Northeastern University
Khoury College of
Computer Sciences

Review Summarization with Pointer Gen and BERT

DS 4420 Final Project

Matthew Martin, Marjolein Pawlus

Khoury College of Computer Sciences, Boston, MA, U.S.A
Submitted 14 April 2020

Abstract

In this paper, we explored abstractive summarization through implementing a Pointer Generator Network while also incorporating the BERT transformer. The Pointer Generator Network (PGN) is based on the model in *Get to the Point: Summarization with Pointer Generator Networks*[1] by Abigail See. To modify the model, the BERT transformer was used as the encoder instead of a normal LSTM since BERT did not exist when initial PGN was developed . The basis behind this was that BERT could help decrease training time while improving summarization quality. The dataset used was Amazon product reviews from 2018, specifically from the software sector. The dataset consisted of over 12,000 reviews and abstractive summaries. Based on our dataset and training capabilities our model performed adequately. The model trained for 48 hours and was able to learn some meaning to produce outputs that made sense for the data at hand.

I. Introduction

In today's world there is an exuberant amount of textual data flowing into a company that they need to monitor. In most cases these texts will be lengthy and will take a lot of time for someone to read and extract meaning from. With the data inflow of the current day it is humanly impossible to read and summarize every text. Recent advances in Natural Language Processing research has now made it possible to summarize these vast amounts of texts whether that be through extractive or abstractive methods. Extractive methods generate summaries by directly taking from the source text, whereas abstractive summaries generate novel words and sentences in a paraphrasing manner, such as humans do. The extractive method is much easier to implement as it is certain to be grammatically correct. However, a lot of meaning is lost with extractive methods. There are various abstractive summarization methods such as sequence-to-sequence learning and text rank. We decided to expand on the PGN approach by using a transformer encoder, specifically the BERT transformer on review texts.

II. Experimental Setup

A. Data Preprocessing and Exploration

The dataset consists of reviews on software products from Amazon. The dataset was developed by Jianmo Ni from University of California: San Diego in 2018. The reviews and summaries in the dataset had varying lengths. The review lengths ranged from 1 word to over 5000 words. Summary lengths ranged from 1 word to 27 words. To preprocess the data we excluded any reviews or summaries that were null or empty strings. Also, BERT has a max token size of 512, so any review more than 512 words was removed. In addition we removed any reviews that were less than 4 words and any that had summaries less than 3 words. A distribution of review and summary lengths can be seen below.

