ISM-6251 – Spring 2020

Darya Gahramanova

Predicting Review Helpfulness of Amazon Electronic Products

Introduction

Customer reviews are one of the most powerful tools that influence online shoppers' buying intention: around 95% of online shoppers read reviews before making a purchase. This is why online shopping platforms highly encourage customers to leave feedback, and undoubtedly, online retailers try very hard to have the best reviews possible. A lot of people contribute to future buyers' decisions by leaving helpful reviews, but sometimes people just leave unreasonably negative comments – and if they did not violate the rules, their comments would be approved and possibly hurt the company's reputation. Or, on the other hand, a product may have a lot of good reviews, but end up being of a very low quality – because most of those reviews were fake. Amazon, one of the world's online shopping platform leaders, figured out the importance of customer reviews decades ago. Not only it checks and filters the reviews before approving, but also sorts the reviews by their helpfulness using a simple trick: shoppers can leave votes on other customers' reviews if they found them helpful or unhelpful. The problem is that new reviews do not have enough votes to be displayed first, so Amazon seems to put both the most helpful and the newest reviews on top.

Still, though, sometimes fake reviews can make their way through validity check. This can be done either by random people who want to lower the rating of a product, or companies that hire people to post good reviews. Moreover, the process of review sorting heavily depends on manual human input. Although customer involvement is what makes Amazon reliable and trusted, implementing an algorithm that would automatically identify helpfulness of a review and place it respectively could potentially benefit both the seller and its potential customers. This machine learning project focuses on the first step of this idea: predicting review helpfulness using just its contents.

In a perfect case scenario, several factors would be considered: whether the purchase is verified, the length of the review, the nature of the product reviewed, the usefulness of the information in the review, whether a photos or videos were attached, and even other reviews that the user has ever written. This project, though, will focus on learning text patterns from the review body only. The next stages of the project may involve using additional features, scraping data about the users and whether media files are attached with the review. Moreover, this project only predicts two levels of the review helpfulness: either unhelpful or helpful. In further stages of this project, to be more precise with sorting, review helpfulness can be identified as a continuous number instead. Being able to automatically determine review quality can make the process of sorting much faster and easier, potentially benefiting customers, retailers, and online shopping platforms.

The data for this project has been obtained from AWS, where the company has released 130+ million customer reviews ranging from 1995 to 2015 and grouped by product type. The product group used for this project is electronics. They are one of the most popular products on Amazon, and a lot of customers significantly count on their reviews because 1) many electronic products are expensive, 2) they may arrive broken more often than other products, 3) there is a wide variety of brands to choose from.

Although these reviews are from quite a while ago, not much has changed in their language or structure, so this dataset is a good candidate for predicting helpfulness. There is one major difference in how Amazon was handling the votes though: in past, reviews could be voted as either helpful or unhelpful,

