Amazon Review Sentiment Analysis

By: Yoo Bin Shin, Genesis Qu, Nirvan Silswal

1. Problem Selection

For our final project, we chose to approach a sentiment analysis problem using a generative and discriminative model. We worked with the Amazon Reviews dataset, which contains over 23,000 different Amazon reviews as well as their recommendation status (whether the customer recommended the product or not). Our goal was to use natural language processing algorithms, specifically a Multinomial Naive Bayes generative model and a Long short-term memory (LSTM) and Recurrent Neural Network (RNN) discriminative model, to solve the binary classification problem on the recommendation status of a review and compare the models' performances.

2. Dataset

The Amazon Reviews dataset was obtained from Kaggle [1]. It consists of 23,486 anonymized reviews on women's clothes with 10 feature variables–Clothing ID, Age, Title, Review Text, Rating, Recommendation Status, Positive Feedback Count, Division Name, Department Name, and Class Name. As we defined the problem as a binary classification problem, the "Review Text" feature was used as the inputs and "Recommendation Status" as the target outputs.

3. Methods

a. Data Preprocessing

Prior to training the model with the real Amazon reviews data and generating synthetic data, we performed preprocessing on the raw text to make sure that the model will take in meaningful data.

Using the nltk library and regular expression, we performed the following steps of data cleaning on the review text data. We first converted the frequently appearing contractions to a standard format. For example, "could've" was mapped to "could have." All words were converted to lower case so that the same words in different capitalizations would be counted as the same word. The text was then stripped of special characters, numbers, and punctuations, leaving only words and letters. Elongated words such as "sooooo" or "goooood" were shrunk to "so" and "good." After removing common stopwords such as "with," "the," and "then," taken from the nltk stopwords dictionary, we performed tokenization and stemming.

Using the graphing functionalities in the wordcloud library, we generated the following two word clouds to visualize some of the most frequent words in both the recommended reviews and the not-recommended reviews.

