the model predicts that the app is free, but it is paid

Analyzing the significance of these parameters, I conclude that I want to use a metric that, when optimizing the model, **minimizes FP and FN**.
The metric that will help me in this is the **F1 score**.
Below, I will present the parameters and the evaluation of the model's performance.

TP	TN	FP	FN	F1 score
249	26633	108	1648	**0.221**

An F1 score of 0.221 is a terrible result.
One can try to improve this result by experimenting with different values for the parameters C and gamma.
Optimization methods can be used for this purpose.

5.5. Model Optimization

Since I do not have an explicit formula for the quality function with respect to the parameters C and gamma, I cannot apply gradient methods here.
I am forced to use exploratory methods to search for parameters that will maximize quality.
I will use the Grid Search method, which will search a grid of parameters to find the best set of solutions.
The search grid consists of 6 different values for gamma and 200 different values for the parameter C.
As for the values of C, they are values from the range $(10^{-2}; 10^{2})$.
Furthermore, the formula for the next value of C is described by the expressions:

$$C_i = 10^{k_i}$$

$$C_{i+1} = 10^{k_i+0.02}$$

$$k_i \in [-2; 2]$$

$$k_1 = -2$$

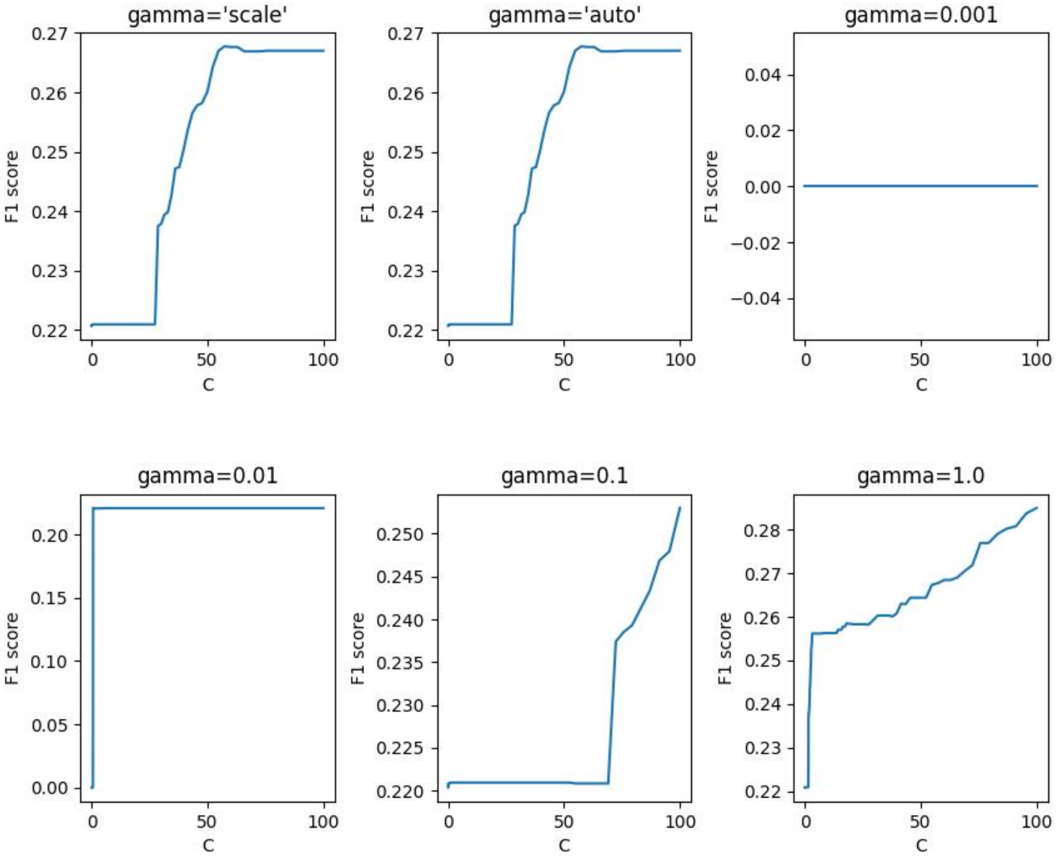
$$1 \leqslant i \leqslant 200$$

The set of values for the parameter gamma is:

