Homework 2 - Text Analytics - GitHub Link

Kristiyan Dimitrov - ktd5131

For this homework, I worked with the latest English Wikipedia corpus. It took me approximately one hour to download an 18 GB XML data dump, which I then spent ~2.5 hours converting to a ~7.5 GB JSON file containing ~3.5 million Wikipedia articles. It's important to note that during the conversion from XML to JSON I discarded any articles with less than 1000 characters (aka "stumps") in order to enforce some "quality control" as longer articles are rarely poorly written from a language perspective.

For preprocessing the data (CODE) I did the following:

sentence_tokenize → lowercase → word_tokenize → remove english stop words.

I decided *not* to lemmatize in order to preserve plural forms of words (e.g. "dog" & "dogs"). The next step was the most time-consuming. In fact, I managed to apply the preprocessing described above to only 55,170 articles out of the ~3.5 million. Even so, it took ~8 hours on my MacBook Pro. The reasons for this are primarily the slowness of I/O to the hard disk. More explicitly, I had to read in each article from the json file, preprocess it, and then save each sentence to a new line in an article-specific .txt file. The size of the 55,170 .txt files is 731MB. Extrapolating that size to the entire corpus of ~3.5 million I estimate roughly 46GB of space would be required. Future work should focus on parallelization and batch processing to speed up preprocessing, although I am not sure how this would be achieved given the entire corpus is in a *single* JSON.gz file. Alternatively, I could train the model directly with preprocessed data without writing it to .txt. However, this is not an option for this assignment. [UPDATE]: After Lecture 4 I used the multiprocessing module to parallelize my preprocessing code. This resulted in a roughly **4x** speed up. I was able to preprocess 100,000 articles in ~3.5 hrs (twice the number of articles in half the time). Code is in this notebook.

Training the actual model (CODE) on the 55,170 articles was quick. I tried two models - one with the default settings (CBOW) and a skip-gram one with modified parameters. The quantitative results are summarized in the table below. In either case, the model went through 81,255,439 words in 6,227,939 sentences per epoch. [UPDATE] Training the model on the expanded dataset of 100,000 articles was also fairly quick, although not as quick as the other models, because I selected a higher vector size of 300. It took ~12 minutes.

Model	Training Time	Workers (# threads)	Effective word rate	Vocabulary size	Embedding size	Window (context size)	Min_count	Iter (epochs)
Default (CBOW)	709 seconds	3	562,999 / second	264,452	100	5	5	5
Modifed (Skip-Gram)	1816 seconds	5	424,173 / second	66,165	200	5	50	10
Larger Corpus (CBOW)	728 seconds	10	--	122,749	300	5	25	5

In terms of qualitative evaluation I tried the following word combinations

1. model.wv.most_similar(positive=['jobs'],topn=1)
    - Default model game me: “workforce”
    - Modified model gave me: “employment”
    - Large Corpus gave me: “employment”
2. model.wv.most_similar(positive=['woman','queen'], negative=['king'], topn=1)
    - Default model gave me: “housewife”
    - Modified model gave me: “girl”
    - Large corpus gave me: “prostitute”
3. model.wv.most_similar(positive=['green','red'], topn=1)
    - Default model gave me: “blue”
    - Modified model gave me: “blue”
    - Large corpus gave me: “blue”
4. model.wv.most_similar(positive=['cat','cats'], negative=['dog'],topn=1)
    - Default model gave me: “sphynx”
    - Modified model gave me: ”shorthair”
    - Large corpus gave me: “sphynx”
5. model.wv.most_similar(positive=['male','man'], negative=['female'],topn=1)
    - Default model gave me: “creature”
    - Modified model gave me: “woman”
    - Large corpus gave me: “men”
6. model.wv.most_similar(positive=['two’, 'three'],topn=1)
    - Default model gave me: “four “
    - Modified model gave me: “four”
    - Large corpus gave me: “four”
7. model.wv.most_similar(positive=['continent', 'river'],topn=1)

- Default model gave me: “estuary“
- Modified model gave me: “headwaters” (The headwaters of a river or stream is the farthest place in that river or stream from its estuary or downstream confluence with another river)
- Large corpus gave me: “estuary“

I’m happy with how both models apparently can count “two, three → four”. Both reasonable associate jobs with either “workforce” or “employment”. However, neither one of them was able to recognize plural and singular forms of words (cat + cats - dog → dogs?). It’s also worth noting that the two CBOW models give similar results to each other.

The below notes on RNN, LSTM, Transformers, and BERT are based on these YouTube videos “LSTM is dead. Long Live Transformers!”, “BERT-Explained”, “Transformers Explained”

- RNN & LSTM
  - RNN’s have significant problems with vanishing or exploding gradients due to their recurrent nature.
  - LSTMs are a kind of RNN. They have 2 hidden states. They add on values as you go through the layers, so in some sense they are similar to ResNets. This solves the problem of vanishing/exploding gradients.
  - Another important aspect is that pre-training aka transfer learning doesn’t work very well for LSTMs.
- After LSTMs, Transformers were developed. They contain two general parts:
  - Encoder contains:
    - Input Embedding
    - Positional encoding - encodes some information about the sequential order of words. Otherwise, we would just have a bag of words, which ignores that. This is achieved by layering sine & cosine on top of the input embedding.
    - Multi-threaded attention - typically 8 different attention mechanisms; one for conjugation, one for vocabulary, one for grammar, etc. A given attention mechanism work on the basis of a few matrices: Query, Key, Value. The idea of attention mechanisms is that they allow the network to learn what parts of the input to focus on.
  - Decoders
    - Output embedding (whatever has already been predicted is fed into the Decoder after being embedded.

        - Positional Encoding (uses sine & cosine just like Encoder)
        - Self-Attention block - how much is each word in the sentence related to other words in the translated sentence
        - Results from the Encoder are also fed into a Multi-thread attention layer.
    - Transformers are particularly useful for Language translation. For example, if we're translating from English to French, the role of the Encoder is to understand English and sentence context. The Decoder's role is to map the English words to French words.
    - A key benefit of Transformers is that they have a lot of all-to-all comparisons that can be done fully parallel! This speeds up training significantly and is different from RNNs & LSTMs which have to be computed serially for each token.
- BERT:
    - Already pre-trained on a large amount of data
    - Can be fine-tuned for specific tasks by adding an additional layer and tuning it with application specific data.
    - BERT is made by multiple Encoders stacked on top of each other. That's why BERT is short for Bidirectional Encoder Representation of Transformers. If we stack multiple Decoders on top of each other we will get the GPT architecture.
    - While BERT is good for translation, just like Transformers, it's also good at additional tasks such as Question Answering, Sentiment Analysis, and Text Summarization.
    - Different versions of the BERT pre-trained model have from 110 to 340 *million* parameters. Based on how many checkpoints are used, the necessary hard disk space can range from 1.7 to 7 GB.
    - Similar to Word2Vec it is trained in a self-supervised manner - it tries to predict masked words in the text. This vastly expands the available pool of data i.e. basically all the world's data is valid training data. It also makes it cheaper because you don't need hand labelled data. It also trains itself by doing binary prediction of whether one sentence follows another.
    - LSTMs tend to work better when your dataset is relatively small.