Franky Liu

Recommendation Model based on Steam Reviews - Written Report

Introduction

The project will explore the Kaggle dataset 100 Million+ Steam Reviews, which contains data such as steamid, language, review sentiment, helpfulness score, review body, among many others. The goal of this project is to combine two different machine learning models into an intelligence pipeline to create a recommendation system based on review sentiment and predicted playtime, which will all be implemented in Apache Spark. First, we will apply NLP techniques such as tokenization, stopword removal, and IDF feature processing to clean and prepare our data. After, using these processed features, we will build a logistic regression model that classifies user sentiment and predicts if they would recommend the game or not (0 or 1). Following that, we will build a linear regression model that predicts playtime based on inputs of the full review text and predicted sentiment from the logistic regression model. Lastly, we will integrate the outputs from the previous two models into a single intelligence pipeline to create a recommendation system. This system recommends games by predicting how much time a user might spend on different titles and select the title with the highest projected playtime. This recommendation system aims to simulate a real-world machine learning pipeline that could be the framework of much more powerful recommendation systems that platforms like Steam might use.

Methods

For this project's methodology, it is broken down into 4 sections - Data Exploration, Processing, First Model, and Second Model and Recommendation Pipeline.

Data Exploration - In this section, our primary methodology is to employ data frame transformations to clean and explore the relationships between the data. First, we dropped some columns that were not used in our project - such as geo locations. Next, we casted all of the columns to types that we can use, i.e. the playtime of certain games from string to float so we can apply transformations to it at later stages. Then, we filtered out data that is useful for our project, such as only reviews in English and are not null. Lastly, to clean our data, we employed a filter to try and only record reviews from real accounts. We did this by first checking to see if the account bought the game and if not, they at least owned another game. This filter is effective because most spam accounts would not have spent money and by checking user inventory, we can effectively remove accounts that were most likely not made by real humans. Furthermore, we filtered again to check for users that have a certain playtime to make sure that most reviews are fair and are made from actual gameplay. After our data had been optimized, we did preliminary exploration of the relationships between features.

Processing - In this stage of the project, we utilized the NLTK library and tokenized our review text. We built a function - process word that takes strings as input and output tokenized words with stop words removed. For example, with a string ‘This is a sample review! Make sure to upvote this.’, the output would be ['sample', 'review', '!', 'Make', 'sure', 'upvote']. In this function, we also kept specific punctuations because we believe that these carry some element of sentiment, such as ‘!’ and ‘?’. Lastly, we also kept ‘-’ because some words such as the game title ‘Counter-Strike’ are being removed due to ‘-’. After we have successfully built this function, we moved on to addressing the problem of skewness of the dataset where positive sentiment is over 86% while negative is only 14%. In order for the model to not overfit, we will downsample positive sentiment to roughly the same as negative sentiment. This will also help our models later on because processing 40 million rows would run into memory issues in the current Spark environment. After we tokenized the reviews, we did some data exploration such as looking at what the top 20 most used tokens are.

First Model - Logistic Regression was the model we chose to be our first model. It is utilized to predict the sentiment of the review text and predict if the user would recommend (binary 1) or not recommend (binary 0) the game. Before we can initialize the model, we have to first convert our tokenized text into features. To do this, we created a pipeline and introduced both a hashing conversion and an IDF conversion. First, we hash our tokenized text into 8000 features, then we initialize IDF vectors after the hashing has been completed and have our models fit to the finished pipeline (Figure 8). After that, we splitted our model by 60-20-20, which represented the training set, validation set, and test set respectively. Following that, we also initialized an evaluator and analyzed our results, which will be discussed in a later section.

Second Model and Recommendation Pipeline - Due to issues with SDSC and time constraints, the second model and the recommendation pipeline was not able to be completed on time. However, since the majority of the framework has been completed, this can easily be revisited in the future to continue to be built to completion.

Results

From our data exploration stage, we generated a top 20 reviews distribution of the number of reviews per game (Figure 1), as well as the distribution of playtime per game (Figure 2). We also explored the average playtime for all users across the data (Figure 3), where we had to log the results as the data was skewed (Figure 3). Lastly, we explored the relationship between games owned and total playtime, which had around a -13 correlation, meaning that playtime per title went down as the number of titles increased in an user’s inventory (Figure 4).

