STEAM REVIEW DATA ANALYSIS AND PREDICTION MODELLING

Andrew Ghafari*
M.S. CS @ UCSD
aghafari@ucsd.edu

Jay Jhaveri*
M.S. CS @ UCSD
jjhaveri@ucsd.edu

1. INTRODUCTION

For this assignment, we decided to use the steam dataset [1]. To give you a brief overview of the dataset, it contains reviews (almost 400k rows of data) from Steam's best-selling games. The dataset is comprised of 8 columns:

i. Date posted: (The date a review is posted)
ii. Funny: (How many other players think the review is funny)
iii. Helpful: (How many other players believe the review is helpful)
iv. Hours played: (How many hours does a reviewer play the game before make a review)
v. Is early access (Did this user get this game as part of early access)
vi. Recommendation (Whether the reviewer recommends the game or not)
vii. Review (The text of the user review)
viii. Title (The game's title that's being reviewed).

We were motivated to use this dataset for this assignment because we are gamers ourselves, and we have previously used Steam to get recommendations for games before purchasing them, so it was a fun idea to try and do a project that revolved around this.

Keywords: *Recommendation Systems, Steam Reviews, TFIDF, Read Not Read, Machine Learning, Multi class Prediction, Sentimental Analysis, Natural Language Processing.*

2. Dataset and Data Exploration

The raw dataset we are using comes from Kaggle; it has been scraped from the steam store and hence is not perfect from a programming perspective. The following are the base preprocessing techniques we did per column:

- Date posted on: Converted the given date into a comparable data type using the DateTime library.
- How many people found it Funny, Helpful and the Hours played by the reviewer converted to float.
- Is early access converted to 0/1
- Did the user Recommend the game or not converted to 0/1
- Text preprocessing for the review text and the game name using:
- Using the html2text library to remove basic HTML tags and hyperlinks
- Remove all special escape sequences and greater than symbols from the text using string.translate()

We will further preprocess this data depending on the model we plan to use.

In addition, we manually labeled the games' genres because we also wanted to do a predictive task centered around the genre of the games in hand. As I have said, we added another column to our existing dataset to describe the multi-class genres (that we have found using Steam's website). We added multiple genres to each game (as per Steam's description).

We will now give you a few interesting insights about the data at hand.

A- <u>The number of reviews we have per genre.</u>

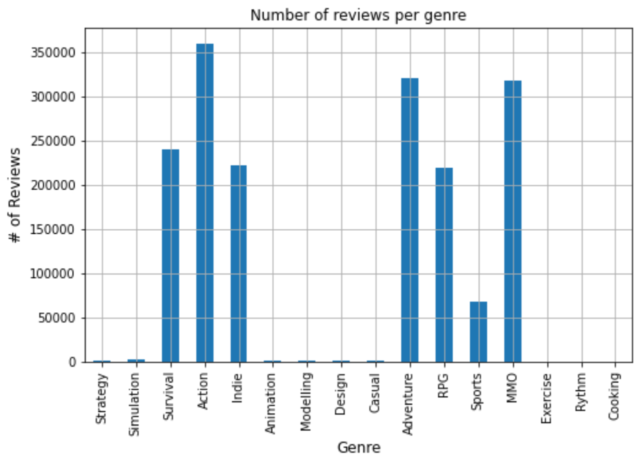


Fig. 1 #Reviews per Genre

We can see a disparity between the numbers, which was expected because a lot of current popular games tend to be in action, survival and adventure instead of animation, exercise and cooking.

* Both authors contributed equally

B- <u>The number of reviews we have per number of genres.</u>

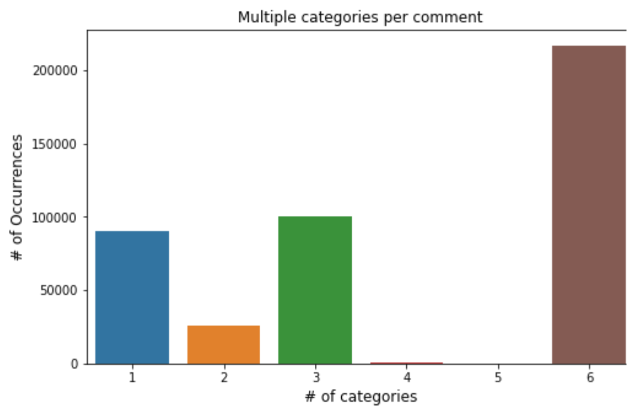


Fig. 2 # Reviews per # Genres

We can also see a disparity between the numbers due to the difference in the number of reviews per game that we will plot now.

C- <u>Number of reviews per game</u>

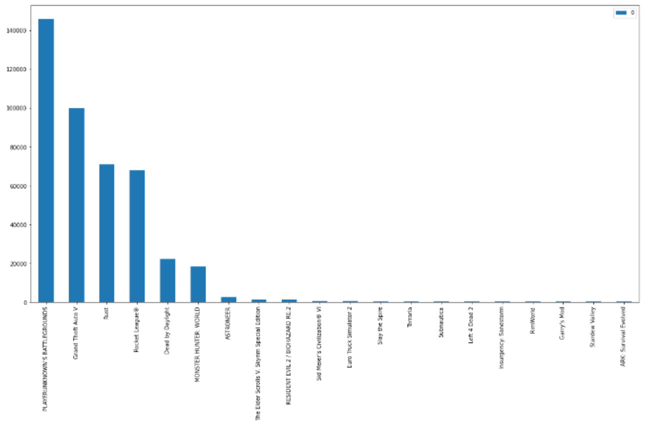


Fig. 3 # Reviews per Game

This clearly explains the huge differences we see in the above two plots.

D- <u>The number of reviews we have per length of the review.</u>

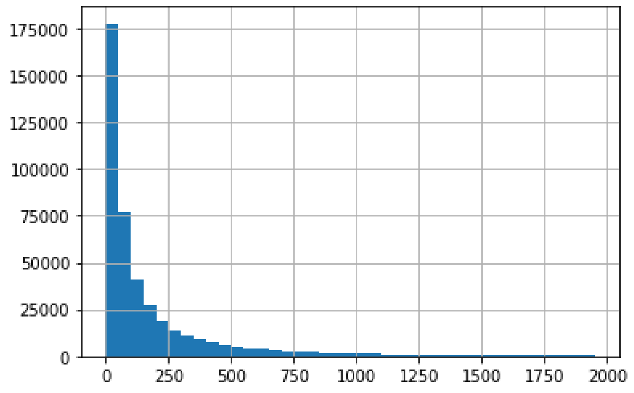


Fig. 4 # Reviews per length of reviews

This was also expected because a lot of reviews are shorter in nature.

E- <u>Percentage distribution of reviews per recommendation.</u>

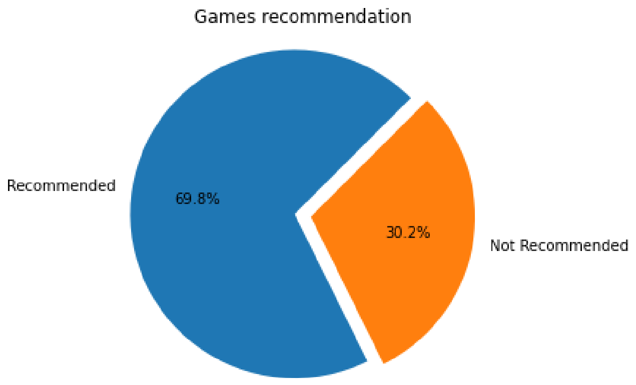


Fig. 5 Distribution of Games Recommendation/Review

Don't mind some OG pacman xd. Anyways, we can see almost double the number of reviews recommending a game, which may indicate that gamers tend to give positive reviews more than negative ones, unlike e-commerce reviews. Also, this might create problems while training a prediction model, so we will try to balance the positive and negative reviews.

