NLP Document Classification: Cyberpunk 2077 - Sentiment Analysis

Eric Rios Soderman, Wafiakmal Miftah, Zhonglin Wang

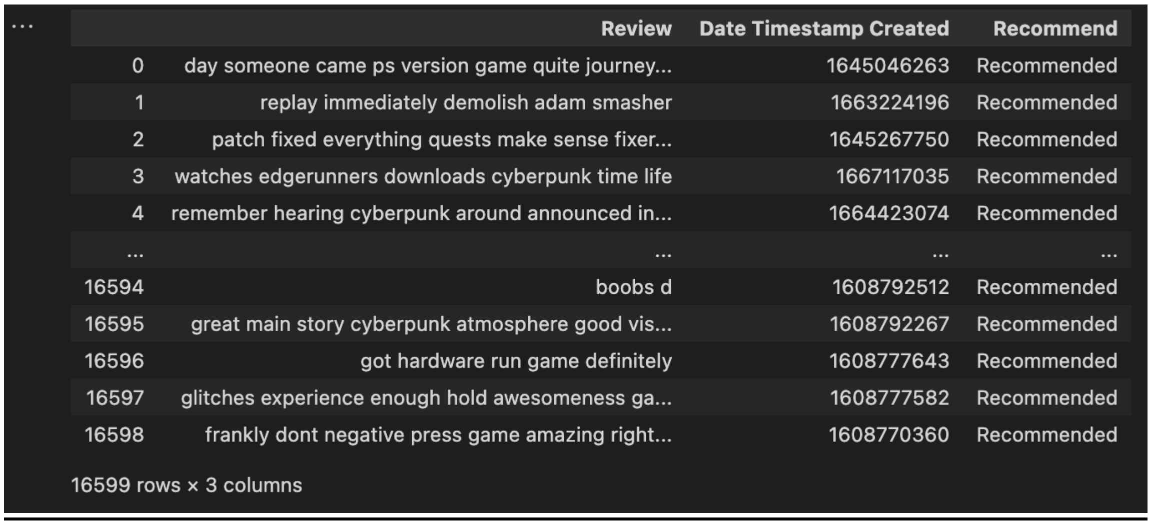
Brief Overview of Tasks and Results (Step 1)

For our group assignment, we had to find a dataset on which we would apply a generative model and a neural network model. Our dataset of interest was a set of the most recent 16,599 reviews on Cyberpunk 2077, which we scraped from the Steam game store on December 8, 2022, by using the Steam API. As for how to scrape the data, we obtained the code from a Medium article regarding review scraping from Steam, (Muller, 2021). Then we conducted data cleaning on the dataset, which was then reduced to 16,460 reviews, classified as 10,964 recommended reviews and 5,496 not recommended reviews. Since our reviews have labels of "Recommended" and "Not Recommended", we decided to construct a generative model based on an approach combining bag of words with the Naive Bayes theorem of conditional probability (Ersoy, 2021) and apply a discriminative model consisting of a bidirectional LSTM neural network model to predict whether a review has a positive sentiment (recommended) or a negative sentiment (not recommended). Our models would be trained on a training set of 67% of the data and tested on 33% of the data, and both of them would later be trained and tested in the same manner on a synthetic dataset created by the generative model.

As for the resulting accuracies, the generative model and the BiLSTM model had overall accuracy scores of 82.36% and 92.2% on a specific run respectively in predicting the labels of the original data. For the BiLSTM, it is important to note that the accuracy fluctuates per run, often oscillating between 80% and 95% after 5 epochs with an embedding size of 64. In the run with 92.2%, it surpassed the generative's model predictive performance during the third epoch. However, when training on the synthetic data, an advantage of the generative model was made apparent. This model achieved 97.20% accuracy at predicting the labels on the training set of the synthetic data. As for the BiLSTM, it achieved 92.8% and did not surpass the generative model's performance. We theorize the main cause behind the performance disparity may have occurred because the BiLSTM considers order and long range dependencies in a bidirectional manner (Dolphin, 2020), whereas our bag of words used a Naive Bayes approach when classifying reviews as well as synthetically generating them. This implies that every review in the synthetic data was a bag of words with no regard to order, much less the concept of a long range, contextual dependency. In addition, during the synthetic generation process, each review used tokens exclusively found within the tokens of its recommended or not recommended class, and there was a unique token difference of 7,000 tokens between the two classes. In this sense, since our synthetic reviews disregard order, long-range, contextual dependencies and contained words obtained randomly from a corpus of a specific class, our generative model was well suited to applying the Naive Bayes theorem to classify the reviews in the test set and gained an advantage over the BiLSTM's affinity for order and handling of long-term memory.

Cyberpunk 2077 Steam Reviews Dataset : Nuanced Preprocessing (Step 2)

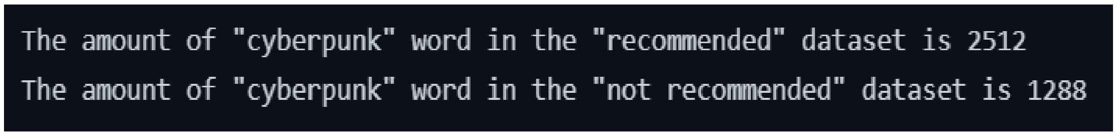
This section will cover the nuances of cleaning and preparing the final dataset, especially moments where we had to exercise a bit of discretion when cleaning. As mentioned previously, the final dataset consisted of 16,460 reviews and 3 columns (the review string, the label, and the date). 10,964 of them were recommended and the remaining 5,496 of them were not recommended. Some reviews had as little as two words while the maximum count of words was 929. In addition, we removed emojis due to their translation into strange sequences of characters and stopwords, knowing that it would affect the grammar during the text generation process of the synthetic data. However, since we considered a bag of words model to both train it and produce the synthetic data from said model, order and sensible English sentences were never going to be accessible under this specific framework. Also, we understood, given our bag of words approach and the nature of user reviews, many words or tokens would be shared across the recommended and not recommended corpora, although with differing counts.



	Review	Date Timestamp Created	Recommend
0	day someone came ps version game quite journey...	1645046263	Recommended
1	replay immediately demolish adam smasher	1663224196	Recommended
2	patch fixed everything quests make sense fixer...	1645267750	Recommended
3	watches edgerunners downloads cyberpunk time life	1667117035	Recommended
4	remember hearing cyberpunk around announced in...	1664423074	Recommended
...	...	...	...
16594	boobs d	1608792512	Recommended
16595	great main story cyberpunk atmosphere good vis...	1608792267	Recommended
16596	got hardware run game definitely	1608777643	Recommended
16597	glitches experience enough hold awesomeness ga...	1608777582	Recommended
16598	frankly dont negative press game amazing right...	1608770360	Recommended

16599 rows × 3 columns

Furthermore, there were two nuanced instances that we considered and addressed before continuing to process the dataset. One was the inclusion of the word "cyberpunk" in any of the reviews. We thought its inclusion would add noise and accuracy reductions to the generative model specifically, due to the use of the word as the cyberpunk genre and use of the word referencing the game's title itself. When utilizing the Naive Bayes theorem, the probability of a word given a class would be sufficiently high for cyberpunk, since it was much more frequent in positive reviews, potentially introducing a positive bias when predicting the sentiment every time the word appeared in a review, whether it was negative or positive.



```
The amount of "cyberpunk" word in the "recommended" dataset is 2512
The amount of "cyberpunk" word in the "not recommended" dataset is 1288
```

As for the second issue, it was the unbalanced dataset, we were worried about the 2 to 1 ratio of positive to negative reviews completely impacting our predictive accuracy for our generative model. Moving forward, we decided to run the sentiment analysis on variations of the dataset with the word "cyberpunk" and without it, as well as balanced with 50 % positive and 50% negative and unbalanced. In addition, we also repeated this process with the synthetic dataset. In contrast, we took no such precautions with the BiLSTM model because we believed it was robust enough in strength, more than a bag of words model, and would be able to leverage its focus on order and long term memory to minimize the impact of these aforementioned issues, so we only took these precautions and executed these variations for the generative model.