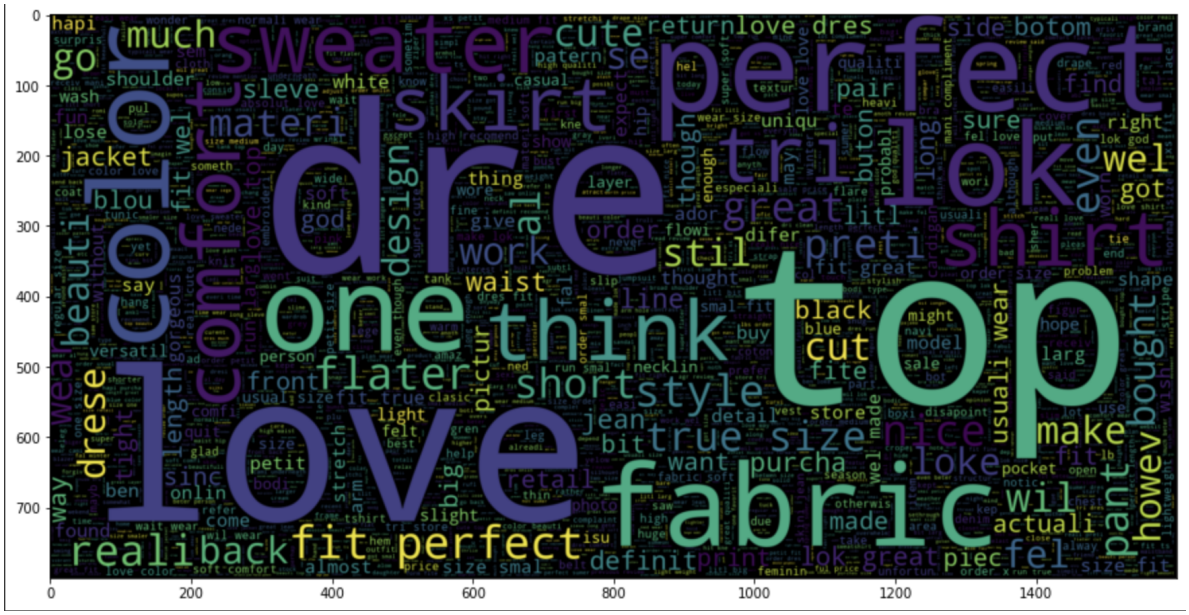

Figure 1: Recommended Review Word Cloud

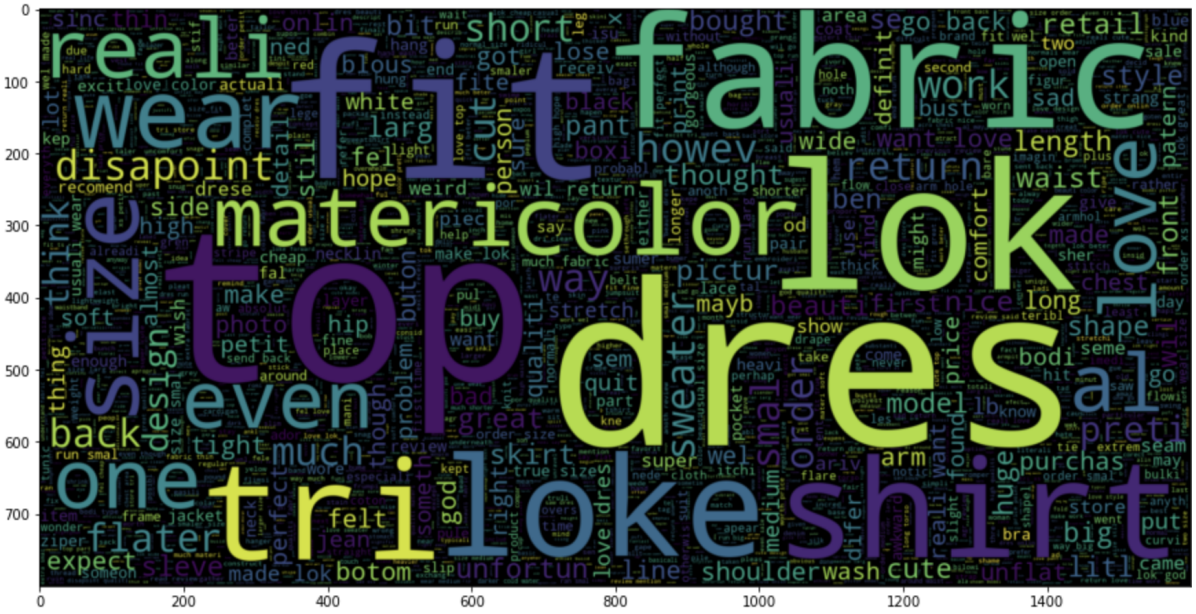

Figure 2: Not-Recommended Review Word Cloud

We noted that of the over 23,000 reviews in our data, around 82% were positive reviews and 18% were negative reviews. We kept in mind the potential effect of training the models on

imbalanced data. It also meant that the no-information rate of the model sat at around 82%, and the model needed to perform better than that threshold to achieve meaningful significance.