F- <u>Frequency of the reviews per hour played before giving it.</u>

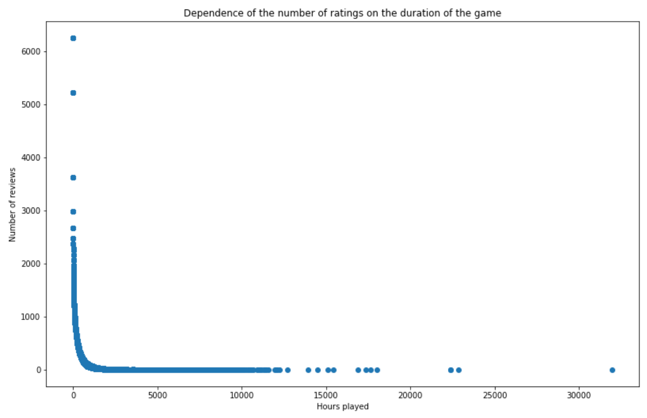


Fig. 6 # Reviews per Hours played before reviewing

This is also expected, as people tend to give reviews after some hours of play time.

G- DataSet WordCloud

One more thing we did is that we used NLTK and the wordcloud library of python to preprocess and to get a WordCloud that is representative of the games we have.

We first tokenized the words of the review, lowered everything to lowercase, removed stop words and punctuation, and lemmatized the reviews to get root words.

This is the WordCloud we got for all the reviews.

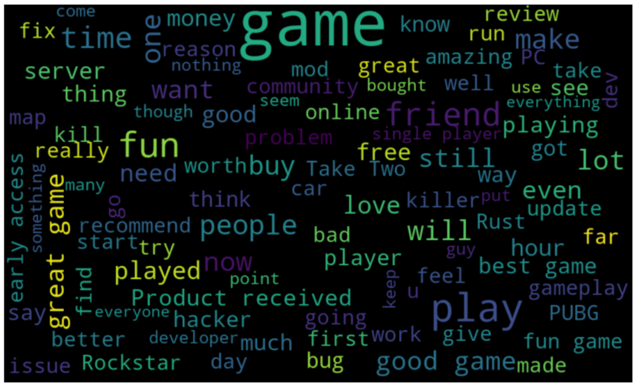


Fig. 7 Wordcloud of all reviews

This is the WordCloud we got for the positive reviews that recommended the game.

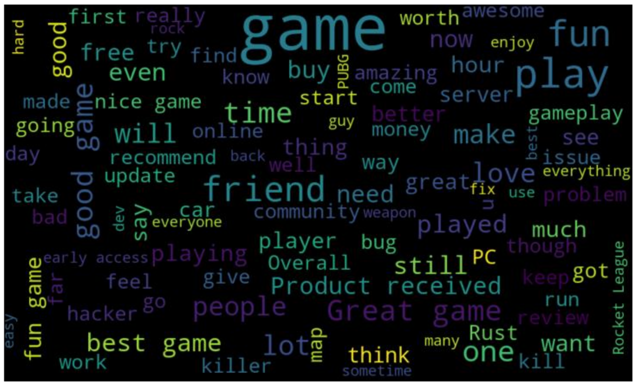


Fig. 8 Recommended wordcloud

This is the WordCloud we got for the negative reviews that were recommended.

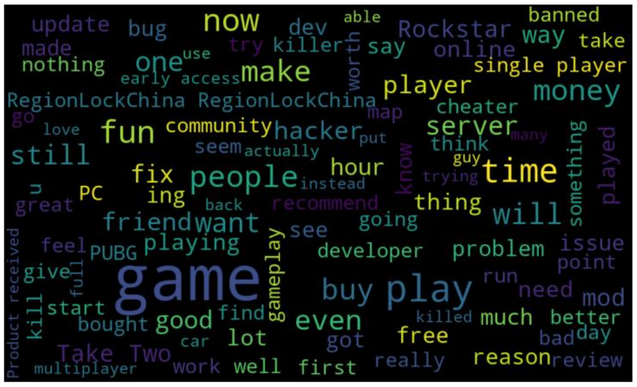


Fig. 9 Not Recommended wordcloud

H- WordCloud of each game

We have 48 distinct games in the dataset, so we aggregated all the reviews that corresponded to each of these 48 games and created a WordCloud that describes each one of these games given the reviews of the gamers that reviewed them. And since we are gamers ourselves, we decided to mask the WordCloud with a controller bitmask to make the cloud look like a controller.

Here is a sample output, and we will append the complete version clearer in the report's appendix.

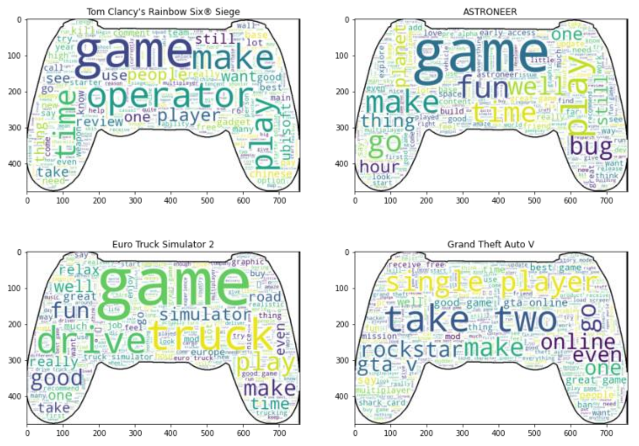


Fig. 10 Wordcloud per game

Although it looks great, if you notice, there are a lot of general gaming keywords in them. We think TFIDF should solve this. Let's try:

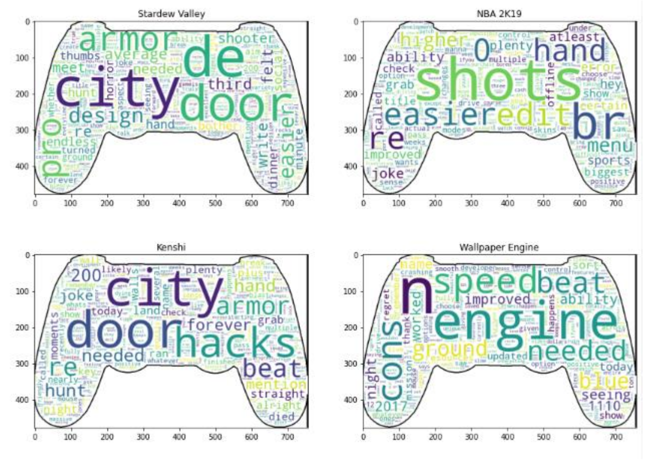


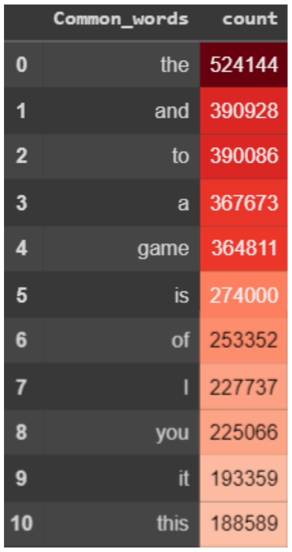
Fig. 11 Wordcloud per game with tfidf

Yay! as expected, this is much better than before, and much more game specific. One can easily get the basic context (genre) of the game from this.

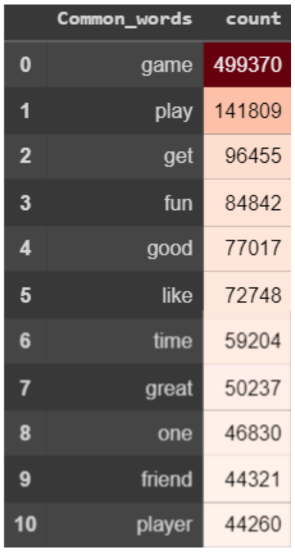
I- <u>The magic of preprocessing</u>

Another thing we did is that we printed out the most common words before preprocessing and after preprocessing. Please note: that we used a similar preprocessing to generate the wordcloud. To reiterate: we lowered everything to lowercase, we removed stop words, we removed punctuation, and we lemmatized the reviews to get root words.