Before continuing to the next section, we want to highlight here the common cleaning steps we undertook across both models:

1. Dropping punctuations and numbers using string function
2. Lowercase all words using string function
3. Delete stopwords (is, a, this,...) using NLTK
4. Replacing game title typo to “cyberpunk”
5. Replacing non alphanumeric characters from the review

<u>Generative Model (Step 2)</u>

Naive Bayes : Foundation of Generative Model

P(A|B) = P(B|A) * P(A)/P(B)

The formula above is the backbone of our generative, probabilistic model, which is founded on Naive Bayes. At a micro level, the probability of a word, the current observation, being classified as a certain category (A) (recommended or not recommended) given a certain word (B) (aliased as P(A|B)) is calculated by the probability of a word being from the recommended/not recommended corpus (P(B|A) times the probability of any word being of a certain category (P(A)) divided by the probability of a word being that specific word P(B) (Ersoy, 2021).

P(Recommended|Like) = P(Like|Recommend) * P(Recommended)/P(Word)

Then we multiply the probabilities of all these words to calculate a final score for that review as positive and calculate a score again for that review being negative. We are essentially choosing the maximum value of the two scores, argmax.

P(Not Recommended|Like) = P(Like|Not Recommended) * P(NotRecommended)/P(Word)

Sentiment Label = argmax(Positive Score, Negative Score)

Whichever of the scores is higher determines the final label to be assigned to that review. If the test review has a word that the model, after training, does not know, the model skips the word and does not alter the sentiment score because of the word, a very intentional decision on our part to handle unknown tokens.

After stating all of these details, we recognize that the bag of words model, using Naive Bayes, is naive in the sense that it disregards order and assumes every word's appearance as independent from each other, even though words often appear together, as if simulating some sort of codependency.

Brief Overview on Implementation

As for the implementation, we created multiple bags of words to create recommended and not recommended corpora to train the model.

- Frequencies of the tokens (2 corpuses, Recommended & Not Recommended)
- Counts of the reviews that the word appears in (2 corpuses, Recommended & Not Recommended)
- Frequencies of all words (One corpus)

The frequencies of the tokens were stored in the recommended and not recommended corpora. If "love" appeared 20 times in recommended reviews and 5 times in not recommended reviews, 20 would be stored in the recommended corpora, and 5 would be stored in the not recommended corpora, and then both would be converted to probabilities by normalizing the counts. The same concept or idea applies for the corpora of the counts of the reviews in which the word appears at least once, which becomes important later on as we intend to apply penalties. Lastly, the frequencies of all words were stored in one corpus regardless of label. From the first and last bag of words, we could calculate all the pieces of the Naive Bayes Theorem. However, we wanted to experiment with penalties, the next section.

Developing Custom Penalties Per Word

$P(A|B) = P(B|A) * P(A)/P(B)$ * Custom TF * Custom IDF
Absolute value of $P(A|B) = P(B|A) * P(A)/P(B)$ * Custom TF
Absolute value of $P(A|B) = P(B|A) * P(A)/P(B)$ * Custom IDF
Absolute value of $P(A|B) = P(B|A) * P(A)/P(B)$ * TF * IDF
Absolute value of $P(A|B) = P(B|A) * P(A)/P(B)$ (Original Naive Bayes)

When opting for the Naive Bayes Theorem, one of the concerns was having overly frequent words skew the predicted reviews towards positive. We also wanted to give more predictive weight to the words that were least frequent in the reviews of the labeled corpus, such as having “atmospheric” or “awesome” be more impactful than “like” or “enjoy” or “game”. One of the ways we sought to handle that was inspired by TF-IDF. The term frequency (TF) looks at the log of the raw counts of a word plus 1 divided by all the words in the corpus, and the inverse document frequency (IDF) is the number of total documents divided by the number of documents in which a given word appears in once (Hamdaoui, 2019). For our case, we decided to experiment and compare custom forms of penalties in the hopes of improving accuracy, departing from the original definitions of the TF-IDF concepts. Our TF deployment was the log of the probability of the word given a label, which is a value that varied per corpus. For the IDF, it is the log of the total number of documents was the reviews in the corpus of a specific label divided by the number of those labeled reviews in which the word appears, so we treated each of the two corpora’s inner contents as the numerators, the number of labeled, review documents, instead of treating the corpora as the documents, which would have numbered as two and produced the same generic penalty for any word that appeared in one or both of the documents. The results of these hypotheses will be shown later.

<u>Neural Network (Discriminative) (Step 3)</u>

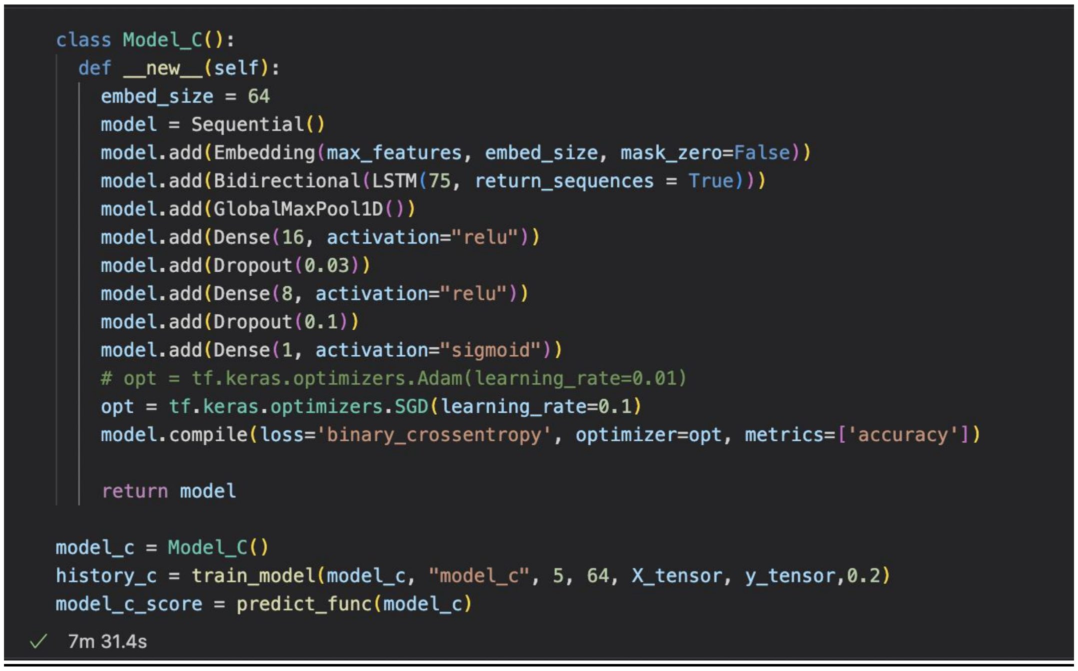
We choose bi-directional LSTM since traditional neural networks suffer from two problems: vanishing gradient and short-term memory. Second, we can use multiple word string sequences to retain key information to classify the categories of our input reviews, due to the long-term memory benefit of the LSTM model. In addition, a bi-directional LSTM considers order from two directions and does simply treat a string like a bag of words. For example, it would be expected to handle the complexity of a review like “It is buggy and sometimes sucking, but Cybepunk 2077 is still a pretty decent game”. A bag of words model might simply tag that review as not recommended due to “buggy”, “sometimes”, and “sucking”. The bi-directional LSTM has forget gates to lose the chunks of the data that are useless to determine sentiment. Also, one of the most salient reasons for a bidirectional LSTM is its bidirectionality. It is able to combine the scores of reading the inputs from both directions to determine a final label. In other words, it is able to account for the variation of having complex sentences full of beginning positive or negative statements that end with the opposite sentiment, such as the example referenced previously about “buggy” and “decent”. In other words, it can help us classify the reviews thanks to how it captures that part of the natural language. After going through appropriate dense layers and dropout layers, our model can find out the most accurate output classification and input meanings.

Overview of Implementation (Step 3)

To implement a BiLSTM, we sourced our entire approach from medium (Hamdaoui, 2019), and slightly modified as necessary. As for our neural network it is discriminative in nature since every review was input as X and the label of recommendation is binary and treated as Y, and the neural network maps X to Y, and the eventual predictions are a binary label of recommendation (sentiment prediction from the text). Then we split the dataset into a train set and a test set with a ratio of 0.33. Furthermore, we added an additional validation set while training the neural network. The ratio for training and validation is 0.2.