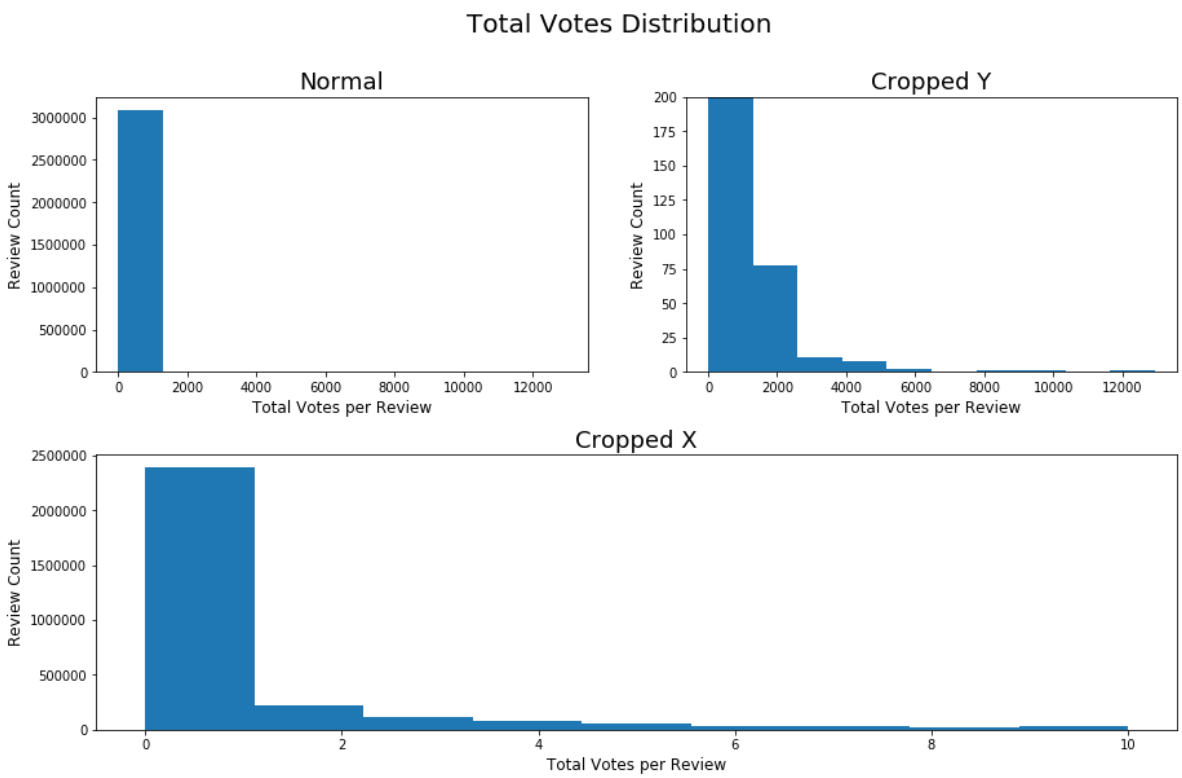
but now customers only have the option to either leave a helpful vote on a review or report it as inappropriate – no downvote options. In our dataset, the number of helpful votes and total votes are provided in the dataset – so unhelpful votes can be calculated, which will be very useful for the predictions.

As mentioned, this project aims to predict binary helpfulness scores of electronic product reviews based on the review text. There are more than 3 million reviews in the dataset, most of them do not have any votes, and those that do are usually upvotes. For this analysis, an equal number of helpful and unhelpful reviews will be sampled. A random prediction would produce an accuracy score of 50%, so the aim is to be able to achieve higher accuracy scores. Predicting something as subjective as helpfulness is a quite hard task, so my theoretical maximum for this analysis is an accuracy of about 80%-85%. Higher precision is preferred over higher recall in this case, because it is better to fail to display good reviews first (false negative) rather than putting unhelpful reviews on the top (false positive).

Getting the data

First, I read in the raw data which was downloaded in .tsv format. It appears that some rows were read into that file incorrectly and were twice the length of the normal rows. That creates problems with pandas' read_csv, so to skip those rows the error_bad_lines parameter must be set to True. The data has 3,091,024 rows and 15 columns: marketplace, customer_id, review_id, product_id, product_parent, product_title, product_category, star_rating, helpful_votes, total_votes, vine, verified_purchase, review_headline, review_body, and review_date. Helpful votes and total votes will be used to calculate the target variable as demonstrated later. The only feature that will be used in this analysis is the review body, as the project is primarily focused on text analytics. Some other variables that might be useful to predict helpfulness in further studies are product category (type of product reviewed), star rating (the rating left by the reviewer), verified purchase (whether the purchase was officially done through Amazon), review headline (the small text in the header of the review), review date (to put less weight on older reviews because they had more time to collect votes), and customer ID (to analyze whether the profile is fake).

The charts below demonstrate how total votes per review are distributed in the dataset. Although the largest number of votes left on a single review is 12,944, most of the reviews do not have any votes. The 75^{th} percentile of the number of votes per review is equal to only 1. Although it is unclear whether the reviews without votes were unhelpful or unnoticed, they will not contribute to prediction of helpfulness and will be discarded in this analysis. However, they might be used to build a more accurate vocabulary specific to Amazon reviews to use in text encoding for the future steps of the project.