You can see that by preprocessing, we removed many words that did not hold any specificity of the review but rather just properties of that language.



	Common_words	count
0	the	524144
1	and	390928
2	to	390086
3	a	367673
4	game	364811
5	is	274000
6	of	253352
7	I	227737
8	you	225066
9	it	193359
10	this	188589



	Common_words	count
0	game	499370
1	play	141809
2	get	96455
3	fun	84842
4	good	77017
5	like	72748
6	time	59204
7	great	50237
8	one	46830
9	friend	44321
10	player	44260

Fig. 12 Before (left) After (right) Preprocessing

3. PREDICTIVE TASKS

We decided to do a couple of predictive tasks. In the first one, we will predict whether the user recommends the game to others based on the review. For the second task, we will use the review's text to predict the genre of the game being reviewed.

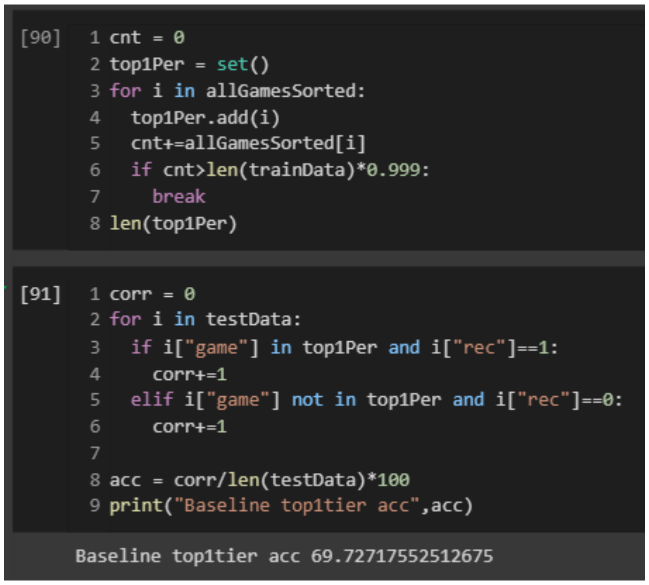
A- <u>Task 1: Recommended/Not Recommended.</u>

Let us move on to the first task, the recommended/not recommended predictive task. We first started doing some baseline models that we got introduced to during coursework, but then we decided to also explore more combinations of known and unknown to better our accuracy.

a. <u>Top N most popular games</u>

Here, we took inspiration from Read/Not Read prediction task that we were assigned during the first part of assignment 1. We took the top 0.999% of the books based on the number of reviews they appeared in. We know this threshold sounds way too high, but if we refer to the exploratory part of this paper again, it becomes more plausible. Even at 0.999%, We only got 35 out of all the games in the dataset. This is due to the data being highly skewed by nature.

We can see that the naive method of directly correlating the popularity of the game with a recommendation like we did for Read/Not Read is not bad, and is giving an accuracy of 69.7 %



```
[90]
cnt = 0
top1Per = set()
for i in allGamesSorted:
  top1Per.add(i)
  cnt+=allGamesSorted[i]
  if cnt>len(trainData)*0.999:
    break
len(top1Per)
```

```
[91]
corr = 0
for i in testData:
  if i["game"] in top1Per and i["rec"]==1:
    corr+=1
  elif i["game"] not in top1Per and i["rec"]==0:
    corr+=1

acc = corr/len(testData)*100
print("Baseline top1tier acc",acc)
```

```
Baseline top1tier acc 69.72717552512675
```

Fig. 13 Top N Review model Acc

b. <u>N-Gram + Ridge Regression</u>

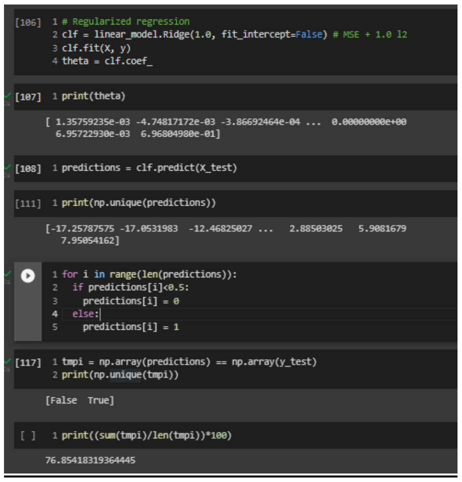
Another baseline model we used is from HW 4, where we learned about predicting ratings using NLP techniques such as Unigram, Bigram, and Ngram. Here, we implemented two of those:

i. <u>Uni-Gram</u>

Uni-gram stands for just picking a single word and predicting the next word, and in turn predicting what the whole text is trying to say using ridge regression.

Before doing any specialized preprocessing, we first split our data into train and test sets with a partition of 80-20. As mentioned before, we followed our basic preprocessing, but instead of lemmatization, we did stemmatization as it works best for language models.

By implementing a naïve unigram model combined with ridge regression activated using a sigmoid function, we achieved an accuracy score of 76.8542%.

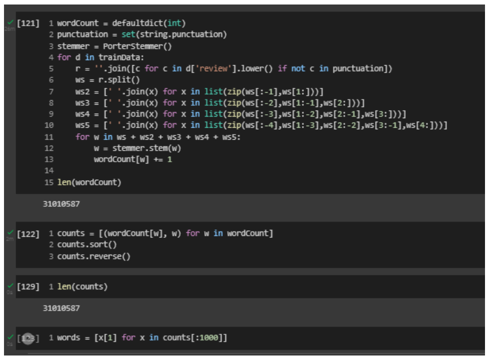


```
[106] # Regularized regression
      clf = linear_model.Ridge(1.0, fit_intercept=False) # MSE + 1.0 l2
      clf.fit(X, y)
      theta = clf.coef_

[107] print(theta)

  [ 1.35759235e-03 -4.74817172e-03 -3.86692464e-04 ...  0.00000000e+00
    6.95722930e-03  6.96804980e-01]

[108] predictions = clf.predict(X_test)

[111] print(np.unique(predictions))

  [-17.25787575 -17.0531983  -12.46825027 ...   2.88503025   5.9081679
     7.95054162]

      for i in range(len(predictions)):
        if predictions[i]<0.5:
          predictions[i] = 0
        else:
          predictions[i] = 1

[117] tmpi = np.array(predictions) == np.array(y_test)
      print(np.unique(tmpi))

  [False  True]

[ ]   print((sum(tmpi)/len(tmpi))*100)

  76.85418319364445
```

Fig. 14 Unigram Accuracy

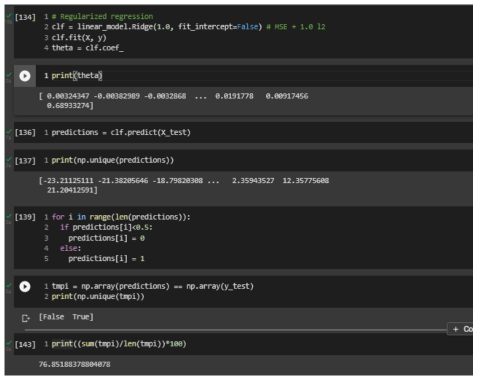
ii. N-Gram

The previous result got us excited to try a more complex language model. Thus, we resorted to an N-gram model, which does the same thing as the unigram but up to five grams.



```
[121] wordCount = defaultdict(int)
      punctuation = set(string.punctuation)
      stemmer = PorterStemmer()
      for d in trainData:
          r = ''.join([c for c in d['review'].lower() if not c in punctuation])
          ws = r.split()
          ws2 = [' '.join(x) for x in list(zip(ws[:-1],ws[1:]))]
          ws3 = [' '.join(x) for x in list(zip(ws[:-2],ws[1:-1],ws[2:]))]
          ws4 = [' '.join(x) for x in list(zip(ws[:-3],ws[1:-2],ws[2:-1],ws[3:]))]
          ws5 = [' '.join(x) for x in list(zip(ws[:-4],ws[1:-3],ws[2:-2],ws[3:-1],ws[4:]))]
          for w in ws + ws2 + ws3 + ws4 + ws5:
              w = stemmer.stem(w)
              wordCount[w] += 1

      len(wordCount)

  31010587

[122] counts = [(wordCount[w], w) for w in wordCount]
      counts.sort()
      counts.reverse()

[129] len(counts)

  31010587

[123] words = [x[1] for x in counts[:1000]]
```

