Tanner Lederman

CS4120

Abstractive Automatic Text Summarization On Research Papers

1. Problem Description

For my project, I chose to try to create an abstractive automatic text summarization model trained on research papers. Text Summarization is important because it summarization allows people to get a good sense on what a paper is going to offer them before they dive in depth into this. There is currently so much information on the web that it is simply not viable to read all the relevant information on a topic, so we either have to come up with a better way to intake more information (such as text summarization) or make decisions without all the data.

Automatic Text Summarization is the field of creating models for computers to generate summaries of a text for us without human intervention. There are two main models for generating summaries automatically: Abstractive methods and Extractive Methods. These two methods are similar to a pen (abstractive) to a highlighter (extractive) [2].

Abstractive Methods involve processing the tokens from the source document and outputting a summary from generating new n-gram sequences from the words that are most important to the source document overall. Extractive Methods, on the other hand, take in the n-gram subsequences of a source text and rank the importance of these sequences, basically stringing together the most important subsequences of the model. They do not actually generate their own text summaries like abstractive methods do, but rather choose the most important parts of the text and bring them out- or "extract" them.

2. Data description and preprocessing steps
   a. Dataset

The data I am using comes from the Elsevier OA CC-BY Corpus. It contains over 100,000 research papers, each formatted in JSON for lean storage and relatively easy access. The JSON documents were split by document part: abstract, body_text, bib_entries, and metadata. Each category has its own subcategories and unique structures, which made the data a little annoying to work with, but the json format at least kept it pretty organized. For the purposes of this project, however, I only used the body_text and abstract portions of the data. I used the body_text as the X features, or the source documents. The abstract is the y target variable, or the desired summary. For my model, I chose to use the first 1000 papers as training sets. Even though this number was chosen randomly, the data is sufficiently large for the task, as each paper has on average 6,250 words.

   b. Preprocessing Steps

In order to run the model. The first thing I did is make all the textual data lowercase and removed all punctuation. I had to split the data into word tokens. I chose to keep them as whole words and split by whitespace instead of using nltk.word _tokenize because I felt that it would negatively impact the summary if there were tokens being generated that were not full words.

I had to cut some data and set some limits before I started my model. First, I cut down the vocabulary from about 110,000 to the top 10,000 most prevalent words. Second, I set the start input length of a source document for the Recurrent Neural Network to be 7000 words, which is considered the upper bound of the average length of a research paper. Lastly, I set the summary size to be 200 words, as the average length of a paper's abstract is 150-250 words.

3. Recurrent Neural Network Models
    a. Basic Encoder Decoder RNN with Attention Model

The Recurrent Neural Network (RNN) is well known for being good at handling sequence to sequence tasks. The Encoder Decoder Model is really two different RNNs in one: an encoder RNN and a decoder RNN.

The encoder portion takes in the source text as an input and "encodes" them, often through the use of an embedding and a LSTM, which is the case in my basic encoder-decoder RNN model. The decoder portion takes in a partial summary which in the first pass starts with a special <START> token. It goes through the same layers as the encoder to create the decoder hidden states. Once we have both the encoder and decoder hidden states, we pass them both into the attention distribution, which is a probability distribution over the words in the source text. This creates a "context vector", which is the weighted sum of the result from the encoder hidden layer state after the attention distribution. Then, the context vector and the decoder combine to create the vocabulary distribution and the word with the largest probability is the next word in the sequence.

This method comes with a few problems. The first of which is that it may not report facts accurately, as it does not know what actually happened, just a calculated result. The second major problem with this model is that summaries often repeat themselves.

(section summarized from source 2)

    b. Pointer-Generator Model

The pointer generator model is a variation and improvement on the basic encoder-decoder RNN. It has all the basic parts of the encoder-decoder RNN, but it performs some extra operations on top. After calculating the context vector, we calculate a final word probability for each word:

$$P_{\text{final}}(w) = p_{\text{gen}} P_{\text{vocab}}(w) + (1 - p_{\text{gen}}) \sum_{i:w_i = w} a_i$$

P_gen is a scalar value between 0 and 1 that represents the probably of generating a word from the vocabulary vs copying a word from the source. So, for example, if you vocabulary only contained the set of words in this document, the P_gen would be 1. The vocabulary distribution –

which was the result of the basic model – is multiplied by P_gen. Then, the summation of the attention distribution is multiplied by (1 – P_gen).

This model has several advantages compared to the basic model. The first is that it makes it easy to copy words from the source text. It simply needs to have a reasonably high attention on the relevant words and make P_gen reasonably large. This may make the model a small bit less abstractive, but it is the most important. The second is that the model is able to copy out-of-vocabulary words from the source text. This is vital as it allows us to handle unseen words while allowing us to use a smaller vocabulary. These models use a lot of computational power and space, so any shortcuts we can take in that regard is very helpful. Third, the pointer-generator model is faster to train than the basic model.

(section summarized from source [1])

c. Strategies I've tested

I have tried three main approaches: two based in the pointer-generator model and one basic encoder-decoder RNN model. The first approach I tried was a full pointer-generator model using the first 7000 words of the source text (padded with 0s if the document was not 7000 words) then in my encoder RNN I pass the words through a Word2Vec model embedding and an LSTM. I also take the current summary and run it through a default Keras embedding layer and a default LSTM. The goal was to output a vocabulary distribution and a separate attention result and run them through a Dense Layer to get the final distribution, which I then finished off by running through the attention distribution. Also, to make things a little bit simpler for myself, I simply set P_gen to be 1, so the attention distribution is only used where it would normally be used in a basic RNN and makes me assume that all our words are in our vocabulary.