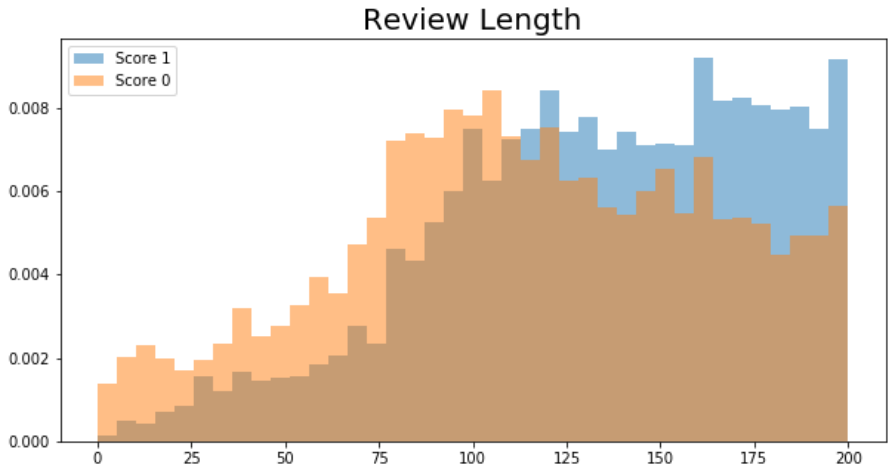


The next step is to filter data even further. I am only using English reviews in this case – so the marketplace is set to US only instead of 5 more countries. Moreover, only the reviews with 7 total votes or more are used. Next, a binary review helpfulness score is calculated using the filtered data: 1) the sum of helpful votes per product is calculated, 2) "helpfulness" ratio is calculated by dividing the number of helpful votes per review by total votes per review, 3) "popularity" ratio is calculated by dividing the number of helpful votes per review by total helpful votes per product calculated in step 1, 4) continuous review score is calculated by estimating the weighted average of "helpfulness" with weight of 0.7 and "popularity" with weight of 0.3 per product, 5) continuous score is split into 3 bins of 0-30%, 30%-70%, and 70%-100%, 6) binary score is estimated by converting the lower 30% to 0 and the upper 30% to 1, and the middle bin is dropped. The weight of the continuous review score is set to 0.7 for helpfulness and 0.3 for popularity, so that more priority is given to the review helpfulness, but the total number of votes per review compared to other reviewers of the same product still affects the score. The middle bin is dropped to have enough difference between unhelpful and helpful reviews. Predicting helpfulness is a difficult task by itself, so keeping ambiguous scores would make it even harder to make accurate predictions at this level.

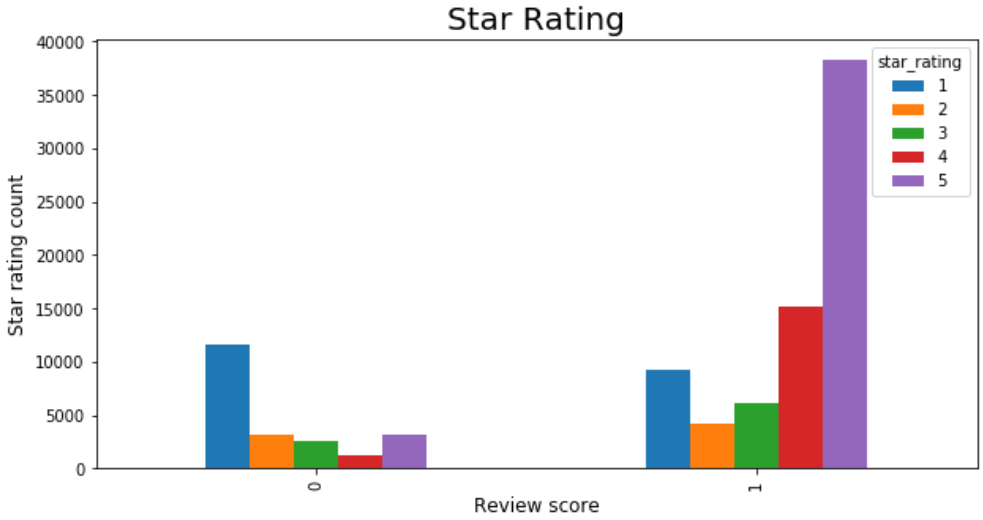
Selecting rows with the lowest continuous scores generates the list of the least helpful reviews as well as some potential outliers in the dataset. The result is a set of reviews with legitimate structure and meaning, but highly negative sentiment. Another possible way of detecting outliers is looking at the longest review, as done is the next step. Although the longest review in this dataset also has structure and meaning, it is full of HTML characters, supposedly incorrectly parsed into the file. This will be fixed during the cleaning process.