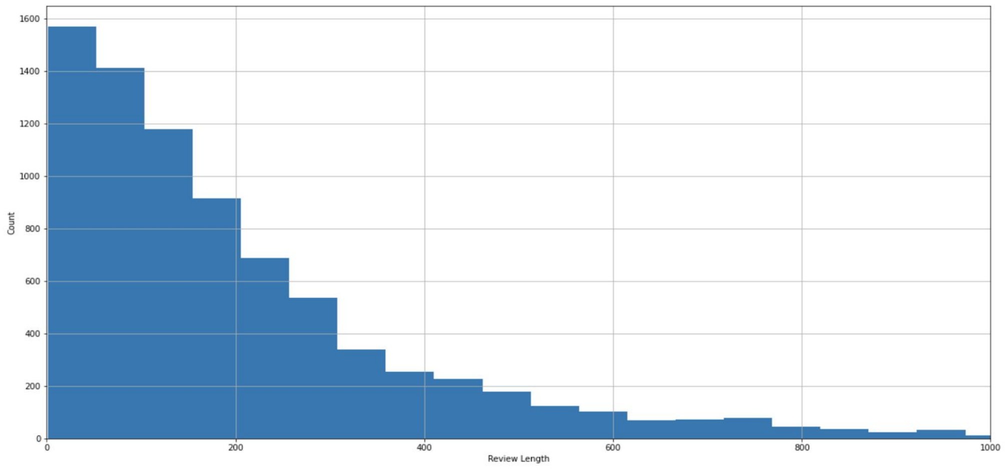


Figure 1: Review Length Distribution

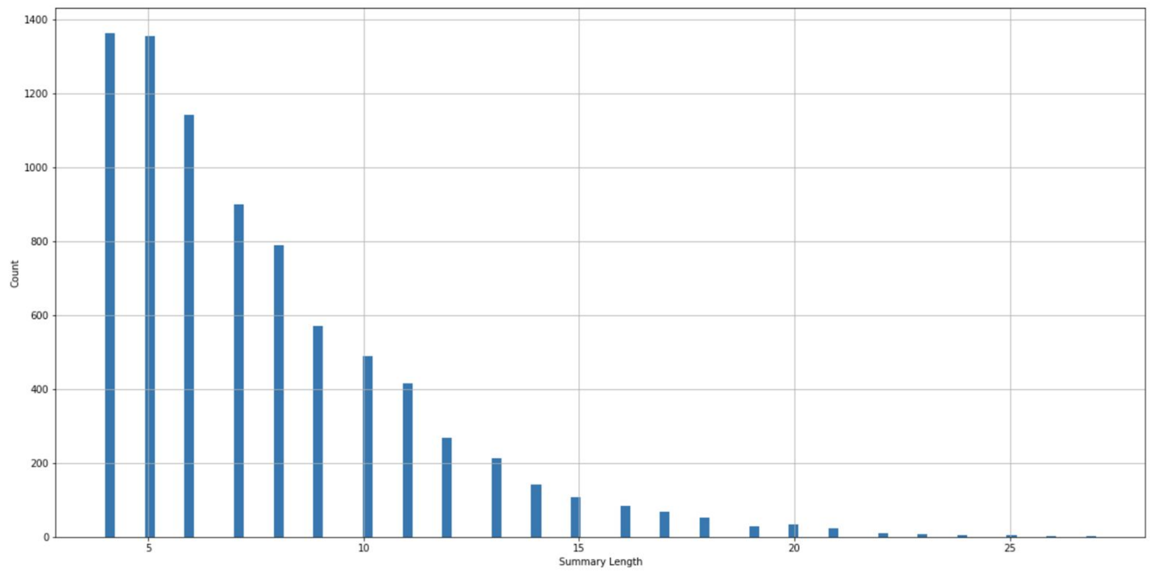


Figure 2: Summary Length Distribution

There were many preprocessing steps that needed to happen due to the architecture of our model. First we needed to create a vocabulary of all the words in our dataset, essentially giving each word a numeric identifier. Next, we needed to tokenize the reviews and summaries. The reviews were tokenized using the BERT tokenizer. One issue this causes is that it uses the WordPiece method that breaks words up into subwords. This can be seen below in figure 3. This would lead to messy summarizations if we were to use the BERT vocabulary and tokenizer for our outputs. To remedy this issue we created a boolean mask that took the first word piece after the text was encoded by BERT tokenizer. This ensures there is one to one word length with the encoder output and the original text. In addition, to implement the PGN we needed to encode our texts using our self-created vocabulary and also create temporary additions to the vocabulary so the model can handle out of vocab (OOV) words. OOVs are words that are not included in our self created vocabulary but appear in source texts.

Original Text

- "Kaspersky continues to do the job right."

BERT Tokenization

- ['ka', '##sper', '##sky', 'continues', 'to', 'do', 'the', 'job', 'right']

BERT Encoding

- [101, 10556, 17668, 5874, 4247, 2000, 2079, 1996, 3105, 2157, 102]

Figure 3: BERT Encodings

B. BERT Encoder

In 2017, researchers at Google Brain published the paper *Attention Is All You Need*[2]. In the paper they introduced a new architecture in the field of natural language processing, the Transformer. Transformers are simple network architectures based solely on attention mechanisms. The transformer architecture utilizes the encoder decoder structure and raised the question if they are the new alternative to long short term memory (LSTM) networks. Transformers deal with long term dependencies better than LSTMs. In the paper the architecture consists of six encoders and six decoders. Each encoder shares the same structure but uses its own weights. The structure consists of one self attention layer and one feed forward neural network (FFNN). The same goes for the decoders. The key difference is that each decoder also has an encoder-decoder attention layer in between the attention and FFNN layers. Self attention helps incorporate relevant understanding based off of other words for the word that is being encoded. Specifically, the transformer uses multi-head attention. Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions by stacking separate attention layers in parallel and using different transformation functions in each. Each encoding and decoding layer contains residual connections around each sublayer and a normalization layer after each sublayer. A diagram of the transformer architecture can be seen below.