Fig. 15 Ngram Formation



```
[134] # Regularized regression
      clf = linear_model.Ridge(1.0, fit_intercept=False) # MSE + 1.0 l2
      clf.fit(X, y)
      theta = clf.coef_

      print(theta)

  [ 0.00324347 -0.00382989 -0.0032868  ...  0.0191778   0.00917456
    0.68933274]

[136] predictions = clf.predict(X_test)

[137] print(np.unique(predictions))

  [-23.21125111 -21.38205646 -18.79820308 ...   2.35943527  12.35775608
    21.20412591]

[139] for i in range(len(predictions)):
        if predictions[i]<0.5:
          predictions[i] = 0
        else:
          predictions[i] = 1

      tmpi = np.array(predictions) == np.array(y_test)
      print(np.unique(tmpi))

  [False  True]

[143] print((sum(tmpi)/len(tmpi))*100)

  76.85188378804078
```

Fig. 16 N gram Accuracy

We found it weird that the Ngram model gave us a poorer accuracy of 76.85188% compared to the Unigram model, but we think it can be explained by how the steam reviews are usually written. If you manually read them, it is rarely a wordy essay; it is more chat messaging shortcuts filled with keywords in a small body of text. Hence it might explain why unigram and Ngram results are so similar.

Also, we can refine this by removing very common shortcuts and increasing the number of top n anagrams. We will try replacing the shortcuts while implementing future models.

c. Term Frequency- Inverse Document Frequency (TF-IDF)

After trying different baseline models, we thought it would only make sense to start using TFIDF, mainly because we are dealing with text documents. This technique will highlight relevant terms and hence features independent of the dataset's general wording.

The term frequency-inverse document frequency (TF-IDF) is a numerical statistic that indicates how important a word is to a document in a collection or corpus. It is a combination of text frequency and inverse document frequency, which are defined below.

The definition for TFIDF is:

$$TF(t,d) = \frac{number\ of\ times\ t\ appears\ in\ d}{total\ number\ of\ terms\ in\ d}$$

$$IDF(t) = log\frac{N}{1+df}$$

$$TF - IDF(t,d) = TF(t,d) * IDF(t)$$

Please note that this was an introductory descriptive task, that you can see the results in Fig. 11 compared to Fig. 10. After seeing the improvement, we decided to build upon this for future models.

d. TFIDF + Logistic Regression

For this task, we started by removing the languages' shortcut cause we believed it would benefit the TF-IDF model (see Fig. 17). We then created a pipeline of applying logistic regression [4] following TFIDF.

For log reg, it's the same feature vector formation as we have used in our course work, but just replacing the xi with tfidf. Our feature vector looks like:

$$\theta^T X_i = \theta_0 + \theta_1 tfidf_1 + \theta_2 tfidf_2 + ... + \theta_n tfidf_n$$

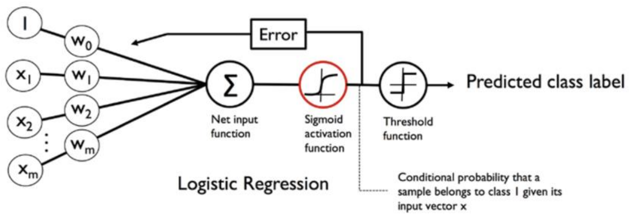
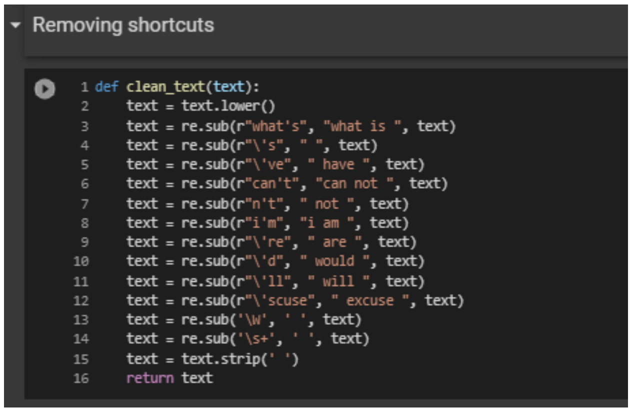


Fig. 17 Log Reg Working

Here, instead of Xi, we will be using the TFIDF of the i th word.

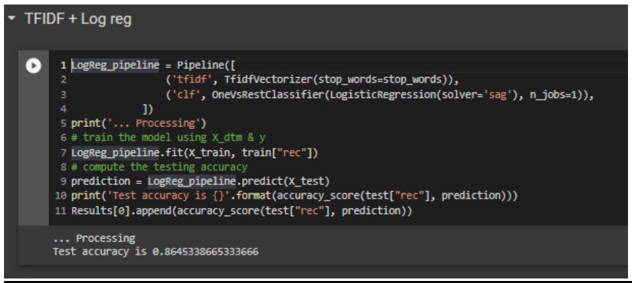


Removing shortcuts

```python
def clean_text(text):
    text = text.lower()
    text = re.sub(r"what's", "what is ", text)
    text = re.sub(r"\'s", " ", text)
    text = re.sub(r"\'ve", " have ", text)
    text = re.sub(r"can't", "can not ", text)
    text = re.sub(r"n't", " not ", text)
    text = re.sub(r"i'm", "i am ", text)
    text = re.sub(r"\'re", " are ", text)
    text = re.sub(r"\'d", " would ", text)
    text = re.sub(r"\'ll", " will ", text)
    text = re.sub(r"\'scuse", " excuse ", text)
    text = re.sub('\W', ' ', text)
    text = re.sub('\s+', ' ', text)
    text = text.strip(' ')
    return text
```

Fig. 18 Removing Shortcuts

Here, we got an accuracy of 86.45%, which is quite impressive for a plain, old, trustworthy log reg model.



TFIDF + Log reg

```python
LogReg_pipeline = Pipeline([
                ('tfidf', TfidfVectorizer(stop_words=stop_words)),
                ('clf', OneVsRestClassifier(LogisticRegression(solver='sag'), n_jobs=1)),
            ])
print('... Processing')
# train the model using X_dtm & y
LogReg_pipeline.fit(X_train, train["rec"])
# compute the testing accuracy
prediction = LogReg_pipeline.predict(X_test)
print('Test accuracy is {}'.format(accuracy_score(test["rec"], prediction)))
Results[0].append(accuracy_score(test["rec"], prediction))
```

```
... Processing
Test accuracy is 0.8645338665333666
```

Fig. 19 TFIDF+LogReg Acc

e. TFIDF + Naïve Bayes

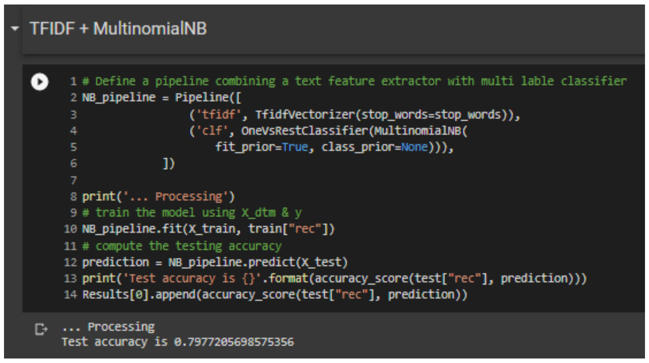
The most predominant probability model that is used to make predictions in ML is by far Naïve Bayes algorithm since it gives us the possibility of calculating the probability of an instance of data belonging to a certain class given our prior knowledge.