One possible difference between helpful and unhelpful reviews might be the difference in length of the review: supposedly, more helpful reviews are longer. To visualize this, two histograms (one for each binary score) representing corresponding reviews lengths are plotted together. Resulting chart shows

that indeed unhelpful reviews tend to be shorter than the helpful ones, but the intersection of the two histograms is too large to indicate that review length is a good predictor of helpfulness.



Another variable that might represent the difference in helpfulness is the star rating left by the reviewer. Because the least helpful reviews were all negative, a plot of star rating per helpfulness score might be insightful. The chart shows that 1) there are much more helpful reviews than unhelpful ones in the dataset, 2) there are much more 5-star ratings than 1-star ratings, 3) helpful reviews have different score ranges, although mostly positive, while most of the unhelpful reviews have a 1-star rating.



These variables are used for the insights only and might be helpful in future studies. In this case, the models are expected to learn from review body only, and if possible, infer review lengths and sentiments from there. Therefore, the review body is selected as the independent variable and the binary score as the dependent variable in this analysis. Moreover, because there are much more positive reviews than negative ones, 15,000 of each type are sampled for equal representation. This will ensure that the models have enough negative reviews to learn from, and that the baseline accuracy of the models is 50%.

Text Preprocessing

One of the most common steps in text preprocessing is getting rid of the “stopwords” – very common words that would probably not be useful for the predictions. In this analysis, I am keeping some of those

stopwords, as the models used here should be able to infer that very frequent words are not that important. Or maybe some of them are actually important – especially the negative words like “no”, “not”. Sometimes removing them could change the meaning of the reviews. I defined a custom stopword list that mostly includes different forms of the verb “be”, very common prepositions, empty spaces and some parts of words that are specific to my cleaning strategy.

Next, the cleaning function is defined: it gets rid of HTML characters, turns “n’t” into “not”, transforms numbers of different lengths into 0s (I assume that differentiating between different numbers would not matter much in this case), strips punctuation, only chooses words/numbers, and puts them back together with spaces between them. This function returns a single string of words with single spaces between them rather than a tokenized list of words, because some models take strings and others take tokens. Cleaned strings are easily tokenized later as needed. An example on a single review shows how the cleaning is performed, and then cleaning is applied to each row of the sample dataset.

The next function takes each row of the cleaned text and stems each word in it. The function is again applied to each row of the cleaned text and outputted to a different column. The goal is to test whether stemming improves model performance.

NLTK library offers BigramCollocationFinder and TrigramCollocationFinder classes to find words that are most used together in a text. This is useful not only for gaining insights about a dataset, but also for producing a cleaner input for a machine learning model by combining collocations into a single string. In this case, I am using bigrams, or word pairs, as after experimentation I found that in this case, they are producing a better result than trigrams. BigramCollocationFinder takes a single list of words as an input, so all reviews in the dataset are tokenized and combined into a single list as if it is one text. If applied without a filter, the finder outputs a lot of pairs consisting of numbers, pronouns, and other not very useful words. Therefore, broader stopword blacklist is created to filter those words out. The result is a much more useful word pairs frequently occurring in the dataset. The next step is simply combining each word pair with an underline in a separate column of the dataset – again, to compare its performance to simply cleaned reviews and stemmed cleaned reviews.

All three versions of cleaned text (as different versions of the independent variable) and the binary review score (as the dependent variable) are split into 80% train and 20% test sets. In the following sections different vectorization strategies will be tested on the three versions of the cleaned text and trained using different models.