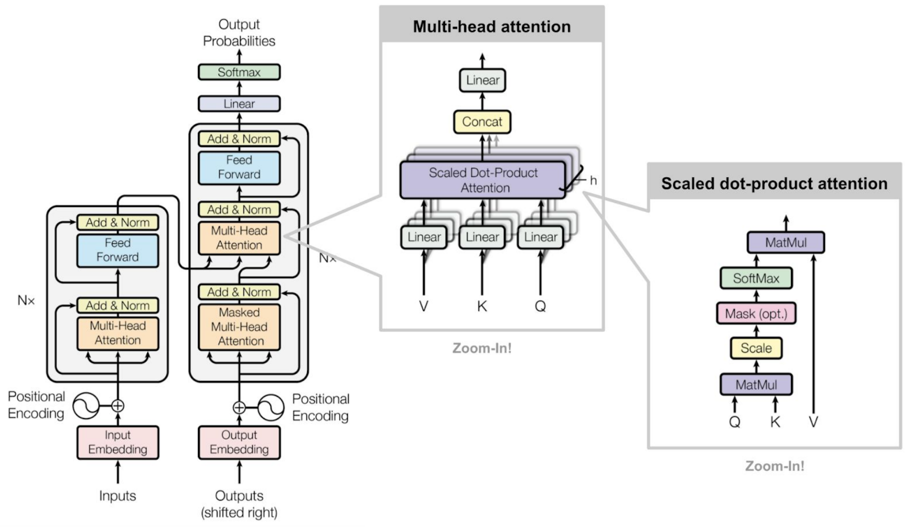


Figure 4: Transformer architecture[2]

In 2018, researchers at Google released *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*[3], in which they revealed a state of the art pre-trained model based off of the transformer architecture. The goal of BERT was to enable bidirectional training in order to get context from both the left and right. This is done by reading the entire sequence at once. BERT uses positional embeddings to get a sense of location in the sequence. Another advantage to BERT is that it is pre-trained through unsupervised tasks, masked language modeling and next sentence prediction. This makes BERT adaptive to new tasks since you do not need to retrain from scratch. BERT's architecture is essentially a transformer encoder stack, as previously described. Specifically BERT has 12 encoder layers in its base version. Additionally, BERT uses larger FFNNs (768 hidden units) and more attention heads (12) than the transformer proposed in *Attention Is All You Need*[2]. The output will be 768 hidden units for each position in the sequence. The resulting dimensions are (batch x sequence length x hidden units). This output is then fed into a decoder for our task of summarization. For the purposes of this project we used DistillBERT, which is a distilled version of BERT provided by Hugging Face, Inc, in order to not max out runtime due to the model's size. DistillBERT has 40% less parameters than BERT, but retains 95% of performance. The model has only 6 encoder layers and 66 million parameters.

C. Pointer Gen Implementation

In 2017, a major advancement was added to the field of text summarization via the paper *Get To The Point: Summarization with Pointer-Generator Networks*[1]. The paper introduces an alternative to sequence-to-sequence models for abstractive summarization through a pointer generator network. The model can copy words from the source text via *pointing* to retain information and can produce novel words via the *generator* words from a fixed vocabulary. In addition, coverage was added so that summaries were not repetitive. The paper uses a bi-directional LSTM encoder and unidirectional LSTM attention decoder architecture. For our model we are using a transformer encoder, via DistillBERT, and a unidirectional LSTM attention decoder. The transformer creates hidden representations of the source text to be fed to the decoder. First these representations will go through an attention module. The attention distribution, a^t, is calculated by the equations below.

$$e_i^t = v^T \tanh(W_h h_i + W_s s_t + b_{\text{attn}})$$
$$a^t = \text{softmax}(e^t)$$

Figure 5: Attention Distribution

The attention distribution helps create the weighted sum of the hidden states with their respective attention weights is known as the context vector, h^*. This is then concatenated with the current decoder state, s_t, and run through two linear layers to generate a vocabulary distribution for the next word, P_{vocab}. The equation for P_{vocab} can be seen below.

$$P_{\text{vocab}} = \text{softmax}(V'(V[s_t, h_t^*] + b) + b')$$

Figure 6: Vocabulary Distribution[1]

Next, a generation probability, P_{Gen} needs to be calculated. This generation probability determines how likely the next word in the sequence is to be generated from P_{vocab} versus copied from the input sequence. This is computed from the context vector h^*, the current decoder state s_t , the decoder input x_t. The equation for calculating P_{Gen} can be seen below.

$$p_{\text{gen}} = \sigma(w_{h^*}^T h_t^* + w_s^T s_t + w_x^T x_t + b_{\text{ptr}})$$

Figure 7: Generation Probability[1]

This is then multiplied with the vocabulary distribution. To incorporate out of vocabulary (OOV) words, the attention distribution, a^t, is multiplied by (1 - P_{gen}) to create a score of copying a word from the input text by sampling the attention distribution. This is then added with an extended vocabulary, a vocabulary that includes OOV words from the source text, multiplied by the P_{gen}. By doing this, our model has the ability to produce OOV words. This summation is the probability of the next word, *P(w)*. The equation for *P(w)* can be seen below.

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

Figure 8: Word Probability[1]

A diagram summarizing these points and the Pointer Gen model architecture can be seen below in figure 9.