The second approach I tried was basically the same as the first approach but I trimmed the vocabulary down to 10,000 words. This lowered the computational complexity a lot: from about 91 million parameters in the model to about 8 million. There are two major problems I kept running into with these two approaches: the variable input size into the decoder with the current summary and the size of the output being reduced to one from vocab_size without any real warning.

The third approach I tried was a basic encoder-decoder RNN. This model took in the source text through an embedding layer of Word2Vec embeddings. Then, it went through a LSTM layer that scaled the data down to the size of the summary. Since Keras cannot take in recursive loops where the output is fed as the input to the model for the next iteration, we have to use alternative approach. Instead of through a loop, in this model, I chose to create the summary all at once. In this model, the summary would have been the top 200 words produced from this distribution, which is definitely less than ideal.

d. Other Summarization Model- Latent Semantic Analysis (Extractive)

Latent Semantic Analysis is a possible alternative model I can explore. It is a extractive model that uses unsupervised learning to extract a representation of text semantics based on observed words. It creates a base matrix where the rows are the words (n words) and the columns the sentences(m sentences). Each entry (i,j) is the weight of the word i in the sentence j. These weights are usually computed using the TFIDF technique and if a sentence does not contain a word the weight is zero. It subsequently splits this in to three matrixes: U, the base matrix, Σ, the diagonal matrix where the row i (topic) corresponds to the weight of the topic in the document/corpus. V^T is the topic-sentence matrix. Once we have created that, we take the number of topics we feel are necessary to create a summary and select the sentences that convey this information.

(paragraph summarized from [4])

4. Results
    a. Summarization Results.

I was not able to obtain results from my model, as I had trouble getting my model to compile and fit. In general, these models should produce summaries of in one of two ways: either of set length or until an end of sequence token is returned from the model. In my case, I chose to do a set length.

i. Abstractive vs. Extractive

In general, results for an abstractive model will be a little less human readable and will be much more variable than the extractive alternatives. However, the abstractive method tends to perform better than the extractive alternatives. Abstractive models are highly variable and are prone to repeating themselves over and over. However, they have the potential to bring together parts of the data to create a summary truer to the document than extractive models. Current models, however, while are getting better than most extractive methods, are still far from perfect.

Extractive models are simpler and safer. They generate sequences directly from the source text, so the challenge of an extractive method is finding the most important subsequences in the text. The genius- and problem- of these types of models is that it comes directly, word for word, from the source document. It cannot extrapolate or aggregate contextual clues to find the most all-encompassing summary. (paragraph summarized from [4])

b. ROUGE

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is an automatic evaluation method for summaries. There are four different ROUGE measures: ROUGE-N, ROUGE-L, ROUGE-W and ROUGE-S. These methods, at a high level, count the number of overlapping units such as n-gram, word sequences, and word pairs between the computer-generated summary and an ideal summary created by a human. ROUGE-N focuses on n-gram recall between the two summaries. It is calculated by taking the sum of the overlapping n-grams and dividing it by the sum of the count of n-grams between both summaries. ROUGE-L focused on comparing the longest common subsequence in each sentence in both the computer-generated and the ideal summary. The measure used is a LCS-based F-measure to compare similarity. ROUGE-W is the weighted version of ROUGE-L. The weighting in ROUGE-W is favors consecutive n-gram sequences over non-consecutive sequences. The further apart the n-grams, the lower the score in ROUGE-W. ROUGE-S measures skip-bigram co-occurrence.

(Summarized from [3] section 3.3)

5. Conclusions, Challenges, and Future Work

a. Conclusion:

Abstractive Text Summarization is hard and computationally expensive.

b. Challenges:

I faced quite a few challenges during the project. The first one I faced was parsing the JSON data, it was very nested and a little hard to extract without losing a lot of information. The second error I got was that running the model was overloading my application data. This happened for two reasons: the first was that my generator function (generateStepper) had an infinite loop, and the second was that my original model was so big that just compiling it caused it to overload the memory space on my computer. The third challenge I faced was scaling down the model, I handled this by making assumptions about what was necessary to create a summary: I trimmed the vocabulary to the more prevalent words, I shortened the summary from 200 to 20 words for my basic model, etc. The final challenge that I was not able to overcome was properly compiling and fitting my model. The error messages were not very readable. When my model would compile, it would generate the wrong vector size for the output, the input sizes were off, etc. And when it wouldn't compile, it would say that the graph was disconnected, the gradients were off, or the loss function "categorical_crossentropy" does not exist (it does), to name a few.

c. Future Work:

I want to get at least the basic encoder-decoder RNN to compile and fit at some point. I am really curious to see the results that I would get, even if I predict that they won't be that great. I also might try the extractive method I looked at, as it seems much more straightforward.

Works Cited

1. See, Abigail et. Al. "Get To The Point: Summarization with Pointer-Generator Networks" https://arxiv.org/abs/1704.04368
2. See, Abigail "Taming Recurrent Neural Networks for Better Summarization" http://www.abigailsee.com/2017/04/16/taming-rnns-for-better-summarization.html
3. Lin, Chin-Yew. "ROUGE: A Package for Automatic Evaluation of Summaries" https://www.aclweb.org/anthology/W04-1013.pdf
4. Allahyari, Mehdi. "Text Summarization Technioques: A Brief Surbey" https://arxiv.org/abs/1707.02268