Neural Network Structure

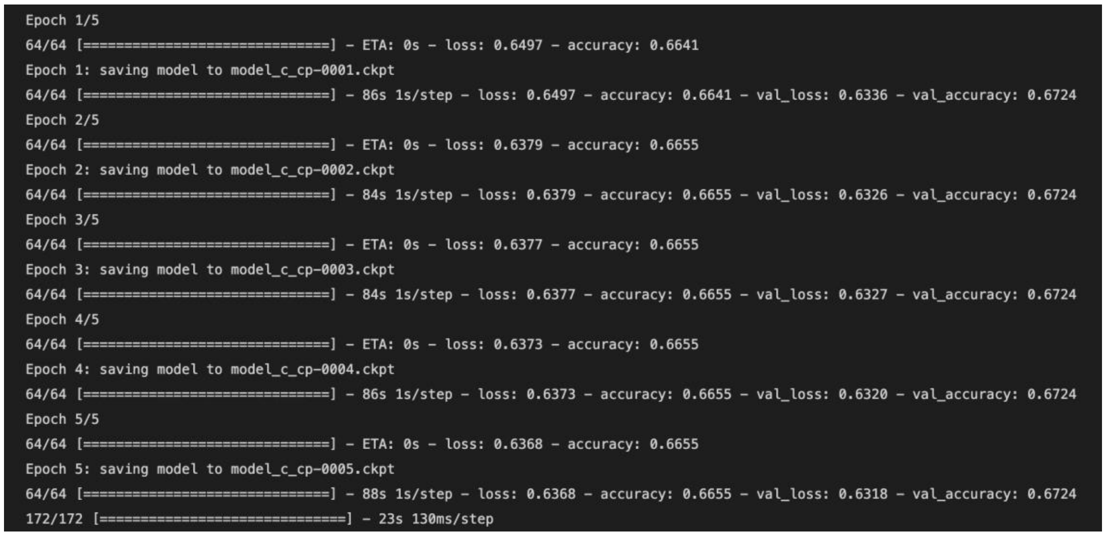
SGD model (not the final model we use, debugged below):



```
class Model_C():
  def __new__(self):
    embed_size = 64
    model = Sequential()
    model.add(Embedding(max_features, embed_size, mask_zero=False))
    model.add(Bidirectional(LSTM(75, return_sequences = True)))
    model.add(GlobalMaxPool1D())
    model.add(Dense(16, activation="relu"))
    model.add(Dropout(0.03))
    model.add(Dense(8, activation="relu"))
    model.add(Dropout(0.1))
    model.add(Dense(1, activation="sigmoid"))
    # opt = tf.keras.optimizers.Adam(learning_rate=0.01)
    opt = tf.keras.optimizers.SGD(learning_rate=0.1)
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

    return model

model_c = Model_C()
history_c = train_model(model_c, "model_c", 5, 64, X_tensor, y_tensor,0.2)
model_c_score = predict_func(model_c)
```

✓ 7m 31.4s

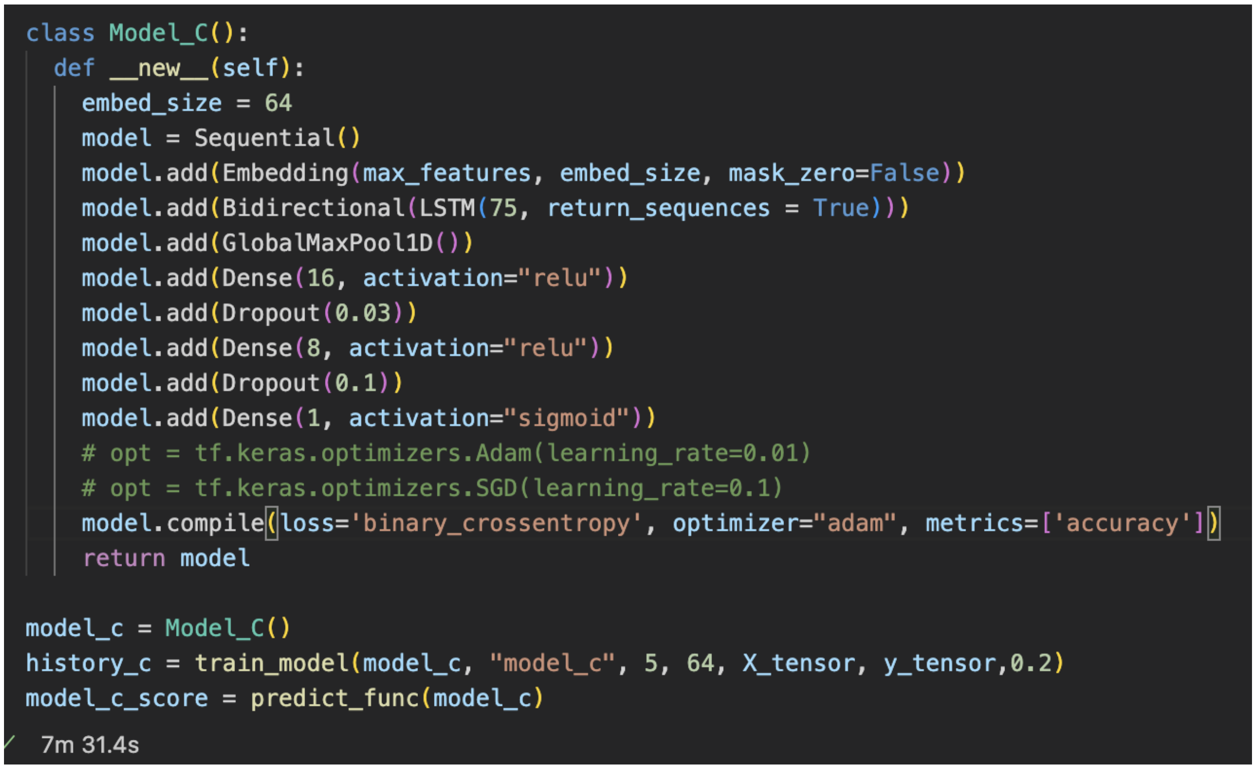
We use a bidirectional LSTM as our model and add Multilayer Perceptron (MLP) for binary classification. The first layer is the input layer, and it is followed by a bidirectional LSTM model. We set the parameter “return_sequences” to be True. We have two hidden layers: the first takes 16 units and the second takes 8. We used the function “pad_sequences” to generate equal-length reviews as our single input. Each individual input is a vector with 929 entries, and each entry is an integer representing the frequency rank of a certain word. We used “relu” as our activation function for each hidden layer and used “sigmoid” as the activation of the output layer. We then proceeded to use 5 epochs and 64 for embed_size to train our model. We used “binary_crossentropy” loss parameter and “sgd” (Stochastic gradient descent) optimizer (learning rate = 0.01) to train the model. We found the loss does not decrease and the model always stayed in a low performance:



```
Epoch 1/5
64/64 [==============================] - ETA: 0s - loss: 0.6497 - accuracy: 0.6641
Epoch 1: saving model to model_c_cp-0001.ckpt
64/64 [==============================] - 86s 1s/step - loss: 0.6497 - accuracy: 0.6641 - val_loss: 0.6336 - val_accuracy: 0.6724
Epoch 2/5
64/64 [==============================] - ETA: 0s - loss: 0.6379 - accuracy: 0.6655
Epoch 2: saving model to model_c_cp-0002.ckpt
64/64 [==============================] - 84s 1s/step - loss: 0.6379 - accuracy: 0.6655 - val_loss: 0.6326 - val_accuracy: 0.6724
Epoch 3/5
64/64 [==============================] - ETA: 0s - loss: 0.6377 - accuracy: 0.6655
Epoch 3: saving model to model_c_cp-0003.ckpt
64/64 [==============================] - 84s 1s/step - loss: 0.6377 - accuracy: 0.6655 - val_loss: 0.6327 - val_accuracy: 0.6724
Epoch 4/5
64/64 [==============================] - ETA: 0s - loss: 0.6373 - accuracy: 0.6655
Epoch 4: saving model to model_c_cp-0004.ckpt
64/64 [==============================] - 86s 1s/step - loss: 0.6373 - accuracy: 0.6655 - val_loss: 0.6320 - val_accuracy: 0.6724
Epoch 5/5
64/64 [==============================] - ETA: 0s - loss: 0.6368 - accuracy: 0.6655
Epoch 5: saving model to model_c_cp-0005.ckpt
64/64 [==============================] - 88s 1s/step - loss: 0.6368 - accuracy: 0.6655 - val_loss: 0.6318 - val_accuracy: 0.6724
172/172 [==============================] - 23s 130ms/step
```

We then pivoted to using the “adam” optimizer to replace the “sgd” optimizer. The “adam” is a replacement optimization algorithm for stochastic gradient descent. It is also a gradient descent process (Brownlee, 2017).