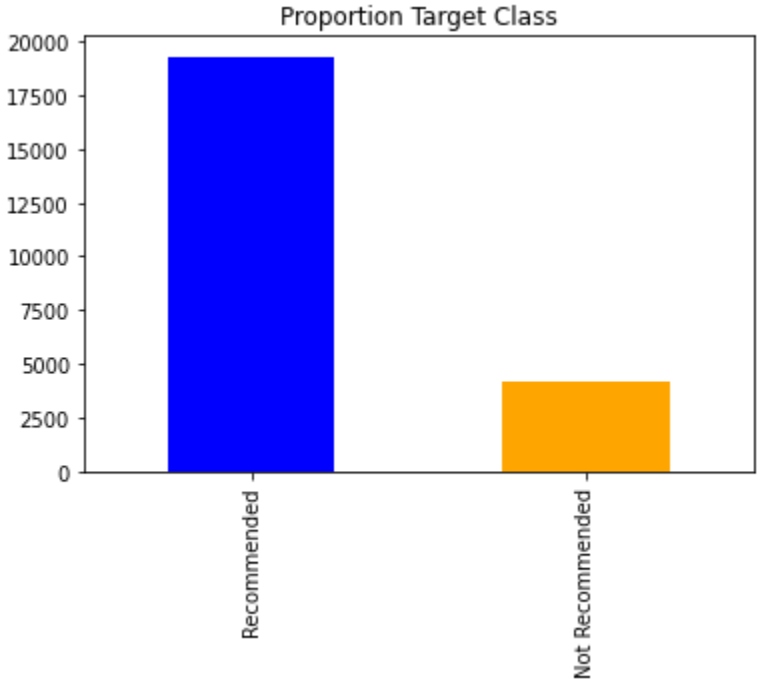

Figure 3: Proportion of target classes in real dataset

b. Generative Solution

We used the Multinomial Naive Bayes Classification model for the generative model [2]. Multinomial Naive Bayes classifiers are often used for classification with discrete features, such as word counts for text classification. This classifier works under the bag-of-words assumption, where a review is encoded with the frequency of words that appear and no positional information of words. The classifier is a probabilistic model based on the naive Bayes assumption. It assumes the probability of a word appearing in a review of a given class (Recommended / Not Recommended) is independent, allowing classification of the recommendation status by multiplying the individual probabilities of words that appear in the review. In this project, the Multinomial Naive Bayes class from the sklearn library was used.

c. Training on Real Data

To train the Multinomial Naive Bayes model on real data, the preprocessed review data had to be encoded to a vector form representing the text with the frequency of words used. To do so, the CountVectorizer class from the sklearn library was used. The CountVectorizer allows the conversion of a collection of text to a matrix of token counts. Thus, the CountVectorizer's fit_transform method was used to convert each preprocessed text review into equal length vectors, where the length of the vector (number of features) was determined by the vocabulary

count (total count of unique words used in all reviews). Each feature took the value of the occurrences of that word in a given review. For instance, the review “very very pretty” can be encoded as [1 2 0], given that the vocabulary dictionary defines the feature in index 0 as “pretty”, index 1 as “very”, and index 2 as “bad”.

The real dataset was split into training and testing data using sklearn’s train_test_split method. A train_size of 80% was used, splitting the dataset into a train set of 18,788 reviews and test set of 4698 reviews. The prior mentioned fit_transform method was used on the train data to define the vocabulary dictionary then subsequently encode the train data. The transform method was used on the test data to use the defined dictionary to encode the test data. There were a total 10,578 unique words in the vocabulary dictionary, resulting in the train data to take the shape of (n_reviews, n_features) = (18788, 10578) and test data = (4698, 10578). The target labels were “Recommended” and “Not Recommended”, thus each class was encoded as 1 and 0 respectively.

The MultinomialNB() class’s fit method was used to train the probabilistic model on the train data. Then, the predict method was used to generate inferences on the test data. To analyze the results, we used sklearn’s metrics class to calculate three different evaluation metrics. First, accuracy_score was used to calculate the percentage of correct labels out of all predicted labels. Second, precision_score was used to calculate the proportion of predicted recommended labels that were correct. Third, recall_score was used to calculate the proportion of recommended labels that were predicted correctly.

To further analyze the performance, the metrics class from the scikit-plot module was used to draw ROC curves. The ROC curve was used to visualize the Recall-Precision pair depending on the decision threshold.

d. Generating Synthetic Data

Once the Multinomial Naive Bayes classifier was trained on real data, the probabilities learned from the training data was used to generate synthetic data.
The MultinomialNB() class defines the “empirical log probability of features given a class” in an attribute called feature_log_prob_. This takes the shape of (n_classes, n_features). Hence, in this case, there were two rows–the first (nb.feature_log_prob_[0]) corresponding to the “Recommended” class and the second row (nb.feature_log_prob_[1]) corresponding to the “Not Recommended” class. As each review was encoded using 10,578 features, with each feature accounting for the frequency of a unique word appearing in the review, a row contained 10,578 log probabilities of a particular word appearing in the given class. The total probability of each row summed to 1.

