Kalen Willits
Data Science Portfolio

Teaching Your Computer To Read With Machine Learning

Jupyter Notebook and prototype can be found in the project repository on GitHub.

In today's instant gratification paradigm product documentation is getting replaced with YouTube videos and support calls. The willingness to understand is becoming a lost art. Technical products that require lengthy documentation are struggling to educate their customers. This results in frustration on both ends. I propose that we can supplement this documentation with a well-trained chatbot interface which would increase customer understanding by getting directly to the information they are looking for in a conversational way. Getting a chatbot to find documentation is simple enough, getting it to generalize well on new data is a challenge worth researching.

I envision scanning a document using my smartphone, asking its digital assistant questions about the document, and having it reply as an expert on the subject. This requires an enormous amount of computational power and the clever use of natural language processing. To start our experiment, let's first limit the success criteria to reading in Wikipedia articles and telling us if the article supports a true or false statement provided by us, the user.

There are several parts to this chatbot. First, it's going to have to be trained on natural language, then read in a document, and last the user's input. Now work

backwards using the chatbot's understanding of language to search the read in document for similarities to the user input. If it finds the user input is similar enough to parts of the article, it will return true. For testing purposes, we need an article that contains information that is not too subjective so that there is little debate if the user's input actually is true or false.

I have chosen to use the "Rules of Chess" Wikipedia article for this purpose. Once the article is read in and processed, we can investigate how the chatbot will gain meaning from it. Once all the stop words are removed, we can begin processing. The task of processing an article involves something called tokenization. Tokenization cuts up the article into pieces called tokens. The relationship of these tokens can be measured by various algorithms.

Word tokens as a word cloud generated from Wikipedia's "Rules of Chess" article and processed for NLP.

One such algorithm is the ngrams algorithm. This counts how many times a token appears in an article. Based on the word cloud, if we give the ngrams algorithm the word "move" it will be sure to return an integer symbolizing how many times that word appears. Now tokens don't have to be a single word, they can also be two words, three words, up to "n" words. My thought was we could assign weights to

tokens with more words in them because they tend to happen less often and represent a relevant statement. This algorithm works well for a basic search engine, but what about finding meaning in the words? That is where Gensim's Word2Vec model comes in. Word2Vec calculates a word vector based on the other words that appear around them. This is the part of our strategy that requires a lot of data and power.

Instead of training the model from scratch, we will use a pre-trained model from Google. This model was trained on over 100 billion words and likely will give us better results. Unfortunately, there is not the same option for the looping ngrams model. Once the model is equipped with data, we need to find the right parameters. To avoid burning up my notebook, we are going to use a random search on Google's Cloud Console to run a compute engine virtual machine. Even with cannibalizing training data for a faster performance, it took nearly 17 hours to test 100 iterations of random parameters. The results of the random search are listed below with an accuracy score just barely touching 80%.

gate	weight_mod	window	epochs	vector_scope	vector_weight
3	1.74327329492608	13	9	12	14.1647849325647

Top model parameters as per accuracy on a fullx100 random search.

The gate is the parameter that decides if the user input is true or false. After several tests I found a gate parameter of 20 tends to have better results. This likely is an effect of the over-fitting going on due to the lack of computational resources and data cannibalization. Nevertheless, I used a gate value of 20 in the prototype.

The testing data was designed in 3 sections. One piece was made of the same read

in “Rules of Chess” article as sentence tokens, these inputs should all be predicted as true. The next was a random article from Wikipedia and is expected to be predicted as false. Last was a handwritten section of queries labelled true or false accordingly. We must also understand that true or false is in relation to article relevancy and is not providing absolute meaning. For example, we can ask the chatbot to check if the earth is round, but if we read in the article about chess it should return false. Below is the transcript from that interaction with the prototype.

```
-> Hello!
-> I am the prototype chatbot interface for Wikipedia doc
-> What topic would you like to discuss?

<- Rules of Chess

-> Please allow me a moment to train on rules of chess.
-> Okay I've got it, tell me a fact and I'll tell you if

<- The earth is round

-> That is false.
```

However, if we continue the conversation to query information about the rules of chess, we will be provided with expected results.

```
<- In the game of chess, there are 2 kings, 2 queens, and

-> That is true.
```

Here the chatbot returns true. It has achieved this because the user article contains enough relevant information that appears in the read in article to score our fact above the gate parameter. Unfortunately, a limitation to this algorithm is the understanding of quantity. We can change the numbers in our input to anything we want and produce the same results.

```
<- In the game of chess, there are 16 kings, 16 queens, ar

-> That is true.
```

The deciding factor of this input was the word chessboard. If we increase the number of chessboards and add the “s”, this will return false. The number before the “chessboard” token carries almost no meaning to the chatbot. Hence, the 16 kings and queens, this only adds value to the relevance because each player has 16 pieces and that number appears elsewhere in the article. We could exploit this limitation by repeating top words in the same input.

```
<- move rules move rules move rules

-> That is true.
```

This is where we could bring up the debate if this user input should be classified as true or false. Should we discount it just because it’s not grammatically correct or is the relevancy to the read article enough to count as a true statement? – This is a question to be answered by the whoever uses this model in production and likely would be specific to the nature of its use case.

Which brings us back to the purpose of this model’s development. Is it reasonable to deploy this across a general platform such as a digital assistant? – Likely not. Providing a binary relevance could be an essential building block to an impressive chatbot, but it would require specific scripting and constant training on new user input. The new training input would need to be classified by hand to increase performance to acceptable scores. Over time and with a deep analysis of user data, this chatbot could provide respite for support calls on specific subjects. – But we are a long way from the instant transfer of knowledge that we set out to accomplish.

Given the complications that are introduced by natural language processing it seems that using these techniques for general purpose machine understanding is not practical in a user environment. This is likely the reason we see chatbots providing scripted responses and digital assistants provide answers without understanding. It simply takes too much power to create machine understanding in a reasonable amount of time. The narrower the scope of the chatbot the better. At

least for the time being, we are still going to need to do our own research and create our own understanding of product documentation. However, enterprise level chatbots that are designed for a specific understanding are becoming a proven concept and with the right resources, will prove to be an invaluable asset in the customer service field.

Research and Sources

Unsupervised NLP: How I Learned to Love the Data

Transfer Learning for NLP: Fine-Tuning BERT for Text Classification

A Beginner-Friendly Guide to PyTorch and How it Works from Scratch

A Beginner's Guide to Word2Vec and Neural Word Embeddings

Generating WordClouds in Python

An Essential Guide to Pretrained Word Embeddings for NLP Practitioners

Quickstart using a Linux VM

Deep NLP: Word Vectors with Word2Vec

Google's trained Word2Vec model in Python

Rules of Chess

Top 1000 Wikipedia Articles

Webster's Unabridged Dictionary by Various

Confusion matrix, accuracy, recall, precision, false positive rate and F-scores explained

Google's Pretrained Word2Vec Model

10 Facts about Chess

Top 10 Myths About Chess

Stop Words in NLP

Why Is Random Search Better Than Grid Search For Machine Learning

Wikipedia Random Article Link

Kalen Willits / October 16, 2020 /