Constructing QG and QA Systems Using T5 and BERT

Ihita Mandal
Carnegie Mellon University
imandal@andrew.cmu.edu

Raashi Mohan
Carnegie Mellon University
raashim@andrew.cmu.edu

Shravya Nandyala
Carnegie Mellon University
snandyal@andrew.cmu.edu

Pranav Rajbhandari
Carnegie Mellon University
prajbhan@andrew.cmu.edu

Abstract

In this project, we utilized pre-trained T5 and BERT models in order to perform the tasks of question generation (QG) and question answering (QA). The task of question generation is to create questions given a document as context. The task of question answering is to provide the best answer given a question and a document. We found that the methods we created generally worked well in their respective tasks. The QG system usually produced reasonable well-phrased questions that asked meaningful answerable questions. The QA system would accurately select the segment of the document that best answered the question.

1 Introduction

In our final project, we implemented a QG system and a QA system.

Our QG algorithm first divides our document into sections and uses a keyword extractor to generate "answers" for each section. It then sends the keyword and the context from which it came into a pre-trained T5 model to generate a question.

Our QA system first ranks the sentences in the document in order of similarity to the question using the Naïve Bayes algorithm. The system then iterates through them, using a pre-trained BERT model to extract an answer, and to compare answers to find the best one.

After testing out a various versions of both systems, we found that these algorithms gave the best and most accurate results in terms of the questions and answers that were generated. We also suggest that if further explored, improvements we could make to the systems could include adding support for binary answers and implementing a method for generating questions of varying levels of difficulty.

2 System Architecture

Our system architecture is broken into two independent components: a QG system and a QA system.

2.1 Question Generation

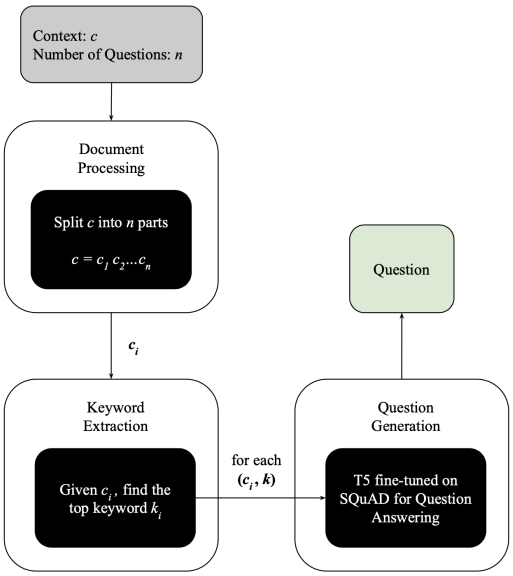

Figure 1: Question Generation System Diagram

Given a context (in this case, a plain text document), the QG task is to generate n unique questions. To accomplish this, our question generation system consists of two modules: the keyword module and the generation module. In the first step, the context is broken down into n sections–we will call each of these sections a clipped context. A clipped context is at least a sentence long. If the document contains fewer than m sentences where $m < n$, the system is capped at m clipped contexts, thus generating m questions. For each clipped context, the text is fed through the keyword module where the top keyword[1] is identified. This is accomplished using the Yet Another Keyword Extractor (Campos et al., 2020) with max ngram size 3 and deduplication threshold 0.3. The top keyword becomes the answer for the question we will generate. Next,

[1]Really, since max ngram size = 3, this is a key phrase. But we will refer to it as the keyword for simplicity.

the clipped context and the keyword are inputted into the generation model–a T5 base fine-tuned on SQuAD for Question Generation (Romero, 2021). From this, a question is returned, thus leaving us with a question-answer pair. We return the question to the user and repeat for all n questions.

2.2 Question Answering

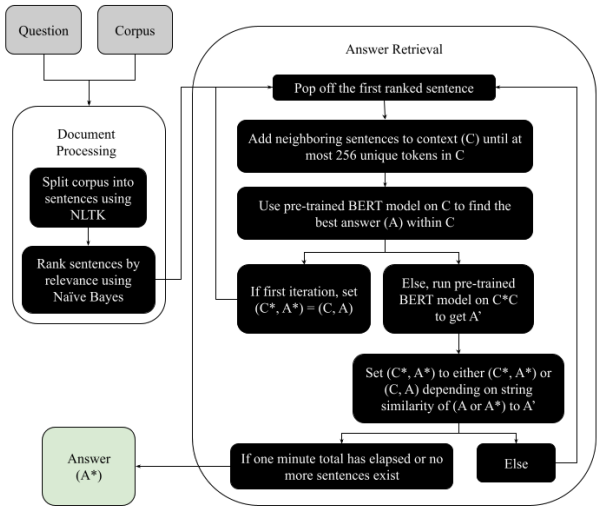

Figure 2: Question Answering System Diagram

Given a corpus (sourced from Wikipedia), and a set of questions, the QA task was to generate the best answer to each question according to the corpus. To accomplish this, our algorithm first uses NLTK (Bird et al., 2009) to split the article into sentences. We then sort the sentences according to similarity to the given question. For this preliminary ranking, we used a Naïve Bayes method.[2] After this, we created the following algorithm to use a pre-trained BERT model (which takes inputs of the question and a context) to search through the sentences, and return the best answer in the first n ranked sentences (and their surrounding contexts):

- For each new sentence, we build up a context C by adding neighboring sentences, keeping the total number of unique tokens less than or equal to 256. (We chose this number since the BERT model could take up to to 512 tokens.)
- We then run the BERT pre-trained model on C and the given question to receive our candidate answer A.