These log probabilities were used to characterize a multinomial distribution that could be used to draw random samples to generate synthetic data. The multinomial random variable class from the scipy library was used and its rvs method was used to generate random samples from a defined multinomial distribution. The rvs method was supplied with the three following parameters.

n : the number of words we want each review (sample) to be
p : the corresponding row from feature_log_prob_ for either the "Recommended" class or the "Not Recommended" class (ex. feature_log_prob_[0] or [1])
size : the total count of sample reviews we want to generate

To ensure the generated dataset is similarly characterized with the real dataset, we varied the length of each generated review. This was done by drawing the relevant parameter n from a separate Gaussian distribution. Specifically, the numpy library's random.normal class was used, by providing the mean (27) and standard deviation (14) of the length of reviews were determined from the real dataset. Additionally, to keep the size of the generated dataset consistent with the real dataset, 18800 reviews were generated for the train data and 4700 reviews for the test data. However, one difference was that the class proportion was modified so that the generated dataset would be balanced. As a result, the generated dataset consisted of 50% samples from the Recommended class and 50% samples from the Not Recommended class. The aforementioned rvs method outputted the matrix of shape (n_features, size) where each row corresponded to a generated encoded review.

For the synthetic data to be trained on the discriminative model, the vector form had to be decoded to text. The inverse_transform method of the CrossVectorizer initialized earlier on the training data was used. This decoded the review from a vector indicating the frequency of each word to a legible list of words.

e. Training on Synthetic Data

The same procedures for training on real data was used to train a Multinomial Naive Bayes model on the synthetic data. The MultinomialNB() class was used to fit on the train data, then the classifier was used to infer the test data and evaluate its performance.

f. Discriminative Solution

We chose a neural network using RNN Cells and LSTM Cells as our discriminative solution. RNN classifiers are often used for modeling sequence data such as time series and natural language. SimpleRNN cells perform simple multiplication of the input and previous output which is passed to an activation function. LSTM classifiers are a form of RNNs, but allow for more flexibility and control within the outputs at the cost of more complexity and operating cost.

This is achieved through adding an update gate, forget gate, and output gate. An LSTM provides a short term memory over a much longer period of timestamps than an RNN, which could be helpful when analyzing long text. We chose to implement both a SimpleRNN network and an LSTM network to compare performance of the two models and discover which network would be better suited for review classification [3].

In this project, we used the SimpleRNN class and LSTM class from the Tensorflow library.

Each model we made was made up of a word embedding layer, a SimpleRNN / LSTM layer of 100 units, and a single dense layer with sigmoid activation. The embedding layer had an input dimension size of (the length of our word index + 1), an output dimension of 50, and an input length equal to the size of the review with the maximum length.

g. Training on Real Data

To begin training the discriminative models on real data, we first had to tokenize our data. Our end goal here is to split up the sentences into their individual words. To begin we utilized the Tokenizer class and pad_sequences module from Keras. After initializing our tokenizer, we called tokenizer.fit_on_texts on both our training and testing sentences. This fit the tokenizer on our data and allows us to generate a word index - a numerical mapping of every word to a numeric representation.

The next step was to encode our training data into sequences. Here we converted our text sentences into their corresponding numeric representations from the word index we just generated. Then we found the maximum sequence length (for use in the padding step).

The final step in tokenization was to pad the training sequences. This is because each sequence must be of the same length in order to be input to our model. Since we already found the maximum sequence length - all we need to do was pad every sequence that is less than the maximum sequence length with 0s. A call to pad_sequences was used to complete our tokenization process.

RNN:

Now that we finished tokenization, we trained our two models. The first model we fit was an RNN model. The RNN was constructed with an embedding layer, a Simple RNN layer with 100 units, and a dense layer with sigmoid activation. The loss function used within this model

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_9 (Embedding)     (None, 59, 50)            589350

 simple_rnn_6 (SimpleRNN)    (None, 100)               15100

 dense_9 (Dense)             (None, 1)                 101

=================================================================
Total params: 604,551
Trainable params: 604,551
Non-trainable params: 0
_________________________________________________________________
```

is binary cross entropy and we used the adam optimizer.
For training the model we used a batch_size of 512 for 5 epochs:

```
Epoch 1/5
37/37 [==============================] - 7s 149ms/step - loss: 0.4708 - accuracy: 0.8149
Epoch 2/5
37/37 [==============================] - 5s 147ms/step - loss: 0.3396 - accuracy: 0.8508
Epoch 3/5
37/37 [==============================] - 6s 151ms/step - loss: 0.2876 - accuracy: 0.8779
Epoch 4/5
37/37 [==============================] - 6s 150ms/step - loss: 0.2217 - accuracy: 0.9074
Epoch 5/5
37/37 [==============================] - 5s 145ms/step - loss: 0.1899 - accuracy: 0.9244
<keras.callbacks.History at 0x7f73eca678b0>
```

Our model achieved an accuracy of 0.9244 after 5 epochs and an AUC of 92%

<u>LSTM</u>:
The LSTM model was constructed with an embedding layer, an LSTM layer with 100 units (where dropout & recurrent dropout = 0.3), and a dense layer with sigmoid activation.

```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_10 (Embedding)    (None, 59, 50)            589350

 lstm_3 (LSTM)               (None, 100)               60400

 dense_10 (Dense)            (None, 1)                 101

=================================================================
Total params: 649,851
Trainable params: 649,851
Non-trainable params: 0
_________________________________________________________________
```

For training the model we again used a batch_size of 512 for 5 epochs:

```
Epoch 1/5
37/37 [==============================] - 35s 864ms/step - loss: 0.5114 - accuracy: 0.8152
Epoch 2/5
37/37 [==============================] - 30s 800ms/step - loss: 0.3445 - accuracy: 0.8426
Epoch 3/5
37/37 [==============================] - 28s 770ms/step - loss: 0.2435 - accuracy: 0.8982
Epoch 4/5
37/37 [==============================] - 28s 767ms/step - loss: 0.2146 - accuracy: 0.9097
Epoch 5/5
37/37 [==============================] - 31s 830ms/step - loss: 0.1978 - accuracy: 0.9185
<keras.callbacks.History at 0x7f73eaacf9a0>
```

Our model achieved an accuracy of 0.9185 after 5 epochs and an AUC of 92%.

h. Training on Synthetic Data

Training on our generated synthetic data followed a similar process to training a model on the real data, but we needed to ensure the synthetic data followed the same format as the real data.

To begin, the generated data came as a series of Arrays with each index in each array representing a word within a generated sentence. This is very different from the final format we want, which is a pandas dataframe where each row represents a generated review. Our first step was to combine the individual words in each array into one string, instead of a series of individual words in their own string.

```
X_train_synthetic_text
```

	Review Text
0	botom boxi brand colect color either fan fit h...
1	athletichourgla back blend color dres enough e...
2	arm back best bodi bradshaw cloth cute debat d...
3	also asum bag blous boxi compfi complet could ...
4	away awkward back beauti bete bordeaux cheapn ...
...	...
18795	around avail bagylos bit even fit gather god g...
18796	amaz arm beauti bit cami cut cute done flare f...
18797	btw daughter definit fite form got holiday ned...
18798	around backshouldersup cut flower go know ligh...
18799	flater lot pleas returnexchang review think

18800 rows × 1 columns

Initially, we tried simply passing the series of arrays to pd.DataFrame() which does create a dataframe object however, each individual sentence gets split over multiple rows within this dataframe which increases the size of X. This was remedied by declaring an empty list and iterating over every array within the generated data and joining each word at every index to the next word within the array while adding a space in between words. This string was then appended to the empty list we declared earlier. After this process is done for both X training and testing data, the two lists can be passed to pd.DataFrame() to build a pandas dataframe object that we can use to train.

This process wasn't necessary for Y as the data was filled with only 0s and 1s which could be passed to pd.DataFrame() while keeping the correct number of rows (18800 for training data and 4700 for testing data).

After these steps we had four variables: X_train_synthetic_text, X_test_synthetic_text, y_train_synthetic_results, y_test_synthetic_results . These variables can be passed to the tokenization process and model to conduct our training and evaluations of models.

From here the synthetic data followed the same process as before. Tokenization was completed following the same steps, and we fit two models: an RNN based model and an LSTM based model.

RNN:
The RNN was constructed with an embedding layer, a Simple RNN layer with 100 units, and a dense layer with sigmoid activation. The loss function used within this model is binary cross entropy and we used the adam optimizer.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 75, 50)            526200

 simple_rnn (SimpleRNN)      (None, 100)               15100

 dense (Dense)               (None, 1)                 101

=================================================================
Total params: 541,401
Trainable params: 541,401
Non-trainable params: 0
_________________________________________________________________
```