Adam model (our final model):



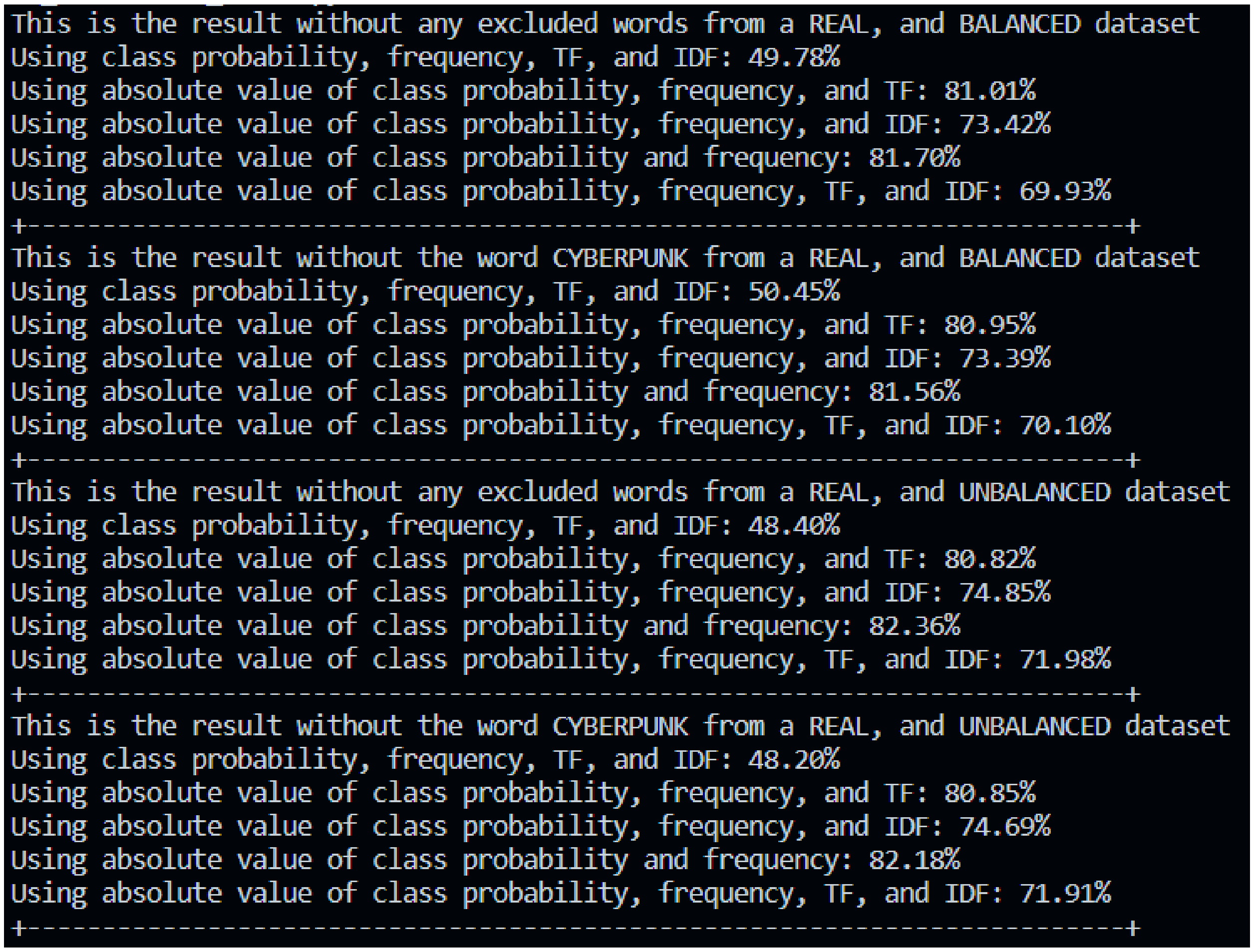
```
class Model_C():
  def __new__(self):
    embed_size = 64
    model = Sequential()
    model.add(Embedding(max_features, embed_size, mask_zero=False))
    model.add(Bidirectional(LSTM(75, return_sequences = True)))
    model.add(GlobalMaxPool1D())
    model.add(Dense(16, activation="relu"))
    model.add(Dropout(0.03))
    model.add(Dense(8, activation="relu"))
    model.add(Dropout(0.1))
    model.add(Dense(1, activation="sigmoid"))
    # opt = tf.keras.optimizers.Adam(learning_rate=0.01)
    # opt = tf.keras.optimizers.SGD(learning_rate=0.1)
    model.compile(loss='binary_crossentropy', optimizer="adam", metrics=['accuracy'])
    return model

model_c = Model_C()
history_c = train_model(model_c, "model_c", 5, 64, X_tensor, y_tensor,0.2)
model_c_score = predict_func(model_c)
✓ 7m 31.4s
```

Here is the final model we used. We kept the structure and all parameters the same as in the SGD model. The only difference is the changing of the “sgd” optimizer to “adam” optimizer. Since the “adam” optimizer takes a self-adaptive learning rate, we didn’t specify the learning rate in the model. During this instance, we got a decent training result on both the original dataset and on the synthetic dataset, which will be referenced later. Now, we will look at the results for both models.

Comparing the Models’ Performance on Real Data(Step 4)

Generative Model Results (Step 4)

Legend (Class probability : P(A), Frequency : (P(B|A), TF : Custom, IDF : Custom)
1st Row : TF * IDF penalty,
2nd Row : Abs value, TF penalty
3rd Row: Abs value, IDF penalty
4th Row: Abs value, Naive Bayes Alone
5th row : Abs value, TF * IDF penalty

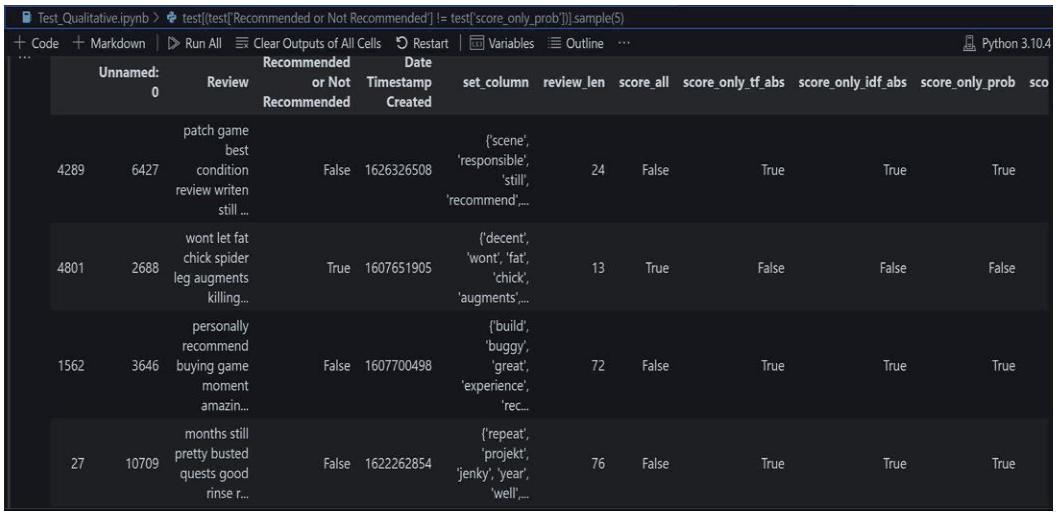


```
This is the result without any excluded words from a REAL, and BALANCED dataset
Using class probability, frequency, TF, and IDF: 49.78%
Using absolute value of class probability, frequency, and TF: 81.01%
Using absolute value of class probability, frequency, and IDF: 73.42%
Using absolute value of class probability and frequency: 81.70%
Using absolute value of class probability, frequency, TF, and IDF: 69.93%
+--------------------------------------------------------------------+
This is the result without the word CYBERPUNK from a REAL, and BALANCED dataset
Using class probability, frequency, TF, and IDF: 50.45%
Using absolute value of class probability, frequency, and TF: 80.95%
Using absolute value of class probability, frequency, and IDF: 73.39%
Using absolute value of class probability and frequency: 81.56%
Using absolute value of class probability, frequency, TF, and IDF: 70.10%
+--------------------------------------------------------------------+
This is the result without any excluded words from a REAL, and UNBALANCED dataset
Using class probability, frequency, TF, and IDF: 48.40%
Using absolute value of class probability, frequency, and TF: 80.82%
Using absolute value of class probability, frequency, and IDF: 74.85%
Using absolute value of class probability and frequency: 82.36%
Using absolute value of class probability, frequency, TF, and IDF: 71.98%
+--------------------------------------------------------------------+
This is the result without the word CYBERPUNK from a REAL, and UNBALANCED dataset
Using class probability, frequency, TF, and IDF: 48.20%
Using absolute value of class probability, frequency, and TF: 80.85%
Using absolute value of class probability, frequency, and IDF: 74.69%
Using absolute value of class probability and frequency: 82.18%
Using absolute value of class probability, frequency, TF, and IDF: 71.91%
+--------------------------------------------------------------------+
```