  If this is the first sentence, we will set $(C^*, A^*) = (C, A)$ and continue.

  Otherwise, we need to compare this context-answer pair with our current best context-answer pair (C^*, A^*). We do this by concatenating the two contexts and running the BERT model on this new string[3] C^*C, returning an answer A'.

  We take the combined answer and use it to select the best context-answer pair depending on whether A or A^* was most similar[4] to A'.

  We then set (C^*, A^*) to either the (C, A) or the old (C^*, A^*) depending on the answer we selected.

- We continue to iterate over all ranked sentences until about a minute had passed. We then returned the best answer we had, A^*.

3 Experiments & Results

3.1 Question Generation

Our initial plan was to construct symbiotic QA and QG systems – QG would generate a question using a BERT-SQG model (Chan and Fan, 2019) fine-tuned on the class generated dataset which was then fed into a working QA system. If the QA system outputted a decent answer, we would know that our QG system had generated a good question. However, we discovered early on that this system was unlikely to succeed because this would require complete faith in our QA system to perform well on good questions, and poorly on badly phrased questions. This is not necessarily true, since the QA system is not very much affected by the phrasing of the question when finding an answer.

Another challenge was that our class dataset was in a different format than what was required for our model. To fine-tune the BERT-SQG model, we needed a question-answer pair in addition to the token location where the answer could be found in the context. Because the location information was missing, we found ourselves unable to fine-tune BERT-SQG.

So, we updated our plan to the following. We first take a given context and process the first n sentences. For each sentence, we identify the top keyword (max ngram size = 3) using Yet Another

[2]We used Naïve Bayes since it was time efficient, and would likely get the relevant sentences in the first few places.

[3]This combined string C^*C is sure to have less than or equal to 512 tokens since it is composed of two strings C and C^* that each have less than or equal to 256 tokens

[4]Experimentally, string similarity was not required, as the answer was always one of the candidate answers, but we included it to be robust

Keyword Extractor. This keyword is now considered the answer. We then feed the context and answer through Google's T5 model fine-tuned on the SQuAD dataset (Colin Raffel and Liu, 2020). This would then output a question, leaving us with a question-answer pair for the given context.

This worked well on some sentences. However, in a sentence where there was no named entity (i.e. only pronouns present), the resulting question would also lack a concrete subject. Thus, we needed to iterate and improve this prototype.

To solve this, we instead divide the document into n sections, keeping the rest of the process the same (ideally the document length is greater than n sentences so each of these sections would be greater than one sentence). We observed that this change drastically improved the quality of questions generated. Upon inspection of the questions by our team members, all questions were grammatically correct with only a few instances of questions lacking concrete meaning (i.e. a question such as "What is it?").

3.2 Question Answering

Our question-answering system underwent a series of improvements throughout the course of the semester to answer a wider variety of questions at a higher degree of accuracy.

- Initially, we tried using a pre-trained BERT model to generate an answer given a document and a question. This performed well when the answer was within the first 512 unique tokens (about a paragraph or two).

  However, this implementation did not perform well if the question referred to a later part of the article. We found that the BERT model originally used could only take 512 unique tokens, and thus it truncated the article. Since the Wikipedia articles contained much more than 512 unique tokens, this method failed to perform well.

- In response, we attempted to rank each sentence using the TF-IDF statistic, taking the first n resulting sentences and using the BERT model on this set. This implementation worked better, as it could now take an answer from anywhere in the article.

  However, a problem with this algorithm was we could no longer consider the context around each sentence, so multi-sentence questions could not be answered well.

- To allow our model to answer multi-sentence questions, we altered this method to include the context around each sentence (the neighboring sentences) when finding an answer with the BERT pre-trained model. This method worked better for test cases that required multi-sentence answers.

  However, we found that the answer to the question was now mainly drawn from the first few sentences in our set, which was not the most accurate system.

- To solve this problem, we created an algorithm to directly compare the ranking results. We restricted the context around each sentence to be at most 256 unique tokens. We then could compare the answers drawn from two contexts directly by running the BERT question-answering model on the joined contexts.