For training the model we used a batch_size of 512 for 5 epochs:

Our model achieved an accuracy of 0.9176 after 5 epochs and an AUC of 94%

```
[234] batch_size = 512
      model.fit(X_train_pad_syn, y_train_synthetic, epochs=5, batch_size=batch_size)

      Epoch 1/5
      37/37 [==============================] - 9s 194ms/step - loss: 0.6721 - accuracy: 0.5889
      Epoch 2/5
      37/37 [==============================] - 7s 192ms/step - loss: 0.4800 - accuracy: 0.7959
      Epoch 3/5
      37/37 [==============================] - 7s 185ms/step - loss: 0.3199 - accuracy: 0.8693
      Epoch 4/5
      37/37 [==============================] - 7s 184ms/step - loss: 0.2234 - accuracy: 0.9110
      Epoch 5/5
      37/37 [==============================] - 7s 191ms/step - loss: 0.2384 - accuracy: 0.9176
      <keras.callbacks.History at 0x7f8f0ae7b610>
```

LSTM:
The LSTM model was constructed with an embedding layer, an LSTM layer with 100 units (where dropout & recurrent dropout = 0.3), and a dense layer with sigmoid activation.

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_9 (Embedding)     (None, 75, 50)            526200

 lstm_1 (LSTM)               (None, 100)               60400

 dense_9 (Dense)             (None, 1)                 101

=================================================================
Total params: 586,701
Trainable params: 586,701
Non-trainable params: 0
_________________________________________________________________
```

For training the model we again used a batch_size of 512 for 5 epochs:

```
Epoch 1/5
37/37 [==============================] - 42s 1s/step - loss: 0.6475 - accuracy: 0.6442
Epoch 2/5
37/37 [==============================] - 36s 972ms/step - loss: 0.4406 - accuracy: 0.8306
Epoch 3/5
37/37 [==============================] - 36s 969ms/step - loss: 0.2926 - accuracy: 0.8803
Epoch 4/5
37/37 [==============================] - 36s 971ms/step - loss: 0.2207 - accuracy: 0.9039
Epoch 5/5
37/37 [==============================] - 36s 975ms/step - loss: 0.1726 - accuracy: 0.9248
<keras.callbacks.History at 0x7faf90772040>
```

Our model achieved an accuracy of 0.9248 after 5 epochs and an AUC of 0.96%

4. Results

Below are the accuracies and AUCs (Area Under Curve) for the generative model and two discriminative models we fit to the real and synthetic data. All three models perform at around the same rate at or above 90%. The also have AUCs of around 90%.

Table 1: Accuracy of Generative and Discriminative Model on Real and Synthetic Data

	Generative (Multinomial Bayes)	**Discriminative (RNN)**	**Discriminative (LSTM)**
Real Data	89.1%	92.4%	91.9%
Synthetic Data	90.7%	91.8%	92.5%

Table 2: AUC of Generative and Discriminative Model on Real and Synthetic Data

	Generative (Multinomial Bayes)	**Discriminative (RNN)**	**Discriminative (LSTM)**
Real Data	91%	92%	92%
Synthetic Data	97%	94%	96%

The discriminative models have slightly higher accuracies compared to the generative model. The accuracies for the real data and synthetic data are around the same. However, in terms of AUC, all three models are better at predicting synthetic data than real data.

Below are the ROC curves from the Naive Bayes classification model on both the real data and the synthetic data:

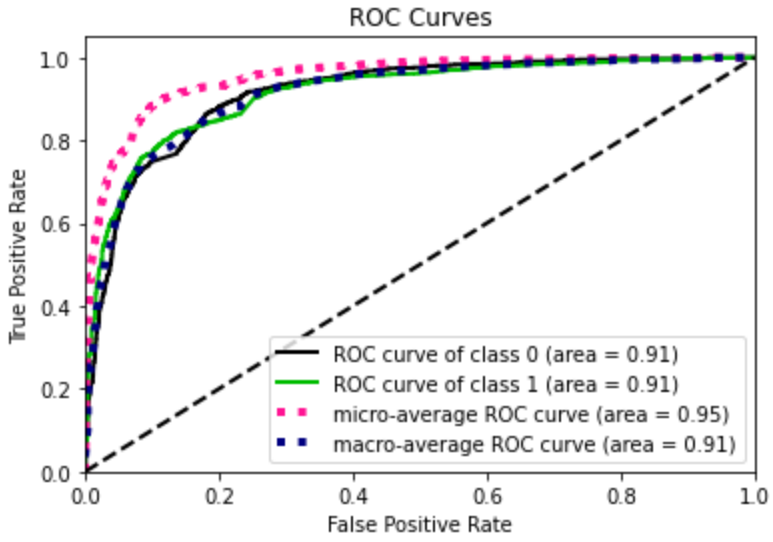

Figure 4: ROC curve on real data

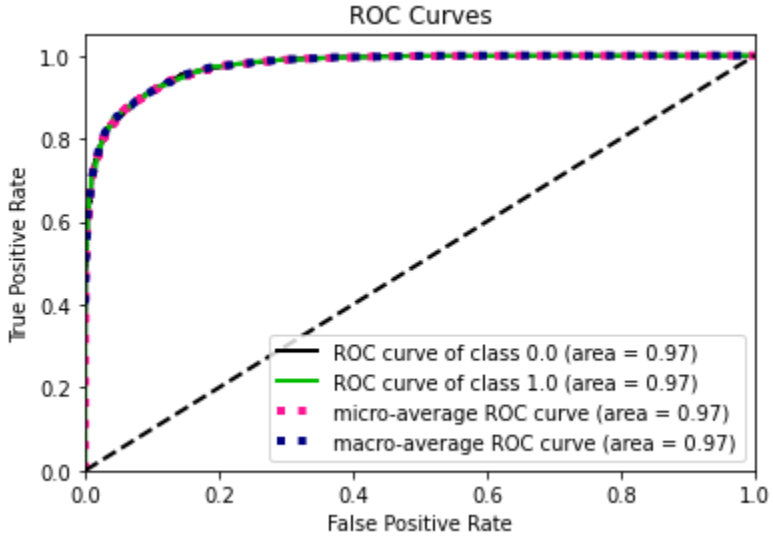

Figure 5: ROC curve on synthetic data

5. Analysis (Pros/Cons)

Overall, all three models were able to predict recommendation statuses of real and synthetic reviews with high accuracy and AUC.

The accuracy simply denotes the percentage of correctly predicted samples over total number of samples. On real data, the models' accuracies came in the ascending order of the Multinomial Bayes, LSTM, and RNN model. The two discriminative models exhibited slightly higher

accuracy than the generative model. Similarly, on synthetic data, both discriminative models outperformed the generative model. However, this time, the LSTM exhibited higher accuracy than the RNN model. The change in accuracies of the same model applied on different datasets aligns with our hypothesis. Most notably, the generative model performed with 1.6% higher accuracy on the synthetic data than the real data. This is reasonable because the generated dataset guarantees the training dataset and test dataset to exhibit more similar probabilistic characteristics. Since both the training and test sets of the synthetic data are solely based on the probabilities that characterize the real training data set, the model will not encounter new, undecodable information from the test data.

A likely reasoning for discriminative models scoring higher on real data is that the Naive Bayes generative model does not take positional information of words into account. When performing sentiment analysis or text classification, the meaning of a word is oftentimes dictated by its context and order. Therefore, it is reasonable that the positional information learned by the discriminative models was effective in improving accuracy.

While accuracy is a good measure to consider when evaluating the inference results of a model, AUC is a more comprehensive metric that is generally preferred as it calculates the tradeoff between sensitivity and specificity at the best-chosen threshold. In this case, a high sensitivity would mean the model has a high probability of identifying a positive review as positive, while a high specificity would mean the model has a high probability of identifying a negative review as negative.

In a real scenario, the models we developed would likely be used by Amazon or the seller of the product to better understand what their customers are saying about their product. Therefore, in this specific situation, we would prioritize a higher specificity over a high sensitivity. A seller would likely be more interested in determining negative reviews so that they can improve their product. Looking at positive reviews wouldn't be as informative and of lower urgency in regards to making modifications to their product.

The discriminative models exhibited highest AUCs on real data, which means their performance, when considering the tradeoff, was most optimal. Although the generative model exhibited highest AUC for synthetic data, it was expected that all models would perform with higher AUC on synthetic data due to the dataset being balanced with equal sample sizes for the Recommended/Not Recommended class.

