Document Reading Chat Bot Technical Report

The problem that we are addressing is the demand for precise and easy to understand information at scale. If the power of search engines have made the encyclopedia extinct, then why must we be grieved with the long tedious process of deciphering product documentation, legal documents, and nutrition labels? The answer is in the data, an enormous amount of data.

Goals

A fully fledged chat bot with perfectly scripted responses is a hefty goal when applied to general information. In this project we are merely proving the understanding of a machine learning model reading in natural language. The scope of success requires a model to demonstrate understanding of both the user article and the read article to provide a reasonable response. For simplicity I will only expect boolean responses with the understanding that measurement is difficult because the correct answer is subjective the reader's interpretation.

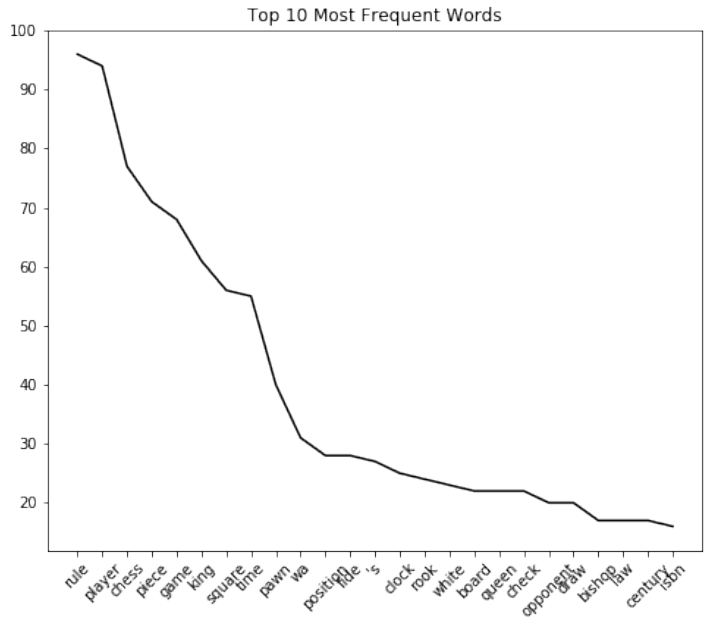
Method

The idea was that we could take an article and have a machine learning algorithm read it in as a matrix, then the user could ask a question and the algorithm would produce a reasonable answer from the text. There are several hidden pieces to this process but the biggest challenge is how to address training. The algorithm needs to be trained on the appropriate language prior to reading the document. For this task I chose to use Gensim's Word2Vec and combined it with a looping ngrams algorithm. This gives the model three inputs, a user input article, a read article, and training data. Training is prior to production since it is a computationally expensive task. The other two inputs are processed when the model is loaded since both of those articles are significantly smaller. Then a similar process appears in the Word2Vec's vector algorithm.

Once the data is loaded in, the ngrams loop begins to create all possible word grams up to the length of the user article. These grams are then counted for each time they appear in the read document. Ranking appears in the form of exponential growth. Grams that are longer will be brought the nth power when ranking is applied. The power that these grams are brought is defined in model parameters as the weight mod. Then the word vectors perform a similar search on synonyms. The hope is that the single word gram's vectors will produce the correct synonyms for the user document. These synonyms are then counted as their vector value and multiplied by a mode parameter known as the vector mod. Both model's counts are summed and this is known as the user article's relevancy score. This score is passed to a gate that is set in the model parameters and returns true if the gate is below the relevancy score or false if the gate is above. Below is the pseudo written to form the basic syntax of the model.

1. Read in user article and read article into the trained model.
2. Create ngrams starting at 1 up to the length of the user article.
3. Search the read article for these ngrams. If one appears, give it a score.
4. The score will equal the length of the gram to the power of weight mod.
5. Input the single grams into Word2Vec's trained model and return a number of similar words.
6. If these similar words appear, give the root word a score according to the vector value of this word multiplied by the vector weight parameter.
7. Compare the sum of the user article's score to the gate parameter.
8. If lower, return false.
9. if equal to or higher, return true.

To begin, I needed article that would contain rigid statements that could be described by boolean data. With this in mind, I chose to use the Wikipedia API and read in the Rules of Chess article. The article is processed, meaning stop words have been removed and the remaining words have been lemmatized. The purpose for lemmatization is to avoid the model perceiving different tenses and or modifications to a word with the same meaning as different words. This strategy was implemented knowing it would cost accuracy but increase the ability to generalize.



In this figure, we can see the most frequent lemmatized words in the Rules of Chess article minus the stop words. This gives us an insight to how our model will look at the data. Right away if a user were to use the “rule” we know that will carry significant weight to the relevancy score assigned to the user article. This can hopefully be countered by increasing the vector weight parameter allowing for synonyms to carry more meaning than reoccurring words in the article. One limitation already getting noticed, is that repeated words that appear in the article will always provide a high relevancy score. For example, if the user article is “Rule rule rule rule rule”, the prediction will almost certainly be true. Which brings up another issue, statements are often subjective and difficult to measure. Since there is no absolute truth for the repetitive article, we have to assume that it actually is true according to the read in document. According to our model, anything that is relevant enough to the read article is considered a true statement. This may need to be addressed for production, but if we can get the model accurate enough now, this current assumption will suffice.

Since the point of this project is to generalize well, all data cleaning and processing needs to be done in a repeatable way. Hence the use of the library.py file in the main directory. A processed article becomes an object with these values already in place. This is then passed to the model along with the parameters and predictions can be made with new user articles as input.

Performance

Now that the parameters have been programmed and data has been chosen, we need to find the optimal parameters for the model to operate with. A test data set has been chosen with labeled values. The Rules of Chess article has and a random Wikipedia article have had their sentences tokenized and labels as true or false accordingly. There are also written user articles that are have been labeled by hand added in to help combat over fitting the parameters. This test data contains 410 user articles and is located in data/test_data.csv.

Because finding the best parameters is so computationally expensive, I have chosen to use a random search method to provide random parameters to the model. Since the behavior of the model is much like a classification algorithm in this setting, a confusion matrix is able to measure model performance against the test data. After running 100 tests the from a Google Cloud Console’s Compute Engine over 17 hours. Parameters were discovered the show the train data having an accuracy score just over 80%. Unfortunately when testing these parameters on the prototype it proved to be heavily biased and agreed with many ludicrous statements. After a few of these interactions I moved the gate from 3 to 20 and this seemed to produce more reasonable results.

Gate	Weight Mod	Window	Epochs	Vector Scope	Vector Weight
20	1.74327329492608	13	9	12	14.1647849325647

TN	FP	FN	TP	Accuracy	Precision	Recall	False Positive Rate
58	48	31	272	0.806845965770171	0.85	0.897689768976898	0.452830188679245

Conclusion

The approach taken here has proven to be to computationally expensive for the user environment. Based on the above accuracy, it would require more specific training before any chat bot or personal assistant could replace a document. However, this approach would be sufficient at a business level. If the model was maintained with custom scripts and updated with user articles, a chat bot could be put into production for a specific topic. This is likely why we see so many chat bots on business websites, but do not see them for individual users.

Recommendations

If this project is to move forward, it will require maintenance for each instance that is created. There does not seem to be a way to read in a document with enough accuracy in a fast enough way that would provide the end user with a better experience. However, this technology would be useful for businesses that have the resources to maintain a chat bot service and provide that service to their customers.