For the results, they are nothing short of impressive. We trained the model and tested it on the test data by having it predict the test data's labels, which we would later compare to the test data's true labels. We ran the tests for the variations of the datasets with and without "cyberpunk" in the reviews and with and without a balanced dataset.

One of the lessons we learned first was how too many numpy float products can cause a type of overflow where the sign of the product changes to the opposite sign (Eubank, 2022). This gave us many initial negative scores, which hurt our accuracy. Then we learned to use absolute values of the scores to solve this issue, and the results were surprising. The generative model's accuracy actually suffered by the inclusion of the penalties, and it makes logical sense. Our custom use of the TF and IDF concepts were originally meant to help penalize more the words that were common across documents and penalize less the words that are less frequent in said documents. Our custom term frequency was always one or two percentage points lower than the Naive Bayes accuracy and was a log version of the probability of the word given the class. As for our IDF, it penalized overly frequent words in each of our two corpora of recommended and not recommended reviews and generally caused a between 10% and 12% deficit in accuracy compared to the Naive Bayes model's accuracy. We think the reason is due to how the IDF, in its original conception, is meant to differentiate the weight of words across inherently different documents, such as comparing words from a medical document to a news document, but our documents are all reviews that are not profoundly different from each other. In addition, there is also the very strong possibility that our penalties strongly punished the weights of our more important words, if they happened to be somewhat frequent. In other words, maybe "amazing" was weighted less than a word such as "cool" with lesser contextual importance, but more relevant since it was penalized in a lighter way.

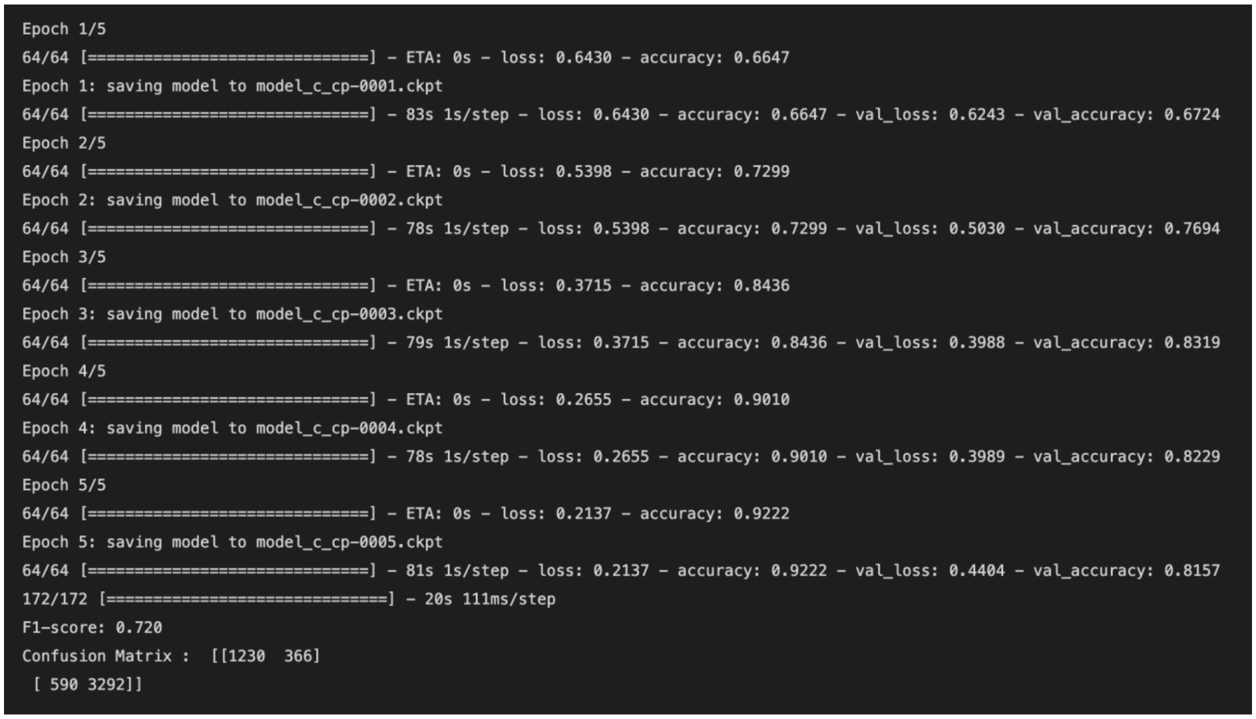
In addition, we also gave a look at the nature of the errors or the ill classified labels. They all fall prey to the fact that the words are shared across both corpora of recommended and not recommended corpuses. For example, "personally recommend buying game moment amazing" are categorized as not recommended as the truth label, but were misclassified as recommended (score_only_prob column). This is due to the words being shared in both corpora, but having a much higher frequency in the recommended corpora. Since these words came from the not recommended corpus, they were definitely used as negation, such as “personally don’t recommend buying this game, not amazing, not an amazing moment”. The same phenomena repeats itself with the other recommended category in the example with “wont let fat chick spider leg augments killing”, where it was misclassified as negative. Like the previous example, the words were shared across both corpora, but held higher weight from the not recommended corpus.


Test_Qualitative.ipynb > test[(test['Recommended or Not Recommended'] != test['score_only_prob'])].sample(5)