TF-IDF Vectorizer

One of the most common approaches to transform text into an input for machine learning algorithms is the bag-of-words approach: words are replaced by a number indicating either their count or some other measure, the number of features is fixed, but the order of words is lost. Term frequency–inverse document frequency is one of the most powerful bag-of-words vectorization techniques because it sets a higher value for the words that appear most in a given document, but lower value to those that appear frequently among all documents. So, frequently appearing words that were not included in the stopword list should not have a high value after the TF-IDF vectorization. However, this results in a dataset with too many features, most of them very close to 0. So, it is better to perform latent semantic analysis: reducing the dimensionality of resulting vectors by combining some values together. This is achieved by Truncated SVD: 10,000 feature vectors are reduced to only 100 dimensions.

One of the benefits of TF-IDF is that the vectorization process is done fast compared to deep learning embeddings. I am thus using this opportunity to experiment with the performance of the three cleaning strategies in different models. First, four different vectorized datasets are created using the following strategies: 1) TF-IDF followed by SVD on the cleaned reviews, 2) Bigram TF-IDF followed by SVD on the cleaned reviews, 3) TF-IDF followed by SVD on the stemmed reviews, 4) TF-IDF followed by SVD on the reviews with merged collocations. Bigram TF-IDF uses all word pairs instead of just single words – an alternative to finding collocations. All models are fit on the train data only.

To evaluate the performance of each cleaning strategy, I am using a traditional yet quite powerful model – random forest. As mentioned previously, because the dataset is a sample of equal parts of unhelpful and helpful reviews, the chance of randomly predicting the label correctly is 50%, so this is the baseline accuracy for this project. It is proven by shuffling the target column and evaluating the performance of a random forest model. Indeed, the accuracy score is very close to 50%.

For the real evaluation, a default random forest is fitted on each of the four vectorized datasets. The accuracy scores are between 72.2% - 75%: not state of the art, but already a lot better than baseline model. Although a plain random forest might not be the best model for text analytics, this comparison provides important results: 1) bigram TF-IDF performed the worst out of the four, 2) the other three performed almost equally better than the plain TF-IDF. It follows that in this case merging collocations is better than using bigram TF-IDF, but there is no significant difference between using single words, collocations, or stemmed words. Maybe a larger dataset would achieve a higher difference in scores.

Finally, a gradient boosting classifier is used on the stemmed text to compare to the random forest's results. It performed almost the same with accuracy of 75.2%, proving that as other traditional machine learning algorithms can also be used with the TF-IDF + SVD vectorization and provide reasonable results without getting into deep learning.

Next, several neural network models are implemented to see whether they it is possible to achieve better results on the same TF-IDF + SVD vectorization strategy. Stemmed reviews are used in this case, which should not be too different from the cleaned text and the one with collocations. The first model has three hidden dense layers of sizes 300, 200, and 100. The model is very prone to overfitting, so dropout layers and early stopping are added. In fact, all models in this analysis take input vectors based on the training data only, so it is easy to overfit. Learning rate of the optimizer (Nesterov in this case) plays significant role in finding optimal weights, and in this case the value of 0.01 worked well. The model resulted in both better accuracy and precision scores of 78.4% - better than the traditional models. In sum, TF-IDF + SVD vectorization works reasonably well for large datasets and can be used both with traditional machine learning and deep learning models.

Doc2vec

One of the weaknesses of the bag-of-words approach is that it ignores the sequence of words. What if it matters in this case? One of the vectorization strategies that considers the sequence is Gensim's Doc2vec model. Doc2vec learns the "meanings" of words based on the words' similarity rather than just frequency. The model could be trained entirely in Gensim, but I am only training the embedding and using it as an input to other models.

Doc2vec's format is different from other models in this analysis: it takes a list of tokens as an input (so the reviews are tokenized first) and creates a dictionary of words and embeddings as it is being trained. I am not using testing data to build this dictionary, so it has the words only from the training set. The

docvecs object contains the vectors of the training set, and the infer_vector methods predicts vectors for the test data based on the words in the dictionary. Resulting vectors are then saved to a dataframe.