NB is also used as a classification algorithm for two class and n class classification problems. The only downside for this model, and hence, why is this described as Naïve is because it makes a very strong assumption that some attributes are conditionally independent, which is something we rarely see in real life data.

$$\underset{\text{Posterior Probability}}{P(c\mid x)} = \frac{\overset{\text{Likelihood}}{P(x\mid c)}\,\overset{\text{Class Prior Probability}}{P(c)}}{\underset{\text{Predictor Prior Probability}}{P(x)}}$$

$$P(c\mid X) = P(x_1\mid c)\times P(x_2\mid c)\times\cdots\times P(x_n\mid c)\times P(c)$$

We then implemented TFIDF with Multinomial Naïve Bayes [6], [7] and got an accuracy of 79.77%.



TFIDF + MultinomialNB

```python
# Define a pipeline combining a text feature extractor with multi lable classifier
NB_pipeline = Pipeline([
                ('tfidf', TfidfVectorizer(stop_words=stop_words)),
                ('clf', OneVsRestClassifier(MultinomialNB(
                    fit_prior=True, class_prior=None))),
            ])

print('... Processing')
# train the model using X_dtm & y
NB_pipeline.fit(X_train, train["rec"])
# compute the testing accuracy
prediction = NB_pipeline.predict(X_test)
print('Test accuracy is {}'.format(accuracy_score(test["rec"], prediction)))
Results[0].append(accuracy_score(test["rec"], prediction))
```

```
... Processing
Test accuracy is 0.7977205698575356
```

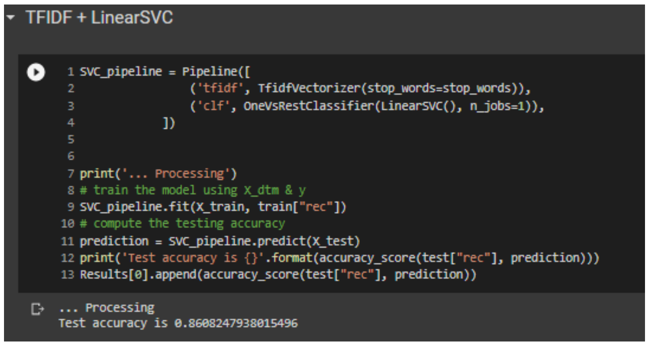
Fig. 20 MNB Accuracy

f. TFIDF + Linear Support Vector Classification

LSVC is a model that builds upon our understanding of SVMs which we briefly covered in class. Basically, it's an algo that attempts to get a hyperplane that maximizes distance between distinctive classes.

$$\min_{w,b} \frac{1}{2} w^T w + C \sum_{i=1} \max(0, y_i(w^T \phi(x_i) + b)).$$

So finally, we tried TFIDF with linear Support Vector Classification [5] and got an accuracy of 86.08%.

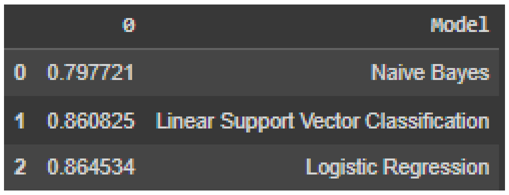


TFIDF + LinearSVC

```python
SVC_pipeline = Pipeline([
                ('tfidf', TfidfVectorizer(stop_words=stop_words)),
                ('clf', OneVsRestClassifier(LinearSVC(), n_jobs=1)),
            ])


print('... Processing')
# train the model using X_dtm & y
SVC_pipeline.fit(X_train, train["rec"])
# compute the testing accuracy
prediction = SVC_pipeline.predict(X_test)
print('Test accuracy is {}'.format(accuracy_score(test["rec"], prediction)))
Results[0].append(accuracy_score(test["rec"], prediction))
```

```
... Processing
Test accuracy is 0.8608247938015496
```

Fig. 21 LSVC Accuracy



	0	Model
0	0.797721	Naive Bayes
1	0.860825	Linear Support Vector Classification
2	0.864534	Logistic Regression

Fig. 22 All Results

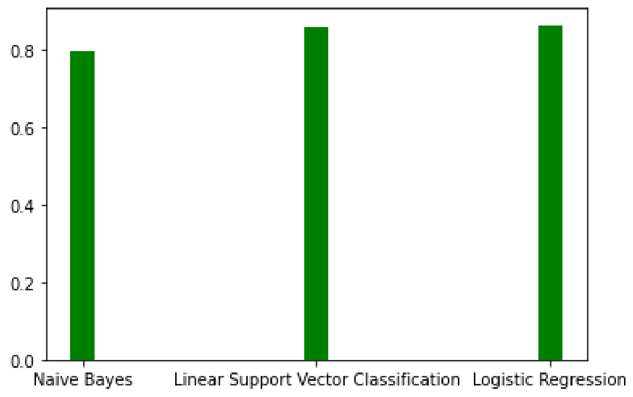


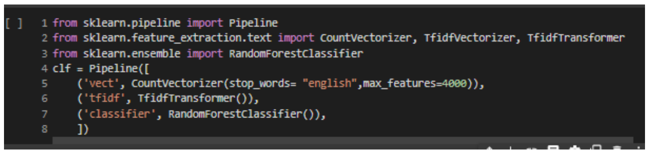
Fig. 23. Bar Plot

Observing the graphs in Fig. 22 and Fig 23. we can see that Log reg gave the best results! Let's try to beat this 0.8645 accuracy by incorporating sentimental analysis into the model!

g. Sentimental Analysis

To perform sentimental analysis, we used the sklearn library. The pipeline we used goes as follows (Fig.24):

- Words to Vectorized form
- TFIDF of all the review text
- Random Forest Classifier



```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer, TfidfTransformer
from sklearn.ensemble import RandomForestClassifier
clf = Pipeline([
    ('vect', CountVectorizer(stop_words= "english",max_features=4000)),
    ('tfidf', TfidfTransformer()),
    ('classifier', RandomForestClassifier()),
    ])
```

Fig. 24 Sentimental Analysis Pipeline

We first split the dataset into 80% train and 20% testing data with a random state of 40.

RF algorithm is a supervised ML algorithm that is extensively used in both Classification and Regression model. The gist of it is that it builds different decision trees using numerous samples and takes majority vote in case of a classification problem and an average vote in case of a regression problem.

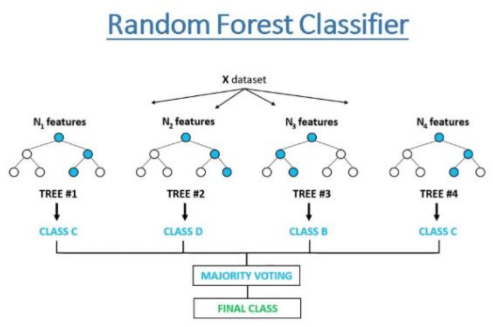
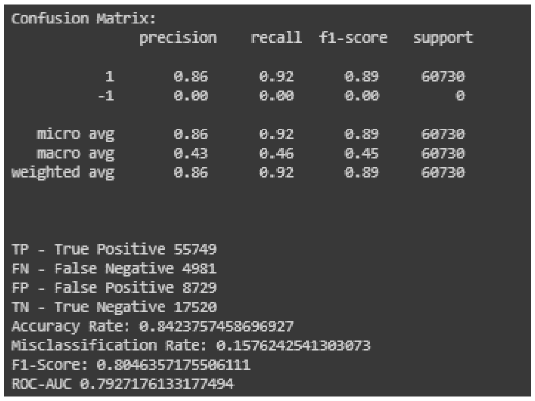


Fig. 25 Random Forest

We ran this train data through the pipeline mentioned above and got the following results:



```
Confusion Matrix:
              precision    recall  f1-score   support

           1       0.86      0.92      0.89     60730
          -1       0.00      0.00      0.00         0

   micro avg       0.86      0.92      0.89     60730
   macro avg       0.43      0.46      0.45     60730
weighted avg       0.86      0.92      0.89     60730


TP - True Positive 55749
FN - False Negative 4981
FP - False Positive 8729
TN - True Negative 17520
Accuracy Rate: 0.8423757458696927
Misclassification Rate: 0.1576242541303073
F1-Score: 0.8046357175506111
ROC-AUC 0.7927176133177494
```

Fig. 26 Sentimental Analysis Results

At first glance, we were impressed by the 84% accuracy but then realized that the ROC-AUC score was not really up to the mark. In addition, if we look at the confusion matrix, we have almost double the number of false positives than false negatives. This indicates that our model was biased in predicting ones.