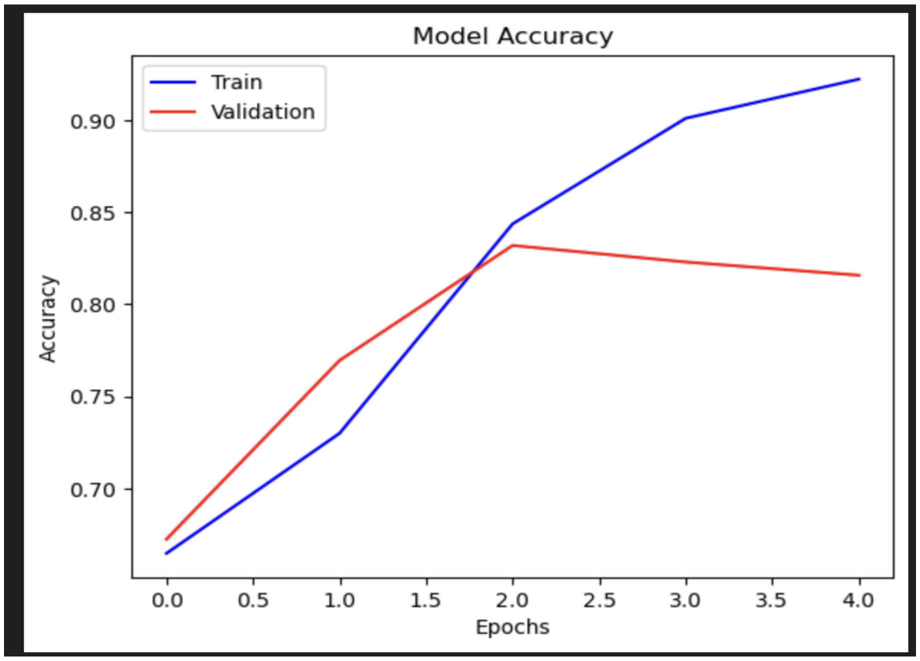
	Unnamed: 0	Review	Recommended or Not Recommended	Date Timestamp Created	set_column	review_len	score_all	score_only_tf_abs	score_only_idf_abs	score_only_prob	sco
4289	6427	patch game best condition review writen still ...	False	1626326508	{'scene', 'responsible', 'still', 'recommend',...	24	False	True	True	True	
4801	2688	wont let fat chick spider leg augments killing...	True	1607651905	{'decent', 'wont', 'fat', 'chick', 'augments',...	13	True	False	False	False	
1562	3646	personally recommend buying game moment amazin...	False	1607700498	{'build', 'buggy', 'great', 'experience', 'rec...	72	False	True	True	True	
27	10709	months still pretty busted quests good rinse r...	False	1622262854	{'repeat', 'projekt', 'jenky', 'year', 'well',...	76	False	True	True	True	

Lastly for this section, it would be remiss to not note the discovery of the variations. Excluding the word cyberpunk from the dataset, contrary to our hypothesis, slightly reduced accuracy across almost all of our penalty variations, meaning that the word does not add noise to the model, but improves it. Equally contrary, our unbalanced dataset yielded better accuracy than using a balanced one by a slight margin. We thought the gap would have been large and in favor of the balanced dataset, but the exact opposite occurred. Remarkably, this informed us that the synthetic data would be generated in the image of the unbalanced dataset with the word cyberpunk included in its reviews.

(BiLSTM Results (Step 4)



```
Epoch 1/5
64/64 [==============================] - ETA: 0s - loss: 0.6430 - accuracy: 0.6647
Epoch 1: saving model to model_c_cp-0001.ckpt
64/64 [==============================] - 83s 1s/step - loss: 0.6430 - accuracy: 0.6647 - val_loss: 0.6243 - val_accuracy: 0.6724
Epoch 2/5
64/64 [==============================] - ETA: 0s - loss: 0.5398 - accuracy: 0.7299
Epoch 2: saving model to model_c_cp-0002.ckpt
64/64 [==============================] - 78s 1s/step - loss: 0.5398 - accuracy: 0.7299 - val_loss: 0.5030 - val_accuracy: 0.7694
Epoch 3/5
64/64 [==============================] - ETA: 0s - loss: 0.3715 - accuracy: 0.8436
Epoch 3: saving model to model_c_cp-0003.ckpt
64/64 [==============================] - 79s 1s/step - loss: 0.3715 - accuracy: 0.8436 - val_loss: 0.3988 - val_accuracy: 0.8319
Epoch 4/5
64/64 [==============================] - ETA: 0s - loss: 0.2655 - accuracy: 0.9010
Epoch 4: saving model to model_c_cp-0004.ckpt
64/64 [==============================] - 78s 1s/step - loss: 0.2655 - accuracy: 0.9010 - val_loss: 0.3989 - val_accuracy: 0.8229
Epoch 5/5
64/64 [==============================] - ETA: 0s - loss: 0.2137 - accuracy: 0.9222
Epoch 5: saving model to model_c_cp-0005.ckpt
64/64 [==============================] - 81s 1s/step - loss: 0.2137 - accuracy: 0.9222 - val_loss: 0.4404 - val_accuracy: 0.8157
172/172 [==============================] - 20s 111ms/step
F1-score: 0.720
Confusion Matrix :  [[1230  366]
 [ 590 3292]]
```



In this part, we surprisingly find that we can obtain around 0.85-0.95 accuracy with only 5 epochs and 64 embed-size, and the accuracy on the training set will increase up to 0.98-0.99 if we use 10 epochs and a 128 embed-size instead. Unfortunately, the training will require more time as a trade-off and probably drifts into a bit of overfitting.

Therefore, we decided to use smaller training epochs and embed-size. It is also worth noticing that the predictive accuracy of our model is fluctuating in a range from 0.80-0.95 (with 5 epochs and 64 embed-size) because we didn’t set a random seed when splitting the training set and validation set. Despite this, the BiLSTM model overall converged quickly and had a relatively good performance on predicting the recommendation label. The model works well when a review is long enough (929 - the maximum of words in reviews in our dataset) for the model to determine the sentiment, and it will work poorly if a review is very short and devoid of sentiment.

From these results, we can conclude that considering order when classifying strings was more effective than simply using a bag of words approach.

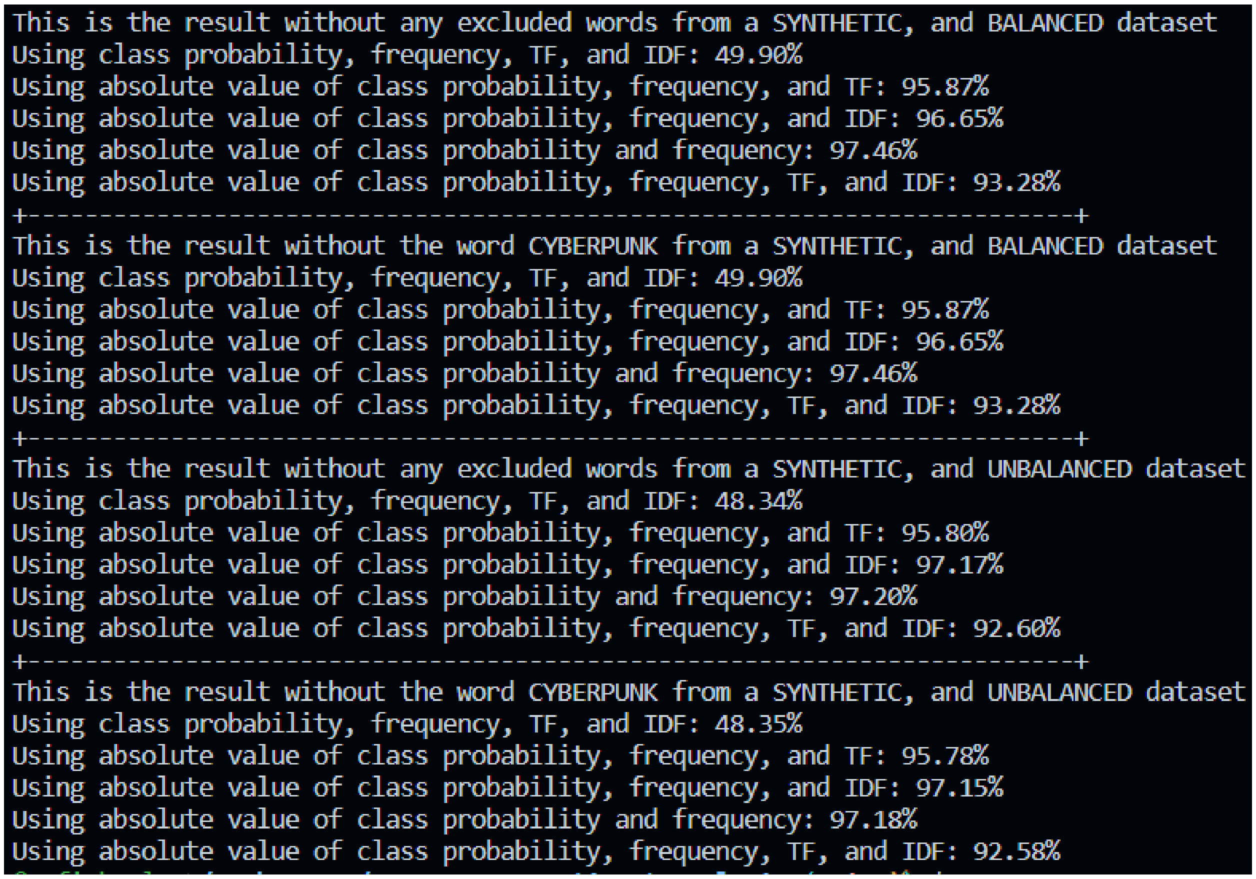
Generating and Training/Testing Synthetic Data (Step 5)