As the AUC metric provides insight on how the model can be tweaked to better optimize for sensitivity or specificity, we determined that the model with high AUC on real data would be most desirable. This way, a business can optimize specificity to make their product better.

Generally, generative models aim to model the distribution of the two classes while discriminative models simply try to find the boundary that best separates the two classes within the dataset. With the same encoding and preprocessing, a generative model is more sensitive to outliers in the dataset, which can disproportionately impact the model's estimates of distribution parameters. Furthermore, the heavy reliance of the generative model on the training dataset may affect its accuracy when applied on test data. For instance, the vocabulary dictionary used to encode the review text is defined by the training data. Hence, when a new and potentially significant word appears in the test set, as the word has not been learned from the training set, the model is not able to use this information during inference. This is in contrast to discriminative model's resilience against outliers since it only relies on an objective function to optimize the boundary.
In terms of computational requirements, the generative model relies on far fewer parameters than the RNN and LSTM models. Assuming that the number of words in the vocabulary are of reasonable length, it requires less time to run, and constant optimization of the model is not necessary as it is based on a purely probabilistic model.

However, a generative model may be sensitive to the preprocessing methods used on the data. The specific ways that different sentences and words are stemmed, tokenized, and which characters or words are taken into account can disproportionately impact the generative model.

While the generative model is sensitive to the training dataset, the discriminative models are prone to change from the finetuning process. Due to the variable parameters and hyperparameters, discriminative models generally require more computation. As the model is inherently based on minimizing an error function, changes to the learning rate, training batch size, number of neural network layers can vary the results drastically.

Another interesting relationship we discovered was that the discriminative models performed better on data that was cleaned to a lesser extent. Our previous method of cleaning the data kept more information than the data cleaning method we ended up using and it led to different results for the models:

	Discriminative (RNN)	**Discriminative (LSTM)**
Real Data	94%	93%
AUC	88%	93%

Table 3: Accuracy & AUC of Discriminative Models on Real Data Using a Less Intensively Cleaning Method

Here we can see that the models performed slightly better after 5 epochs of training on data that was cleaned less intensively. This behavior seems reasonable as the discriminative models attempt to learn the overall sentiment of a sentence & understands context when compared to the generative model which primarily uses keywords. As the data was cleaned more - more information was lost hence the performance of the discriminative models was worse.

When it comes to interpretability, the generative model is much easier to interpret since the features given to the model are counts of unique words appearing in a review. And the parameter aims to estimate the probability of each word appearing in the review given that it originates from a positive or negative class. After training, one can also look up the log probabilities to understand how the training dataset's features are characterized. On the other hand, discriminative models are harder to interpret because its parameters are not substantially meaningful.

Overall, the discriminative models were preferred over the generative models as they demonstrated higher accuracy and higher AUC on real data. While the SimpleRNN network did have a higher accuracy than the LSTM network, we are hesitant to recommend the SimpleRNN network as we found that the LSTM network was impacted by our intensive data cleaning process. In a real world situation, a model that operates with highest automation level while maintaining high performance is desirable. As is accuracy, it is equally as valuable to compare the computation for preprocessing and manual labor a model requires when evaluating it. To gain more insight into the practical aspect, a future investigation on how data preprocessing affects the performance of LSTM would be helpful.

Finally, because our real data was not split evenly, we determined the AUC value to be ultimately a better metric to use for model selection, hence we would recommend using the LSTM network for Amazon review classification.

6. References

[1] Nick Brooks. 2018. Women's E-Commerce Clothing Reviews. (2018). https: //www.kaggle.com/nicapotato/womens-ecommerce-clothing-reviews

[2] https://www.kaggle.com/code/dyahnurlita/sentiment-analysis-using-multinomial-naive-bayes/notebook

[3] https://www.kaggle.com/code/miguelfzzz/natural-language-processing-rnn-lstm-s