Amazon Review Helpfulness Classifier Report

Toluwa Fayemi

Objective

Using "Amazon Review Data (2018)" dataset, predict if a product review (free text) will be useful for other potential buyers.

Approach

It's easy to see why a company (like Amazon!) would find value in a predictive model that can somewhat reliably determine wether a crowdsourced review is helpful or not. As an exercise in machine learning and deep learning it provides us an opportunity to explore the elementary processes of conceptualizing, building, testing and fine tuning a machine learning model.

At a high level, the goal is to build a model capable of predicting the helpfulness of an Amazon product review to a meaningful degree. My approach to this problem was first to construct a dataset from the source database that was both balanced and diverse, extracting the information that we intend to feed into the model (the reviews themselves) and "processing" them: editing the review text to make them efficiently readable by our model while retaining their core sentiment. After that, I'll develop a few Neural Network architectures that will read the data, make predictions, and adjust its parameters based on a how its guesses are to the desired outcomes. Each model will be based either on a different neural network strategy or a different interpretation of the desired outcome.

Features & Labels

Naturally, our first inclination when approaching a machine learning problem is to identify what our **features** (information we will be feeding into the neural network) should be and what our **labels** (information we wish to get out of the model) are. This presents as a fairly simple question, but as machine learning engineers, it's important to think deeply about simple things. In this case, our principal features are easily identified as the product reviews themselves - it behooves us to consider what other factors might influence the perceived helpfulness of a review. For instance, we can argue that reviews posted earliest would be seen by more users, yielding more votes; reviews with higher votes tend to appear first, and are thus more likely to be seen and, in turn more likely to be voted on again; our text normalization process removes punctuation and reduces words to their 'stems', but a careful analysis of the reviews "readability" (which includes grammar and punctuation) could influence how users receive the review; the number of images attached to a review might influence how helpful it is, and further, a sufficiently advanced CNN could (in theory) take the images themselves as inputs of pixel vectors and use that to determine how influential they are to the overall perceived usefulness. There are a myriad of ways in which the features available to us may present themselves as causal determinants of a review's ultimate perceived helpfulness, and on a longer timeline it would be good practice to perform randomized controlled trials on the available variables and their possible representations. For now, however, as a baseline experiment (and because our ultimate deliverable will be an API that will take *only* the review text as an input) we'll stick with analyzing just the review text as normalized tokens.

The labels - our desired outcomes - are a little more malleable. Each review contains an attribute called "vote" which reflects the number of times another user has "voted" that this particular review is helpful. A quick inspection of the Amazon Review Data shows that while most of the reviews have zero votes (recorded as NaN), the number of votes for a given review could fall anywhere between the range of 1 and several thousand. This could be the result of a number of factors (namely, presumably, level product exposure and duration of posting), many of which have nothing to do with the substance of the review itself which makes predicting the exact number of votes (especially given only the text of the review itself) an unreasonably difficult task for our poor model. A much fairer ask would be to ask the model to predict the **ratio** of votes to the *total* votes for reviews of a give product. This will allow us to scale for products with higher exposure and 'normalize' the votes to a floating point integer between 0 and 1. The total number of votes for a given product are not provided to us, but with some creative programming, we can obtain them in a short amount of time.

Once we've added a section for each review's **vote ratio (vr=**[votes/total votes]), we need to determine what these ratio's actually mean - that is to say, we need to make ask ourselves "What constitutes a *helpful* review?" There are many ways we could answer this question, and each answer will inform how we frame our problem and, further, how we design our model.

Multiclass Classification

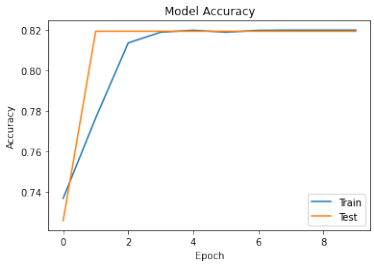
One way we could answer this is to divide the ratios into many categories. In one of my models, I divided the ratios into four groups: **Not Helpful (Vote Ratio = 0.0), Somewhat Helpful (0.0 < Vote Ratio < 0.3), Helpful (0.3 < Vote Ratio < 0.7),** and **Very Helpful (Vote Ratio > 0.7).** By interpreting "helpfulness" as a range of values, we've framed our problem as a **Multiclass Classification Problem".** In a multi-class classification problem, we train our model to make a decision between a number of options; more specifically, our model will spit out a probability distribution in the form of an array [x1, x2, x3, x4] where each element represents the likelihood (from 0 to 1) that that the "answer" is the corresponding index. For this model, I'll use Sparse Categorical Cross Entropy to compute our *loss,* and the standard out-of-the box *Adam* optimizer.

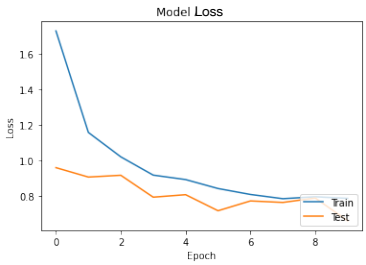
Binary Classification

As the name suggests, we can also treat the problem as a distinction between two options. In this model, I divided the ratios semi-arbitrarily into two categories: **Helpful (Vote Ratio > 0.2)** and **Not Helpful (Vote Ratio < 0.2).** While treating the concept of "helpfulness" as a binary distinction misses some potentially important nuance, it allows for our model to be trained with less data and higher accuracy. For this model, I used standard Binary Cross Entropy to compute the *loss* and, again, the *Adam* optimizer.
As our results will show, how we choose to interpret our data directly (and dramatically) shapes how well the model is able to use it to make its predictions.