The Synthesization Process

It is important to specify that the generative model trained on the unbalanced dataset without removing the string “cyberpunk” was the final chosen iteration for both the project, comparison of the models, and the synthesis of new data. Because the model was a bag of words, the English reviews were going to lack much grammatical sense, but would utilize words chosen randomly from the recommended or not recommended corpora depending on the corresponding label of the review to create. In other words, if the model was going to generate a positive review, it would use words from the recommended corpus. In addition, it is important to note that the synthetic was created to mimic the shape of the image of the original and cleaned dataset. This is best illustrated with an example. In a set of 10 reviews, where 4 of them are negative, and 6 of them are positive, the synthetic will also produce 4 and 6 negative and positive reviews respectively. In addition, the same idea held true for the review lengths. In a set of 4 reviews, if 1 has 4 words, if 2 have 10 words, and 1 has 3 words, the synthetic review will mimic that exact distribution of word lengths, meaning 1 review with 4 words, 2 reviews with 10 words, and 1 review with 3 words.

Following this step, the models were trained on the synthetic dataHere are the results of training the generative and BiLSTM models on the synthetic data. Now we wil discuss their results.

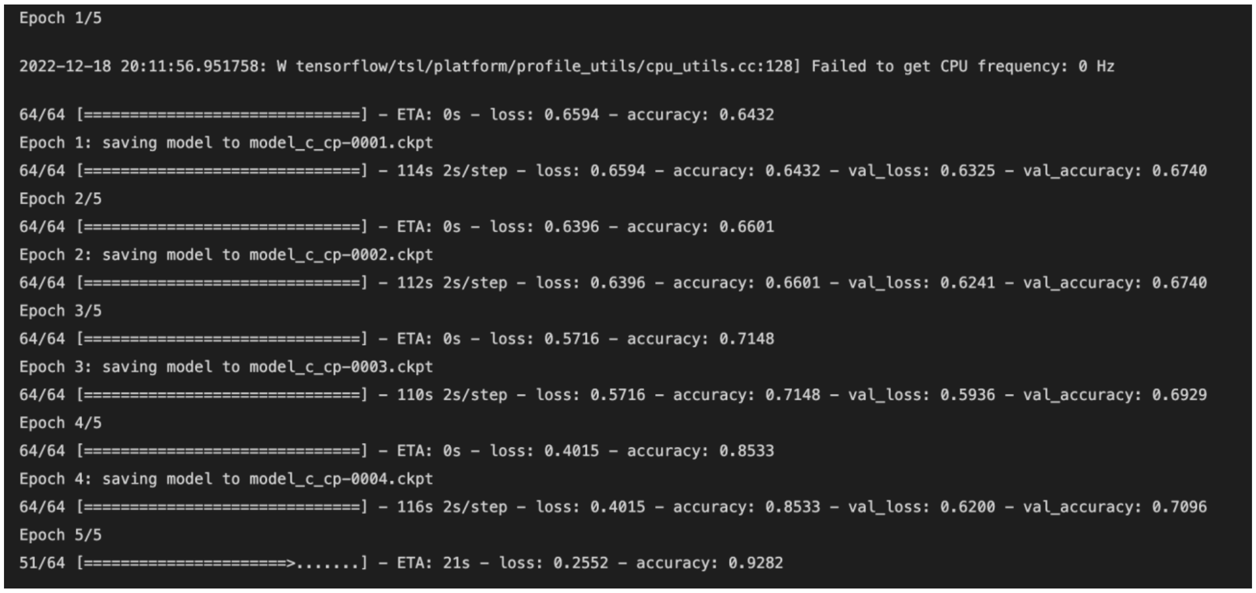
Generative Model:



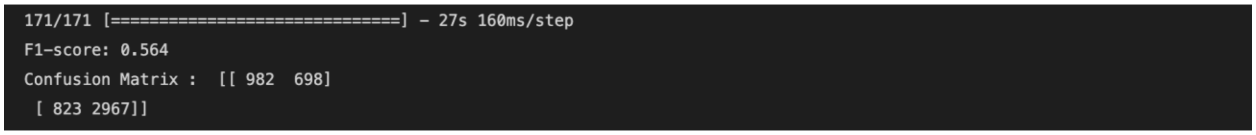
```
This is the result without any excluded words from a SYNTHETIC, and BALANCED dataset
Using class probability, frequency, TF, and IDF: 49.90%
Using absolute value of class probability, frequency, and TF: 95.87%
Using absolute value of class probability, frequency, and IDF: 96.65%
Using absolute value of class probability and frequency: 97.46%
Using absolute value of class probability, frequency, TF, and IDF: 93.28%
+-------------------------------------------------------------------+
This is the result without the word CYBERPUNK from a SYNTHETIC, and BALANCED dataset
Using class probability, frequency, TF, and IDF: 49.90%
Using absolute value of class probability, frequency, and TF: 95.87%
Using absolute value of class probability, frequency, and IDF: 96.65%
Using absolute value of class probability and frequency: 97.46%
Using absolute value of class probability, frequency, TF, and IDF: 93.28%
+-------------------------------------------------------------------+
This is the result without any excluded words from a SYNTHETIC, and UNBALANCED dataset
Using class probability, frequency, TF, and IDF: 48.34%
Using absolute value of class probability, frequency, and TF: 95.80%
Using absolute value of class probability, frequency, and IDF: 97.17%
Using absolute value of class probability and frequency: 97.20%
Using absolute value of class probability, frequency, TF, and IDF: 92.60%
+-------------------------------------------------------------------+
This is the result without the word CYBERPUNK from a SYNTHETIC, and UNBALANCED dataset
Using class probability, frequency, TF, and IDF: 48.35%
Using absolute value of class probability, frequency, and TF: 95.78%
Using absolute value of class probability, frequency, and IDF: 97.15%
Using absolute value of class probability and frequency: 97.18%
Using absolute value of class probability, frequency, TF, and IDF: 92.58%
```

The model is showing signs of overfitting due to the extremely accurate rate, which makes sense because they are a bag of words, and our model is based on a bag of words model. The metrics were slightly better in a balanced configuration of the trained dataset, but removing “cyberpunk” produced no meaningful change in performance. Similar as before, the balanced iteration with the inclusion of “cyberpunk” seems the most optimal, given the sufficiently high accuracy with simply using Naive Bayes. In contrast, the most shocking factor was how well the other penalty variations performed in terms of accuracy, when they were much weaker upon training with real data.