If you recall (refer Fig. 5), we have a significant imbalance between our dataset's positive and negative samples. This might be overfitting the model on only predicting ones. To fix this, we will now try to balance our data.

One of the solutions that we found for this particular problem was to artificially balance the training set by oversampling and under-sampling the unbalanced data [9].

To accomplish this, we took the help of the python library called imbalanced learning. We used its inbuilt function to fix the misbalancing in our data; let's visualize it (Fig. 27).

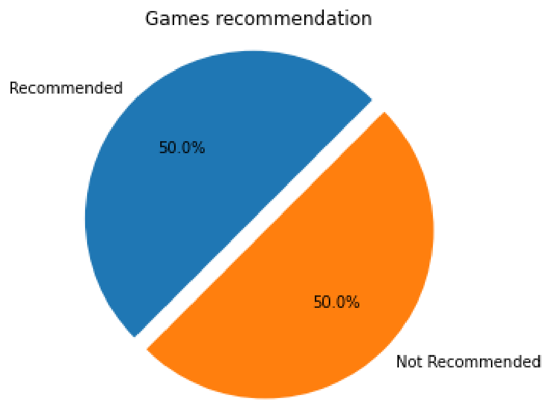
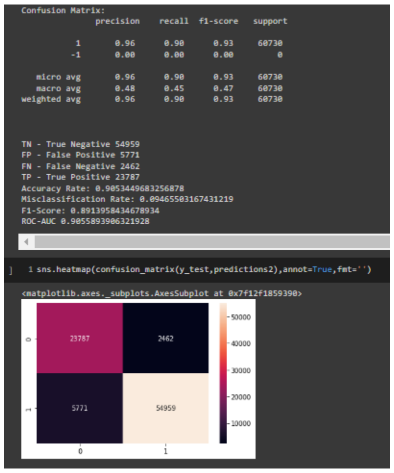


Fig. 27 New Partition of Training set

By killing off the Pacman, we made sure that when we now train this data through the same pipeline, it won't overfit on a single class. Let us look at the new results (Fig. 28).



```
Confusion Matrix:
              precision    recall  f1-score   support

           1       0.96      0.90      0.93     60730
          -1       0.00      0.00      0.00         0

   micro avg       0.96      0.90      0.93     60730
   macro avg       0.48      0.45      0.47     60730
weighted avg       0.96      0.90      0.93     60730



TN - True Negative 54959
FP - False Positive 5771
FN - False Negative 2462
TP - True Positive 23787
Accuracy Rate: 0.9053449683256878
Misclassification Rate: 0.09465503167431219
F1-Score: 0.8913958434678934
ROC-AUC 0.9055893906321928
```

```
sns.heatmap(confusion_matrix(y_test,predictions2),annot=True,fmt='')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f12f1859390>
```

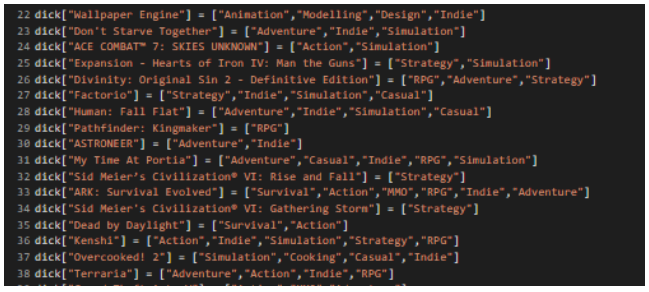
Fig. 28 Results of the balanced model

We got a great result of 90.534 % on our test data. You can see our improvement by balancing our data, as indicated by the high ROC-AUC score.

Let's now begin with task 2.

B- <u>Task 2: Multi-class Genre Prediction.</u>

As mentioned before, our dataset didn't initially contain the genres of each game with it. So, we artificially added them 1by scrapping the steam game store (Fig. 29).



```
dick["Wallpaper Engine"] = ["Animation","Modelling","Design","Indie"]
dick["Don't Starve Together"] = ["Adventure","Indie","Simulation"]
dick["ACE COMBAT™ 7: SKIES UNKNOWN"] = ["Action","Simulation"]
dick["Expansion - Hearts of Iron IV: Man the Guns"] = ["Strategy","Simulation"]
dick["Divinity: Original Sin 2 - Definitive Edition"] = ["RPG","Adventure","Strategy"]
dick["Factorio"] = ["Strategy","Indie","Simulation","Casual"]
dick["Human: Fall Flat"] = ["Adventure","Indie","Simulation","Casual"]
dick["Pathfinder: Kingmaker"] = ["RPG"]
dick["ASTRONEER"] = ["Adventure","Indie"]
dick["My Time At Portia"] = ["Adventure","Casual","Indie","RPG","Simulation"]
dick["Sid Meier's Civilization® VI: Rise and Fall"] = ["Strategy"]
dick["ARK: Survival Evolved"] = ["Survival","Action","MMO","RPG","Indie","Adventure"]
dick["Sid Meier's Civilization® VI: Gathering Storm"] = ["Strategy"]
dick["Dead by Daylight"] = ["Survival","Action"]
dick["Kenshi"] = ["Action","Indie","Simulation","Strategy","RPG"]
dick["Overcooked! 2"] = ["Simulation","Cooking","Casual","Indie"]
dick["Terraria"] = ["Adventure","Action","Indie","RPG"]
```

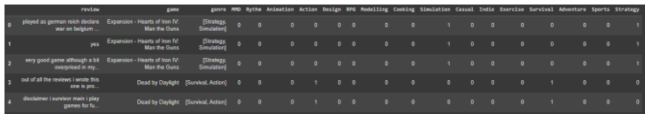
Fig. 29 Sub-Samples of all genres we added

By doing this, we added 16 unique genres to our 48 games and then visualized them, as shown in Fig. 1 and Fig. 2.

To train a multi-class classifier, we utilized the handy OneVsRestClassifier from the Sklearn library [4].

Before applying any of the following models, we followed the preprocessing routine we developed in Task 1.

After basic preprocessing, we applied the One hot encoding technique we learned during our lectures to convert the genres into model friendly 0 and 1s (Fig. 30).