Diversity of Data

As a test, I trained my first model (described below) using data from the collection of "Luxury Beauty" product reviews. I was able to obtain an apparent accuracy of approximately 82%, which isn't too bad for a fairly naive classifier. This result would be satisfactory for a learning exercise, but as a machine learning engineer it's important, again, to think deeply about why our models fail and even more-so about why they seem to succeed. For a number of reasons, the figures from this first model aren't terribly convincing. Firstly, if our objective was to determine wether reviews of Luxury Beauty products were helpful or not, our dataset may be adequate. However, because our task is to determine wether reviews *in general* are helpful,





it's important for us to diversity our data. In my following tests, I selected reviews from multiple categories of physical products (excluding datasets for reviews of non-physical products such as Digital Music and software in order to maintain some consistency of the language). I selected equal subsets of each dataset, combining them into a larger dataset.

Secondly, after normalizing and labelling our data, we find that the non-helpful reviews *far* outweigh the helpful revies. In the intermediary for our second model, it was shown that of about 40000 records, only about 6000 were not labeled **Not Helpful** and of that only about 500 were labeled as **Very Helpful.** With that, our model could theoretically guess **Not Helpful** every single time and still get a score of above 80%. We know that our model isn't doing this, but knowing that it *could* makes our 82% suddenly far less interesting. It does, however, give us something to work off of.

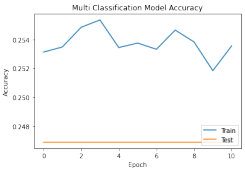
The Model

The architecture of the model itself consists of a pre-trained **embedding layer** constructed from the Stanford University gloVe project, a **LSTM Layer,** two **Dropout** layers, a single dense layer and an Activation layer.
A brief explanation of each of the layers:

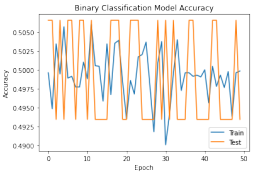
- We can think of the **Embedding Layer** as a filter: our input words (in this cased, indexed) pass through the embedding layer and come out on the other side as a vector representation of the input words. In order to get these vector representations, we use the pre-trained gloVe vector embeddings, which have shown remarkable mathematical relationships between vocabulary words and is scalable to comparable vector sizes (100-dimension vectors in this case).
- **LSTM** stands for Long Short Term Memory and is a sub-architecture of recurrent neural networks that allows the model to "remember" the contexts of word inputs, placing an importance on not only the words used but the relationships between the words as they appear.
- The **Dropout Layer** is important for preventing overfitting by randomly *dropping* (hence the name) neural network nodes.
- For my **Activation Layer,** I elected to use the ReLu function as opposed to the standard Sigmoid activation function for its speed and effectiveness of its gradient.

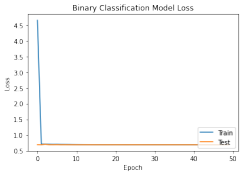
Results

Training our previous **multiclass classifier** model on a now balanced and diverse dataset resulted in an accuracy of approximately 25% and a validation loss of roughly 1.38.



Reducing the "helpful/non-helpful" (our **binary classifier)** distinction to a 2-label duality yields seemingly more promising results, with an accuracy of approximately 50%. While higher, this result isn't satisfactory for the same reason that the results from our initial tests weren't satisfactory. Again - we've got to think deeply about simple things.





When the composition of our model was skewed heavily in favour of **Not Helpful** results, we were able to obtain an accuracy of up to 80%. When the dataset is balanced, our accuracy hovers around 50%. When our accuracy reflects the composition of our data, it tells us that the architecture of our model is having difficulty converting its insights into meaningful predictions.

Final Result & Conclusion

With insights from these models, let's try a different approach. I assumed the problem would require our model to recognize relationships between words and their contexts. However, recent academic literature on similar problems suggests that **keywords** may be a far more promising distinction. I decided to change *how* our model learns - specifically, simplifying it. I tried a new pre-trained embedding layer trained on Google News articles (as opposed to Wikipedia articles a lá gloVe), and added a simple 16-node hidden layer. To test my new theory, I removed the LSTM layer; rather than training the model to remember short term relationships between n-grams (word sequences), I trained it to treat the reviews as what we call Bag of Words inputs. Like the name suggests, the model disregards the order of the reviews and treats them as if we handed it a jumbled up "bag" of the words used in the review. With these changes, I was surprised to see a training accuracy of 95% (likely inflated due to bias and overfitting) and a test accuracy of approximately **80%.** Further, my model obtained an F1 score of 0.80, indication that the model's precision closely reflects its accuracy. This result confirms our intermediary hypothesis, that the keywords (unigrams) presented in a review are more important than the relationships between words in determining the helpfulness of a review. Though this is counterintuitive to my initial assumptions, it's a valuable insight that I hope to use as I continue to test more complex models based on this final one. Finally, it proves, once again, why it pays to think deeply about simple things!

```
results = model.evaluate(test_dataset.batch(512), verbose=2)
8/8 - 0s - loss: 0.5470 - accuracy: 0.7947 - f1_metric: 0.8007
```