In the processing phase, an example of the stop-word removed result can be seen in (Figure 5). An example of how we downsampled our data to remove skewness and prevent memory issues

can be seen in (Figure 6) and a visual representation of the top 20 most used tokens can be seen in (Figure 7).

Lastly, for our logistic regression model, our initiation of the model can be seen in (Figure 8), where we created a pipeline for the hashing and IDF functions. The accuracy of the model was 81.8% for the training set, 81.75% for the validation set, and 81.71% for the test set. The confusion matrix of the model can be seen in (Figure 9). In the confusion matrix, we can see that the model is making more mistakes predicting a bad sentiment compared to positive. This is to be expected since there were not a lot of negative labels to learn from - so much so that we had to downsample our positive labels to prevent overfitting.

Discussion

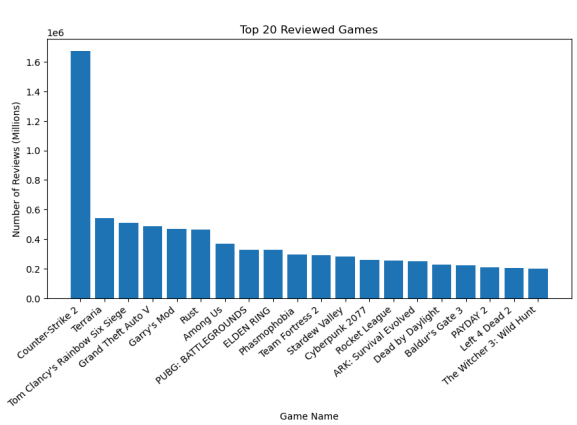
From the results of our model, it is interesting to note the accuracy of the test set and validation set is very similar, it would be a red flag if the test set has a higher accuracy but it is not in this case. This could indicate a couple things, such as the model having a fairly good generalization and not overfitting to the data. However, to play the devil's advocate and explore what potential shortcomings that might have caused this, there are several candidates. The first being that the TF-IDF vectors are too sparse for the model to learn and the results of ~80% might be generous due to the specific seed chosen and the actual variations of the results might be worse than portrayed. Another candidate is that we lost a lot of information when we downsampled our positive sentiment data by around 86% to match with the number of negative sentiment data. Although this was partly due to the fact that the project environment would not be able to train such a big cluster of a dataset and this also brings up another point - which was the environment setup. A major bottleneck in this project was the amount of memory available, which might have been due to setup. This bottleneck placed some restrictions on how much of the data we can pull to work with and prevented us from utilizing caching and deploying more memory intensive models such as Random Forest. Overall, the accuracy of the model was satisfactory and the shortcomings that were noted were a valuable experience to add on to future models.

Conclusion

Our project, overall, is headed in a solid direction. The next step for this project is further implementation and work on the second model and the recommendation pipeline. We learned a lot from the setbacks during this project, and in the future when working with Spark again, we feel that we have the experience equipped to set up a more robust and efficient environment. This is an area of improvement we want to emphasize because we felt, as iterated earlier, our environment setup was a pretty big bottleneck on the models and transformations we originally planned to do. Running out of memory was a big issue and we cut back a lot on the size of the data and opted for simpler models when models such as Random Forest might have performed better. This also applies to our hashing process, as we wanted to try different methods such as word2vec but we were running into memory issues.