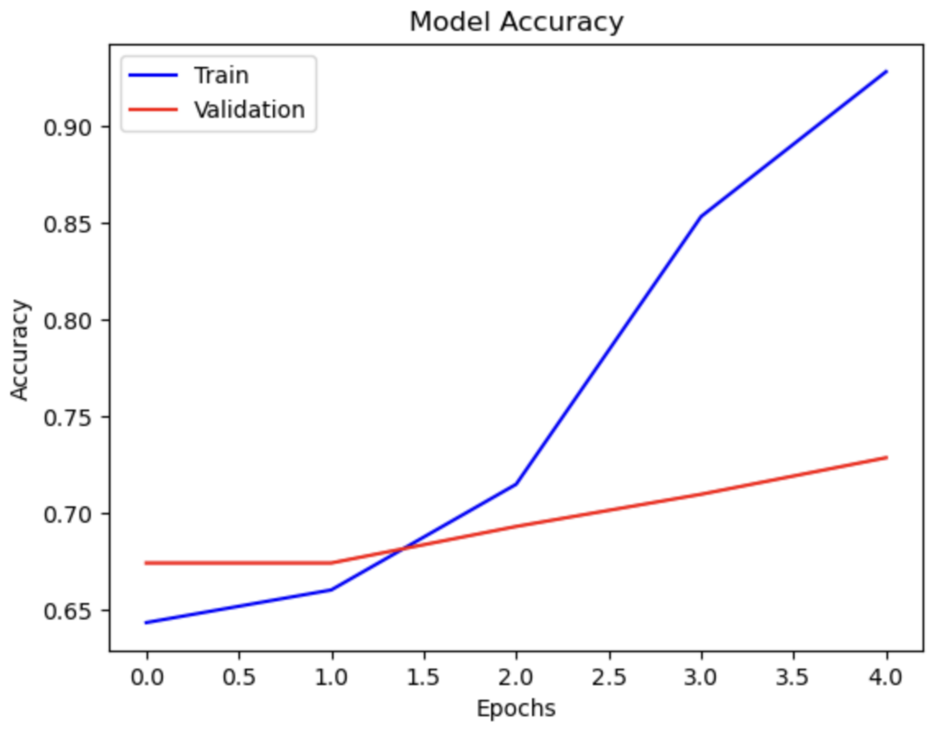
BiLSTM Model (Synthetic dataset)



```
Epoch 1/5

2022-12-18 20:11:56.951758: W tensorflow/tsl/platform/profile_utils/cpu_utils.cc:128] Failed to get CPU frequency: 0 Hz

64/64 [==============================] - ETA: 0s - loss: 0.6594 - accuracy: 0.6432
Epoch 1: saving model to model_c_cp-0001.ckpt
64/64 [==============================] - 114s 2s/step - loss: 0.6594 - accuracy: 0.6432 - val_loss: 0.6325 - val_accuracy: 0.6740
Epoch 2/5
64/64 [==============================] - ETA: 0s - loss: 0.6396 - accuracy: 0.6601
Epoch 2: saving model to model_c_cp-0002.ckpt
64/64 [==============================] - 112s 2s/step - loss: 0.6396 - accuracy: 0.6601 - val_loss: 0.6241 - val_accuracy: 0.6740
Epoch 3/5
64/64 [==============================] - ETA: 0s - loss: 0.5716 - accuracy: 0.7148
Epoch 3: saving model to model_c_cp-0003.ckpt
64/64 [==============================] - 110s 2s/step - loss: 0.5716 - accuracy: 0.7148 - val_loss: 0.5936 - val_accuracy: 0.6929
Epoch 4/5
64/64 [==============================] - ETA: 0s - loss: 0.4015 - accuracy: 0.8533
Epoch 4: saving model to model_c_cp-0004.ckpt
64/64 [==============================] - 116s 2s/step - loss: 0.4015 - accuracy: 0.8533 - val_loss: 0.6200 - val_accuracy: 0.7096
Epoch 5/5
51/64 [======================>.......] - ETA: 21s - loss: 0.2552 - accuracy: 0.9282
```



```
171/171 [==============================] - 27s 160ms/step
F1-score: 0.564
Confusion Matrix :  [[ 982  698]
 [ 823 2967]]
```



The predictive accuracy of our model is 0.93 (with 5 epochs and 64 embed-size), and the accuracy is also fluctuating because of the randomness of the splitting of validation and train set and the stochastic gradient descent process during training the LSTM model. The LSTM model converged fast and had a relatively good performance on predicting the recommended or not recommended label on our synthetic dataset.

In addition, the model begins its epoch with weaker performance, indicating the difficulty of maneuvering the senseless order of the synthetic reviews, which are a bag of words reflective of the generative model that they came from.

Pros and Cons (Step 6)

Generative Model Pros and Cons

1. The bag of words approach completely disregards the ordering of words and assumes all words occurring are fully independent from each other. This is perfect for a probabilistic approach that relies on storing words or short sequences of words in a dictionary. If the classification does not rely on order, it is a good approach.
2. The bag of words' strength is its weakness as well. The ordering of words and how far words are from each other can reveal much context about the sentiment of a text. In addition, assuming that every word occurrence is independent is flawed. Many words have a level of codependency with each other and often appear together. "We are" is one such combination, as well as "I am".

3. The penalties were shown to be quite ineffective and harmful to the accuracy, and the worries about unbalanced datasets and the inclusion or elimination of the word "cyberpunk" were unfounded. They did not add meaningful noise to the dataset and were often inaccurate when compared to their counterparts that were balanced and inclusive of "cyberpunk".

4. Words were shared across the corpora, so the model could misclassify some reviews if the words were likely shared in a context of negation, such as "not bad" or "not amazing".

Neural Network Pros and Cons

Some pros and cons have been discussed in the previous parts, here we just want to emphasize some important takeaways.

1. We first applied “sgd” optimizer in our bidirectional LSTM and it worked poorly, and thus we switched to “adam” optimizer, which featured self-adaptive and optimization of gradient descent. The accuracy of training set is around 0.85-0.95. The reason that “sgd” optimizer cannot converge the model might be we added multiple layers in the structure.
2. We now use 5 epochs and 64 embed-size, taking 7 mins (on a 10 core-gpu laptop) to train the model on a 16000 reviews*929 entries dataset. Using more epochs and embed-size can increase the accuracy and minimize the loss after the iterations, but the training time will also increase vastly (to 20 mins if using 10 epochs and 128 embed-size). The training task can be done on any current main-stream pcs or laptops within 10 mins, or using cloud computing resource (1-core GPU is enough).
3. The accuracy of the training set is around 0.85-0.95, and the accuracy of the validation set is around 0.70-0.80. The F1-score for the model (computed with the test set) is 0.72. This is one merit of our model: high accuracy. However, there are also some aspects that we can further improve. Once we apply our model to the synthetic dataset we regenerated, the F1 score will drop more than we expected. This may be caused by the inaccuracy of the labels we regenerated. This situation suggests that the accuracy of our model depends on the accuracy of gamers’ expression of their sentiment. Also, we didn’t compare the results of our model on reviews with stopwords and reviews without stopwords. There might be difference.
4. There are some limitations of our model. For example, long-term memory neural network may also have gradient issue. The performance also depends on how long the strings can be. Though we know our model is not perfect, we are still glad to see our accuracy exceeds 0.90 on the train set on both synthetic data and original data, and we get a 0.72 F1-score on the original and 0.564 on the synthetic, which measuring the accuracy on test set.

Packages Used for Project

Numpy	collections
Pandas	sklearn
re	string
random	os
seaborn	warnings
nltk	wordcloud
requests	matplotlib

References

Brownlee, J. (2017, January 3). *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Dolphin, R. (2020, October 21). *LSTM Networks | A Detailed Explanation*. Retrieved from Medium: https://towardsdatascience.com/lstm-networks-a-detailed-explanation-8fae6aefc7f9

Ersoy, P. (2021, May 20). *Naive Bayes Classifiers for Text Classification*. Retrieved from Medium: https://towardsdatascience.com/naive-bayes-classifiers-for-text-classification-be0d133d35ba

Hamdaoui, Y. (2019, December 9). *TF(Term Frequency)-IDF(Inverse Document Frequency) from scratch in python* . Retrieved from Medium: https://towardsdatascience.com/tf-term-frequency-idf-inverse-document-frequency-from-scratch-in-python-6c2b61b78558

Keras. (2022). *Dense Layer*. Retrieved from Keras Documentation: https://keras.io/api/layers/core_layers/dense/

Khan, S. (2021, May 21). *Sentiment Analysis (Steam Reviews)*. Retrieved from Kaggle: https://www.kaggle.com/code/shahidkhan1/sentiment-analysis-steam-reviews

Muller, A. (2021, April 17). *Scraping Steam User Reviews*. Retrieved from Medium: https://andrew-muller.medium.com/scraping-steam-user-reviews-9a43f9e38c92

PSS, A. (2021, May 4). *Sentiment Analysis using LSTM*. Retrieved from Medium: https://medium.com/mlearning-ai/sentiment-analysis-using-lstm-21767a130857

Soderman, E. R., Wang, Z., & Miftah, W. (2022, December 19). *NLP-CP2077-Sentiment-Analysis Commit History*. Retrieved from Github: https://github.com/nogibjj/NLP-CP2077-Sentiment-Analysis/commits/main?after=ac4c67772fcb943f50e5fc15d93be0a71c1ceb34+34&branch=main&qualified_name=refs%2Fheads%2Fmain

Eubank, N. C. (2022, September 13). *Numbers in Computers*. Retrieved from Practical Data Science: https://www.practicaldatascience.org/html/ints_and_floats.html