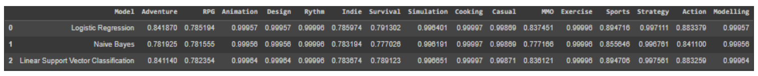
	review	game	genre	MMO	Rythm	Animation	Action	Design	RPG	Modelling	Cooking	Simulation	Casual	Indie	Exercise	Survival	Adventure	Sports	Strategy
0	played as german reich declare war on belgium ...	Expansion - Hearts of Iron IV: Man the Guns	[Strategy, Simulation]	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
1	yes	Expansion - Hearts of Iron IV: Man the Guns	[Strategy, Simulation]	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
2	very good game although a bit overpriced in my...	Expansion - Hearts of Iron IV: Man the Guns	[Strategy, Simulation]	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
3	out of all the reviews i wrote this one is pro...	Dead by Daylight	[Survival, Action]	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
4	disclaimer i survivor main i play games for fu...	Dead by Daylight	[Survival, Action]	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0

Fig. 30 One Hot Encoding

We then proceed to test three models, namely:

- Logistic Regression
- Multinomial Naïve Bayes
- Linear Support Vector Classification

Before applying any of these models, we did an iteration of TFIDF on all the reviews. The results we achieved were terrific (Fig. 31)



	Model	Adventure	RPG	Animation	Design	Rythm	Indie	Survival	Simulation	Cooking	Casual	MMO	Exercise	Sports	Strategy	Action	Modelling
0	Logistic Regression	0.841870	0.785194	0.99957	0.99957	0.99996	0.785974	0.791302	0.996401	0.99997	0.99869	0.837451	0.99996	0.894716	0.997111	0.883379	0.99957
1	Naive Bayes	0.781925	0.781555	0.99956	0.99956	0.99996	0.783194	0.777026	0.996191	0.99997	0.99869	0.777166	0.99996	0.855646	0.996761	0.841100	0.99956
2	Linear Support Vector Classification	0.841140	0.782354	0.99964	0.99964	0.99996	0.783674	0.789123	0.996651	0.99997	0.99871	0.836121	0.99996	0.894706	0.997561	0.883259	0.99964

Fig. 31 Accuracy Per Genre Per Model

If you analyze these results, it can be seen that the most popular genres have an accuracy between 78-90%. In contrast, niche categories like cooking, animation, etc., have a very high accuracy score. This could be due to two reasons:

1) The number of reviews we will have for popular categories will always be much larger than some less popular genres.
2) Genres like cooking, modeling, and animation are more likely to be reflected in the reviews because these categories are very specific. For instance, in any reviews involving NBA2k (basketball game), the reviewer will have to use the word sports or basketball or anything related to that, so the genre is in line with the reviews and that is what the results are showing.

Following are the results of the models as a whole:

Model	Accuracy
Logistic Regression	0.925667958
Naive Bayes	0.91173894
Linear Support Vector Classification	0.925131842

Fig. 32 Accuracy of each model

According to our observation, all three models are relatively equal, but logistic regression scored the highest for the split of data we used.

4. Literature Review

We will now discuss literature reviews that revolve around our dataset.

There are already some models that are built on top of this dataset on Kaggle itself.

One of which was BERT which stands for Bidirectional Encoder Representations from Transformers [2]. This model was a classification task similar to our second task. There are some similarities in the approach and the preprocessing, but the model performed worse than ours. But the article's author said he didn't do enough preprocessing and could've gotten much better results otherwise. But still, it was an excellent read and an interesting approach.

The second one we want to discuss is something we found to be really interesting and built on a dataset similar to ours. It used the new and upcoming technology transformers to do auto-topic modeling [8], which means grouping similar reviews and giving a label to that group. It is a fascinating idea, and the model used was new to us both, so it was a good read and an exciting thing to take away from this.

Another interesting model that was also developed on this dataset was a summary review [4]. Although the model was poorly executed, it was still an interesting idea to do and read.

Further, a lot of the current state-of-the-art models in all of the machine learning space and even more particularly in the multi-class classification and the Recommended/Not Recommended task is most certainly Neural Networks, in all of its models from Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU) and many more RNN or DNN [10].

And as we all may know by now, these models perform exceptionally well, but we decided to implement traditional machine-learning models during this assignment.

Finally, we want to mention our professor Julian's book, titled Personalized Machine Learning [11]. It has really given us a lot of insights and tweaks we can do while applying recommendation algorithms in the real world. In case of some libraries, the book is better documented than the library's own documentation.

5. Results

We have been displaying and analyzing the results as we have implemented them in the Predictive task section itself. Below are all the results in a tabular format.

Predictive Task	Model	Accuracy (%)
Recommended/Not Recommended	Top N most Popular	69.7271
	Unigram	75.8542
	Ngram	76.8518
	TFIDF+Log Reg	86.4533
	TFIDF+Multinomial NB	79.7721
	TFIDF+LSVC	86.0825
	Random Forest (Sentimental Analysis)	84.2375
	Random Forest (Balanced TrainData)	90.5345
Multi Class Genre Prediction	TFIDF + Log Reg	92.5667
	TFIDF + Multinomial NB	91.1738
	TFIDF + LSVC	92.5132

6. References

[1]. Mahendra, L. (2019, April 3). Steam reviews dataset. Kaggle. Retrieved November 26, 2022, from https://www.kaggle.com/datasets/luthfim/steam-reviews-dataset

[2]. Bugonort. (2020, March 30). Steam reviews classifier with bert. Kaggle. Retrieved November 26, 2022, from https://www.kaggle.com/code/bugonort/steam-reviews-classifier-with-bert

[3]. Brownlee, J. (2020, August 19). 4 types of classification tasks in machine learning. MachineLearningMastery.com. Retrieved November 26, 2022, from https://machinelearningmastery.com/types-of-classification-in-machine-learning

[4]. Thomaslazarus. (2022, January 3). Steam reviews - summarizing reviews. Kaggle. Retrieved November 26, 2022, from https://www.kaggle.com/code/thomaslazarus/steam-reviews-summarizing-reviews

[5]. Sklearn.svm.LinearSVC. scikit. (n.d.). Retrieved November 26, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

[6]. Sklearn.naive_bayes.multinomialnb. scikit. (n.d.). Retrieved November 26, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html

[7]. Ismiguzel, I. (2021, September 28). Naive Bayes algorithm for Classification. Medium. Retrieved November 26, 2022, from https://towardsdatascience.com/naive-bayes-algorithm-for-classification-bc5e98bff4d7

[8]. MARJANI, A. (2022, February 3). Steam Reviews - Auto Topic Modeling w Transformers. Kaggle. https://www.kaggle.com/code/dardodel/steam-reviews-auto-topic-modeling-w-transformers