$$\gamma \in \{\text{"auto"}, \text{"scale"}, 0.001, 0.01, 0.1, 1.0\}$$

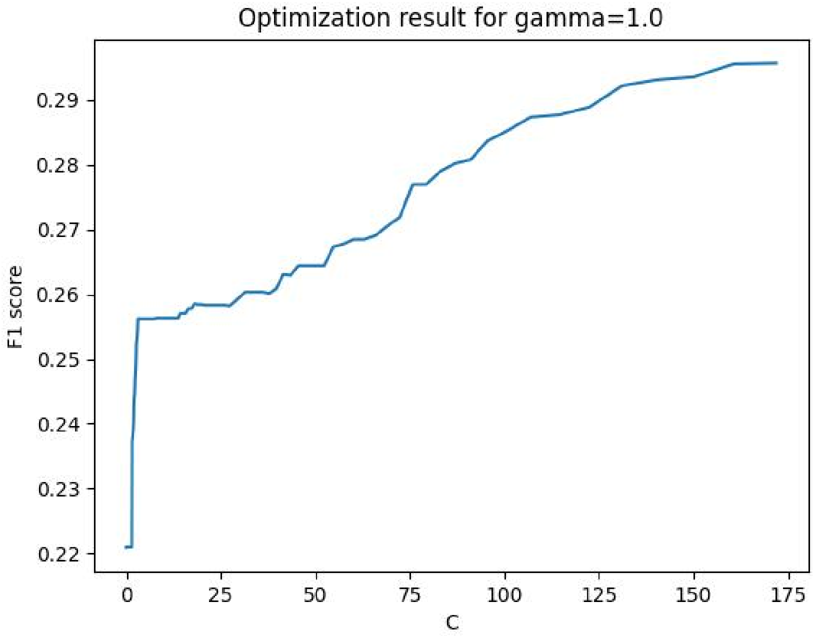
I am running the optimizer for 24 hours (this long due to the slow performance of the model with such a large dataset), and the program saves the results in text files. After the optimizer finishes, I will examine the files to find the parameters for which the F1 score is the highest.

After the optimizer finished, I obtained data that describe the impact of the C parameter with fixed gamma values, which is illustrated by the chart on the next page.



From the chart, it can be seen that the most promising gamma parameters for further optimization are gamma=0.1 and gamma=1.0.
Since the optimization process takes a long time, I chose one gamma value, namely gamma=1.0, as it achieved the highest quality.

I measured the quality for 8 subsequent values of the C parameter, because for such large values of C, the time for making predictions can exceed 20 minutes.
The chart presenting the results of the final optimization of the model quality is on the next page.



The highest quality of the model, for which predictions are made in a reasonable time, is F1=0.2921997523730912, for gamma=1.0 and C=131.11339374215643.
For these parameters, the model makes predictions in about 10 minutes.
Below, I present a comparison of the parameters of the predictions made by the model with these selected parameters, compared to the initial model.

Model	**TP**	**TN**	**FP**	**FN**	**F1 score**
Initial	249	26633	108	1648	0.221
Final	354	26569	172	1543	0.292

As can be seen, the model improved its ability to correctly classify paid applications and reduced the number of misclassifications of free applications as paid ones.
However, the number of FPs increased, which means the model more frequently considers an application as potentially paid.
The model quality **increased by 32.25%**, at the cost of **30 hours of optimization**.
The quality of the final model is still not good. To further optimize the model, a parameter grid with more than 2 dimensions is required, but such optimization could take months.

6. Conclusions, Comments

- I believe that for 140,000 data points, each with 5 features, **SVM is not the best algorithm**, especially when there are complex relationships between the features.

- For such a large dataset, **predictions made by the model take too long**, at least 4 or 5 minutes.

- Optimizing such a model is challenging because **exploratory search** must be used, and each iteration takes a very long time. Additionally, classification time depends on the parameters and can even reach 30 minutes.

- The model would be much **more efficient for a smaller dataset**.

- Algorithms based on neural networks, especially deep neural networks, have the potential to model nonlinear relationships between features and can achieve good results for large datasets. With 140,000 data points and 6 features, it is possible that neural networks can effectively detect and leverage these nonlinear relationships for more accurate classification.

- **Parallelizing the classification process** could help address the performance issue.