First, I did a quick check of the results using a random forest. Unfortunately, accuracy score dropped to 70.9% and precision to 66%. Next, I am testing Doc2vec vectorization on deep learning models. One of them is a DNN with two hidden dense layers of sizes 200 and 100 with early stopping, dropout, and SDG optimizer (learning rate = 0.01). It resulted in an accuracy score of 74.9%: better than the random forest, but still worse than TF-IDF.

Another model used is recurrent neural network. RNN is usually is a good model for text analytics where the sequence of words matters. It would not make much sense to use it with a bag-of-words model, but since Doc2vec preserves word sequence, it is worth trying an RNN on it. RNNs usually take too long to train on large datasets if used with an embedding layer. In this case, the text is already vectorized, so it is taking less time to run. The model has two LSTM layers of size 50, dropout of 0.2, and Adam optimizer with the learning rate of 0.01. Because RNNs take 3-dimensional vectors as input, the 2-dimensional data is reshaped first. Although early stopping was initiated, this model was overfitting less than the previous models, and thus ran for more epochs. The result is almost the same as the dense model with the accuracy of 74.6%.

The last model using Doc2vec embeddings is a bidirectional RNN. Standard RNNs are trying to predict the next word based on the current one – but Bi-RNNs consider both preceding and succeeding words. The model uses one 100-dimensional bidirectional layer. It achieves accuracy of 75.1%: slightly better than in previous tasks. The conclusion is that the sequence of words might not matter much in this case, and TF-IDF would achieve better results faster.

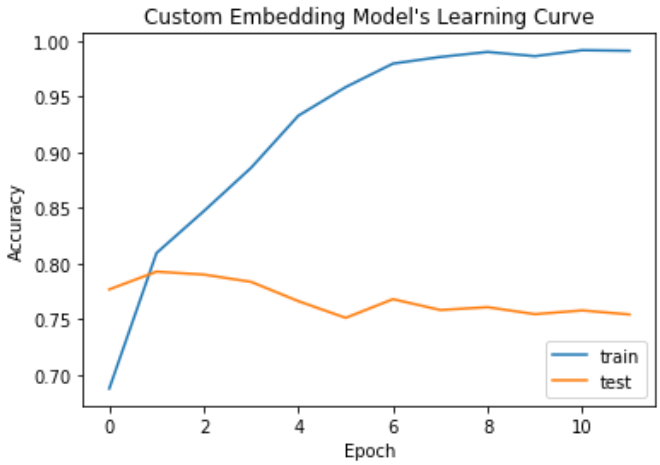
Training with an Embedding Layer

In this section, I am creating a trainable embedding layer within the Keras model rather than using an existing strategy. This is another approach where the sequence of the words matters in contrast to the bag-of-words. The input to the model would be the words in the same order as they appear in the reviews – just replaced with numbers. The numbers in this case are generated using Keras Tokenizer class: it creates a dictionary of words and their respective numbers assigned. This number is equal to the ranking of the words based on their frequency in the dataset. I have specified a dictionary size of 5000 words, and the rest of the words will be marked with an “OOV” (out-of-vocabulary) tag. As before, tokenizer is fit on train data only. texts_to_sequences method vectorizes text based on this dictionary, and sequences_to_texts can transform the list of numbers back to the text form, as shown in the example.

One fact about Keras Tokenizer worth mentioning is that the dictionary will contain the numeric values of all words in the training data – not only the selected 5000. The dictionary can be accessed using .word_index.items() method as demonstrated. However, this is done to ensure that the same tokenizer can be applied to different data. In the model itself only the first 5000 of these words will be considered.

Resulting vectors are of different lengths – just as the reviews. Before being fed them into the model, they have to be padded: zero values are added to the end of each of them to match the length of the longest review. Now the input is ready for a Keras model: in this case, it is a DNN with an embedding layer with 30 dimensions and two hidden layers of size 200 and 100. This model is very prone to overfitting, so I made it shallow, with early stopping and higher dropout (0.5 and 0.4). The learning

curve plotted below illustrates the difference between the train and test scores to demonstrate the need for early stopping.