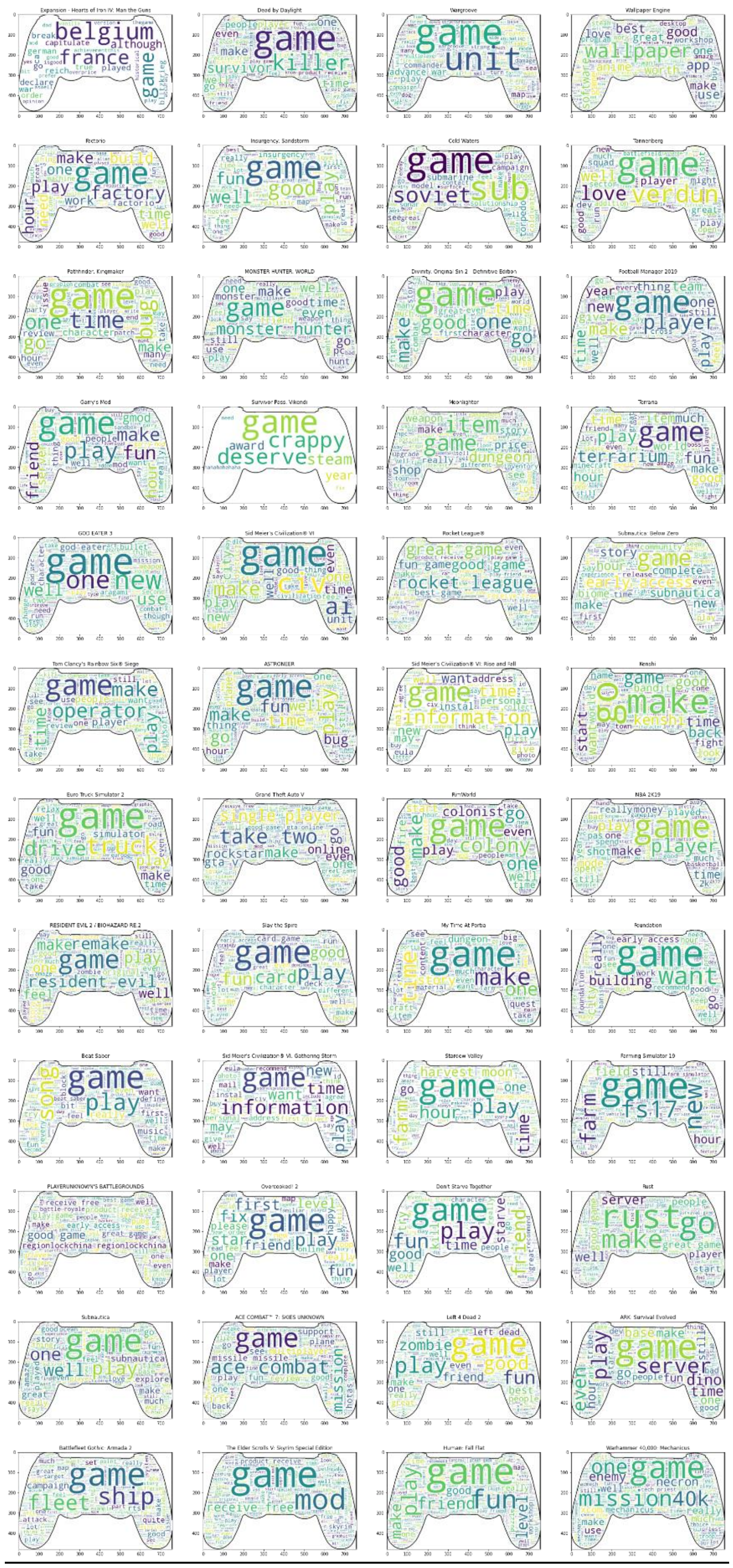
[9]. Nabi, J. (2021, December 7). Machine Learning — Multi-class Classification with Imbalanced Dataset. Medium. https://towardsdatascience.com/machine-learning-multiclass-classification-with-imbalanced-data-set-29f6a177c1a

[10]. Multi-class classification in machine learning. (2022, July 4). DataRobot AI Cloud. https://www.datarobot.com/blog/multiclass-classification-in-machine-learning/

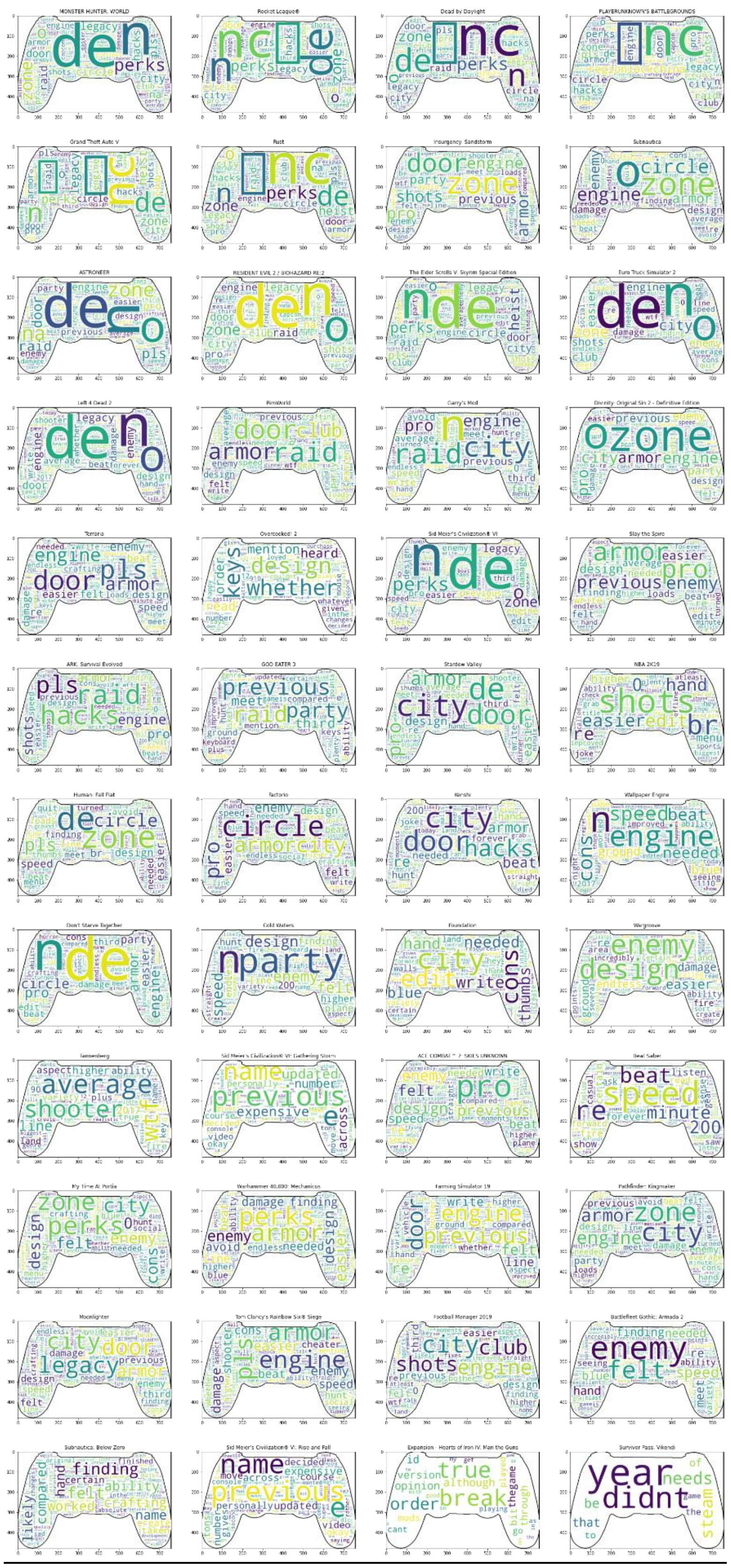
[11]. McAuley, J. (n.d.). Personalized Machine Learning. https://cseweb.ucsd.edu/%7Ejmcauley/pml/

7. Appendix

A- <u>All 48 games wordcloud pre TFIDF.</u>



B- All 48 games wordcloud post TFIDF.


MONSTER HUNTER: WORLD
Rocket League®
Dead by Daylight
PLAYERUNKNOWN'S BATTLEGROUNDS
Grand Theft Auto V
Rust
Insurgency: Sandstorm
Subnautica
ASTRONEER
RESIDENT EVIL 2 / BIOHAZARD RE:2
The Elder Scrolls V: Skyrim Special Edition
Euro Truck Simulator 2
Left 4 Dead 2
RimWorld
Garry's Mod
Divinity: Original Sin 2 - Definitive Edition
Terraria
Overcooked! 2
Sid Meier's Civilization® VI
Slay the Spire
ARK: Survival Evolved
GOD EATER 3
Stardew Valley
NBA 2K19
Human: Fall Flat
Factorio
Kenshi
Wallpaper Engine
Don't Starve Together
Cold Waters
Foundation
Wargroove
Tannenberg
Sid Meier's Civilization® VI: Gathering Storm
ACE COMBAT™ 7: SKIES UNKNOWN
Beat Saber
My Time At Portia
Warhammer 40,000: Mechanicus
Farming Simulator 19
Pathfinder: Kingmaker
Moonlighter
Tom Clancy's Rainbow Six® Siege
Football Manager 2019
Battlefleet Gothic: Armada 2
Subnautica: Below Zero
Sid Meier's Civilization® VI: Rise and Fall
Expansion - Hearts of Iron IV: Man the Guns
Survivor Pass: Vikendi