Statement of Collaboration
Franky Liu, solo team member

Figures and Graphs



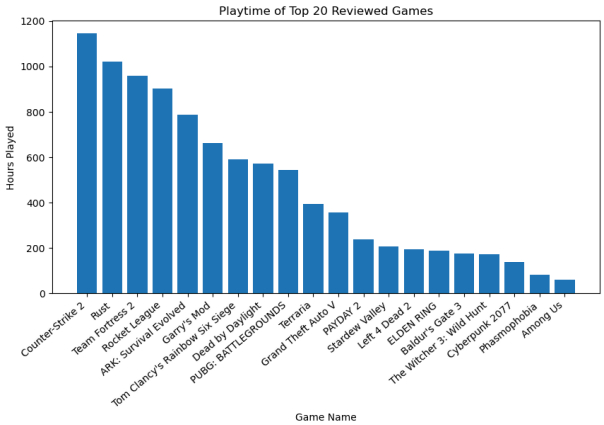
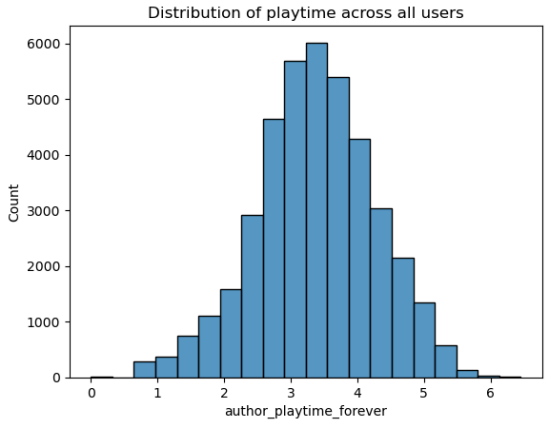


Figure 1 (top left) and Figure 2 (top right)



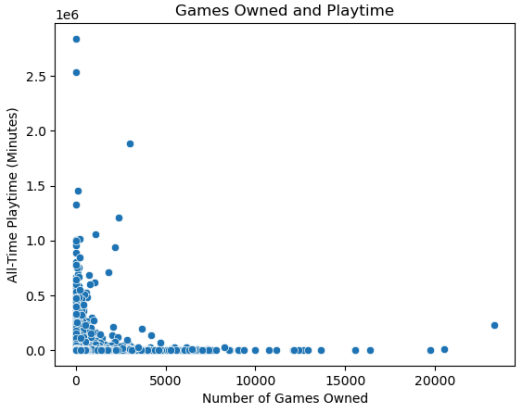


Figure 3 (top left) and Figure 4 (top right)

```python
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()

def process_word(text):
    words = word_tokenize(text)
    tokens = []

    useful_punct = ['!', '-', '?']

    for word in words:
        word_lower = word.lower()
        if word_lower not in stop_words:
            if word.replace('-', '').isalpha():
                word = '-'.join([lemmatizer.lemmatize(w, pos='v') for w in word.split('-')])
                tokens.append(word)
            elif word in useful_punct:
                tokens.append(word)

    return tokens
```

Figure 5 (top)

	voted_up	count
0	1	4946381
1	0	5095265

The new distribution is 49.26% and 50.74%.

Figure 6 (top)

```
from pyspark.sql.functions import explode

exploded_words = reviews_processed_reduced.select(explode(reviews_processed_reduced['processed_review']).alias('word'))
word_counts = exploded_words.groupBy('word').count()
top_20_words = word_counts.orderBy('count', ascending=False).limit(20)

top_20_words.show()
```

```
+------+-------+
|  word|  count|
+------+-------+
|  game|1194649|
|     !| 348395|
|  play| 329517|
|   get| 263727|
|  like| 235515|
|  time| 170259|
|  good| 164295|
|  make| 162311|
|   fun| 151156|
|     ?| 124578|
|really| 123117|
|     -| 119308|
|   one| 118984|
|  even| 116728|
|    go| 109339|
| would| 107414|
|  much|  88661|
|  feel|  88268|
|   buy|  86506|
|  want|  83964|
+------+-------+
```

Figure 7 (top)

```
from pyspark.ml.feature import HashingTF, IDF
from pyspark.ml import Pipeline

hash = HashingTF(inputCol='processed_review', outputCol='features', numFeatures=8000)
idf = IDF(inputCol='features', outputCol='idf_features')
pipeline = Pipeline(stages=[hash, idf])
model = pipeline.fit(reviews_processed)
fitted_model = model.transform(reviews_processed)
```

Figure 8 (top)

```
+--------+----------+------+
|voted_up|prediction| count|
+--------+----------+------+
|       1|       0.0|129326|
|       0|       0.0|781841|
|       1|       1.0|860481|
|       0|       1.0|237462|
+--------+----------+------+
```

Figure 9 (top)