Because of the need to train the embedding layer instead of using a ready one, this model takes very long to train, so I did not run it for too many epochs. The learning rate could be reduced and trained for longer if needed, but the results are already quite promising: 79.9% accuracy score. This is so far the best performing model in this analysis, though it takes a while to train. A solution to this is in fact a very common strategy in natural language processing: using pretrained layers.

The .layers[0].get_weights()[0] method returns the weights of the embedding layer that can be saved and used with other models. However, this is out of the scope of this project, but can be implemented in future steps.

Using Pretrained Embedding Layers

One of the best ways to work with language is using embedding layers that have been previously trained on large data. There are different ways of using pretrained layers, including allowing them to be trainable, freezing some layers and unfreezing others. But in this case, I am using them with fixed weights. The embeddings in the following models are from Tensorflow Hub and were trained based on NNLM with two hidden layers on English Google News 7B corpus – obviously much more training data than just the reviews.

The first model uses the 50-dimensional version of the embedding layers with one additional hidden dense layer of size 30. The result is close to 74.8%: worse than the custom embedding, but trained in a fraction of time. The second model is with 128-dimensional embedding and a 50-dimensional hidden layer. It boosted the performance to 77.2% but took a while to run. Still, trying out different pretrained layers may achieve results close to the ones with a custom embedding layer faster and should be explored further.

Making Predictions

Now any of the models can be used to make predictions on a couple of reviews for demonstration purposes. I chose the the Keras model with the embedding layer because it has one of the highest accuracies. There are two examples given: both reviews have negative sentiment, but one review is predicted as 0 (unhelpful), and the other is predicted as 1 (helpful). I would agree with the model: the first review is rather angry, while the second one provides details on what exactly is wrong.

Conclusion and Suggestions

There are several things that can be done to supplement this analysis. For instance, I have used stemmed text and collocations separately in the bag-of-words approach, but the two preprocessing techniques can be combined and tested against these techniques used separately.

Moreover, NLTK offers a number of POS tagging techniques not utilized in this project: the words can be tagged according to grammar rules, their role in the sentence, their dependent words, and so on. I have not touched this topic because it seems more useful for the tasks that need to understand or generate sentences and may be redundant for this task – but it is surely worth trying it out, maybe in combination with other vectorization techniques.

Another idea is that RNNs were only implemented with Doc2vec embeddings, but they could perform better with custom embeddings. The reason that I have not implemented this is that they take too long to train – but a pretrained layer might be helpful. As mentioned before, even my own custom embedding layer's weights can be saved and reused with other models. Doc2vecs themselves could also be trained in the native Gensim library rather than in Keras.

There are many pretrained layers to try. It might be true that the reviews have their specific "language" and the embeddings should be trained specifically on the review texts to achieve the best results, but I believe that with sufficient experimenting with different pretrained layers and fine tuning, the models with pretrained layers could reach the accuracy of custom models and save a lot of time. For instance, Bidirectional Encoder Representations from Transformers (BERT) is one of the models worth trying.

One of the major results of this analysis, though, is that data like this does not necessarily need to train embeddings to perform reasonably good: a common bag-of-words approach with TF-IDF works just fine. What matters to understand the idea of the reviews is mostly their contents rather than specific grammatical structure or word order, which are more useful for speech recognition or text summarization tasks.

All models in this analysis reached the accuracy of at least 70% - which is already much better than random and can contribute to a better sorting strategy of the Amazon reviews. However, the task was somewhat simplified by dropping the reviews that were "closer to the middle" – neither highly helpful nor highly unhelpful. The future analysis may take these reviews into account as well. Moreover, instead of predicting a binary score, the models can be trained to predict more levels of helpfulness – like very unhelpful, somewhat helpful, very helpful, and so on.

Finally, if this level is achieved successfully, it may be useful to look further into other features of the dataset, and even obtain other information like whether a review has media files attached, or other reviews left by the same user. Not only this would be useful to predict helpfulness, but also to detect fake reviews.