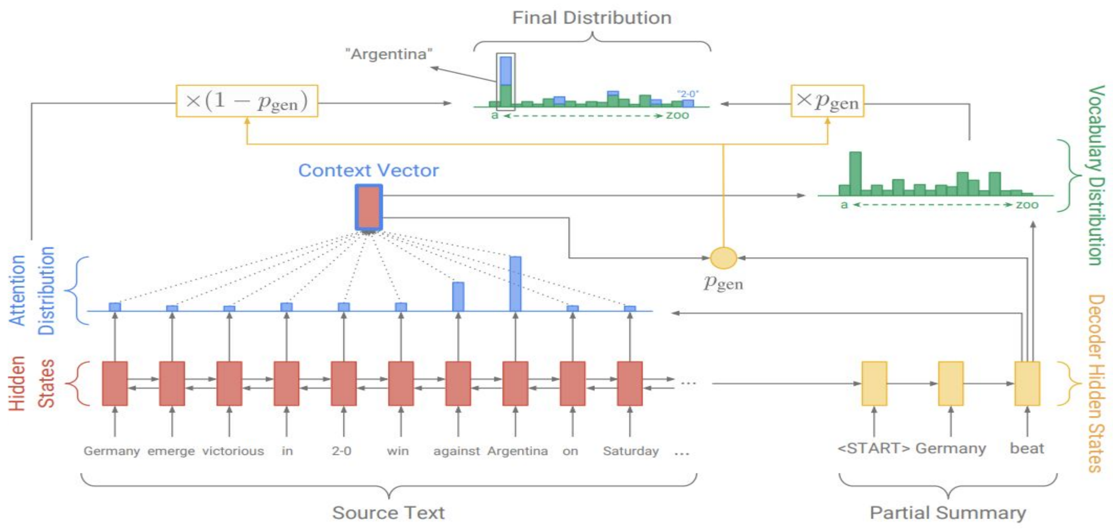


Figure 9: Pointer Generator Network architecture[1]

Next, to implement coverage we created a coverage vector, c^t, which is a weighted sum of the attention distributions over the decoder timesteps. The coverage vector is initialized to be a zero vector for the first time step. This will modify the attention distribution calculation for above to include the coverage vector. This modification can be seen in the equation below.

$$e_i^t = v^T \tanh(W_h h_i + W_s s_t + w_c c_i^t + b_{\text{attn}})$$

Figure 10: Attention distribution with coverage[1]

To monitor loss for the model we use a summation of word and coverage loss. For word loss we simply take the negative log likelihood of the target word. This can be seen in the equation below.

$$\text{loss}_t = -\log P(w_t^*)$$

Figure 11: Loss for word at timestep t[1]

The coverage loss helps to penalize attending to the same locations. The calculation of coverage loss can be seen below.

$$\text{covloss}_t = \sum_i \min(a_i^t, c_i^t)$$

Figure 12: Loss for coverage at timestep t

The final loss for the model can be seen in the equation in figure 13. Lambda is set to 1.0 in our Model. Our model has a total of 83,222,177 parameters. The encoder has a hidden size of 768, an embedding size of 512, and a vocabulary size of 30,522. The decoder has a hidden size of 256, an embedding size of 512, and a vocabulary size of 19,360. The decoder also has a dropout layer with a dropout probability of 0.1 on the outputs of the decoder LSTM. This will help with regularization of the model. The model uses the Adagrad optimizer with a learning rate of 0.01. This learning rate was determined after testing various learning rates to see which returned a lower initial loss for the first few epochs. In the model we only backpropagate on the decoder parameters, due to an overfitting imbalance that would occur due to the DistillBERT encoder. We based our implementation off of the code from the original *Get to the Point* paper[5], a pytorch implementation[6], and *Text Summarization with Pre Trained Encoders*[4] a paper that uses pre-trained encoders for summarization.[7] The model was run using hosted GPUs provided by Google Colab.

$$\text{loss}_t = -\log P(w_t^*) + \lambda \sum_i \min(a_i^t, c_i^t)$$

Figure 13: Total model loss at timestep t [1]

III. Results and Discussion

The model was trained using an Adagrad optimizer with a learning rate of 0.01. Training lasted for 1450 epochs with a batch size of 8. Each epoch took a little less than two minutes to train. This lead to a total training time of about 48 hours. The training loss steadily decreased and then plateaued off. The validation loss was much lower than the training loss and stayed around a constant value for almost all of training. A plot of the losses over training time can be seen below. A major difficulty with running the model was having Google Colab disconnect during training at random points. To combat this we built in fail safes to save the model every 10 epochs.

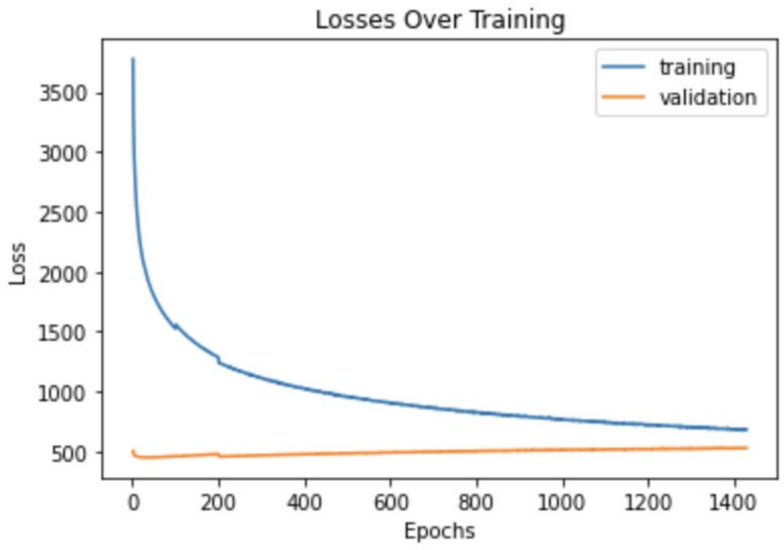
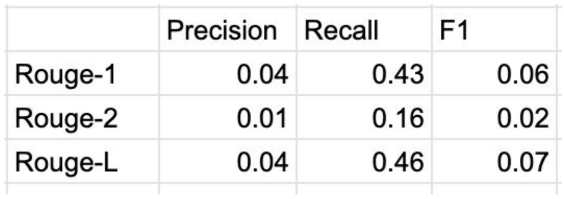


Figure 14: Losses over training

To decode our outputs during test time we implemented a beam search method with using a k value of 5. The beam search method was developed from assignment 4[8] of DS440 Spring 2020. This yielded much better results than with a greedy argmax decoding strategy. Decoding the summaries of the test set also took a substantial amount of time. To evaluate the generated summaries we utilized rouge scores. The Rouge1 Precision score of 0.04 means that 4% of unigrams in the source text are contained in the generated outputs. The recall score of 0.43 means that 43% of unigrams in the generated summaries are in the source text. Rouge2 follows the same logic, but refers to bigrams in the source out generated text. The RougeL score looks at sentence level structure similarity.



	Precision	Recall	F1
Rouge-1	0.04	0.43	0.06
Rouge-2	0.01	0.16	0.02
Rouge-L	0.04	0.46	0.07

Figure 15: Rouge Scores

IV. Conclusion/ Future Work

One of the challenges we ran into was the size and quality of our dataset. With only 8,000 reviews, sometimes very short in length, the quality of our summaries was not as great as was expected. Text summarization is still relatively new to the NLP world and therefore public datasets are limited. The Amazon reviews we used were of varying lengths as well as vocabulary. While we chose only technology-related reviews, a large number of them were vague enough that they could have applied to any product. Our biggest limitation was computing power and time. In the future, we'd love to try our implementation out on a much bigger dataset with higher quality text. We tried running it on a dataset with 24k reviews, but it took 15 min per epoch which made it not a worthwhile option to pursue given our time constraints. Another limitation was BERT's tokenizer can only handle a maximum of 512 tokens. This meant that we had to cut several of our reviews short and therefore lost some meaning in those. While we decided to not implement something to handle these longer reviews due to the time constraints of the project, this is something we would like to add in the future.

References

1. See, Abigail, et al. "Get To The Point: Summarization with Pointer-Generator Networks." *ArXiv:1704.04368 [Cs]*,Apr.2017.*arXiv.org*,http://arxiv.org/abs/1704.04368.
2. Vaswani, Ashish, et al. "Attention Is All You Need." *ArXiv:1706.03762 [Cs]*, Dec. 2017. *arXiv.org*, http://arxiv.org/abs/1706.03762.
3. Devlin, Jacob, et al. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." *ArXiv:1810.04805 [Cs]*, May 2019. *arXiv.org*, http://arxiv.org/abs/1810.04805.
4. Liu, Yang, and Mirella Lapata. "Text Summarization with Pretrained Encoders." *ArXiv:1908.08345 [Cs]*, Sept. 2019. *arXiv.org*, http://arxiv.org/abs/1908.08345.
5. https://github.com/becxer/pointer-generator
6. https://github.com/atulkum/pointer_summarizer
7. https://github.com/nlpyang/PreSumm
8. https://colab.research.google.com/drive/1V24z4u7yWOs60RNcm9wLDMHBAaAncQN9