  The reasoning behind this method was that the BERT question-answering system, when presented with contexts containing each answer, would pick the best answer among them. This allowed us to compare any two contexts to see which had the better answer. In theory, we could search through all possible contexts to find the "maximal" answer, but to keep the system's runtime within this century, we trusted the ranking to give the relevant sentence within the first n sentences (n was set to keep the answer time under a minute).

  One case that was not handled in our code was that in which the BERT system provided an answer from the joined contexts that was not related to either of the proposed answers. We assumed that this would not be a problem, and used string similarity to choose the best answer. Experimentally, this worked out since in all of our test cases, the new answer was one of the proposed answers.

  We also changed to using Naïve Bayes for the ranking system since, in test cases, it performed better than our implementation of TF-IDF.

3.3 Attempting to Connect the QA and QG Systems

3.3.1 QG → QA

After creating our two systems, we attempted to use the results of the Question Generation system to train our Question Answering system. We started by generating questions from various Wikipedia articles.

However, by using the YAKE library to search for keywords to generate questions, the answer to the generated question became a "keyword," which, by design, were very common in the article. This made narrowing down the instance of the answer that generated the question difficult. This was a problem since the training algorithm required us to provide answer locations in addition to the answer.

We attempted to solve this by searching through the article to select the sentences that were most similar to the question-answer pair. We did this using a Naïve Bayes algorithm that accepted a sentence as the relevant location of a question-answer pair if it was over a certain threshold.

Using this, we created a training set on the scale of 10^3. However, after fine-tuning our QA system on this training set, we noticed that our model performed similarly or worse than before we fine-tuned it. Additionally, after failing at this task, we improved our QA algorithm to a more functional version, so we decided that this fine-tuning was not necessary.

3.3.2 QA → QG

As mentioned in section 3.1, our original plan was to use the QA system to evaluate output of the QG system in addition to using QG to train QA (similar to the spirit of the GAM algorithm). We planned evaluate the QG system on whether the QA system correctly answered the question, giving it a positive reward if it was a similar answer and a negative reward if it was not.

However, the models we ended up using were pulled from various packages, which defined training functions for the neural models. This training architecture was not suitable for the simple positive/negative reinforcement learning system. In addition, we encountered the problem described above, where we were not sure if our QA model could distinguish if a question was well formulated when answering it. Thus, we decided not to improve the QG system in this way.

4 Discussion

Currently, the QG system performs decently. In the test file provided by the course staff, the system generates three coherent and meaningful questions. When faced with a random document from the corpus of Wikipedia articles, the T5 model almost always generates a good question. However, a caveat to this performance is that we were not verifying that the generated question corresponded to the inputted answer (i.e. the quality of the question-answer pair was left unchecked). Since we had no use for the question-answer pair in our current implementation, this proved to be a benign issue. However, in the future, if we choose to use this data as a means to fine-tune our QA system, we would need to implement a verification system.

The QA system also performs well. It generally outputs the relevant section of the text corresponding to the answer of the given question. The method we created was an effective way to use the BERT pre-trained QA system to compare two candidate answers to select the best one. We could also see the method working in real time, since when the Naïve Bayes pre-ranking did not select a good first sentence, the optimal answer A^* started out as nonsense, then eventually became a valid answer in a few iterations.

A problem with our system was that it could only answer the questions using strings contained in the text. This was in general not a big problem, since it would select either a correct answer or a section of text from which the correct answer could be inferred. However, this resulted in verbose answers. In particular, when presented with simple yes/no questions, the algorithm would returning the section of text stating that the statement was true or false. In future versions of the algorithm, we could improve this add a layer of post-processing. This could include classifying the type of question using the syntax, then using that to change the answer to better match the question.

5 Conclusion

There are a few future improvements we could make to our systems.

1. **Binary questions.** Currently, our model is not able to differentiate between whether or not a question is looking for a yes/no answer. In most cases, this is simple enough to identify based on the structure of the question (clear

question words like "is" or "does" imply a binary answer). Now, our QA system returns a sentence that is technically correct but lacks the concision of a good answer. To solve this, we could implement another module which first parses the question, classifies if it is a binary question or not, and then forwards the question to the correct answering system. In the case of binary questions, we could search the text for the question's key phrase and find a sentence either agreeing with or negating the question statement. Otherwise, we would feed the question through our normal QA system.

2. **Dictating question difficulty.** Our implementation does not allow us to generate questions of varying difficulty. However, it would be advantageous to generate a variety of questions at varying difficulties to have a better QG implementation. To do so, we could train different models for different difficulties (easy, medium, hard). We would train these models on a dataset of questions, and tailor the training set to match the goal difficulties. This would allow our system to be able to output questions of differing difficulties.

3. **Improve keyword extraction.** Our implementation only extracts key phrases of a maximum ngram size 3. This is limiting because if an important phrase exceeds this size, it is skipped over. In a future implementation, we would adjust this hyperparameter or consider other methods to be able to create questions with answers that may span multiple lines, or answers that "jump" across lines in the article.

References

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python*. O'Reilly.

Ricardo Campos, Vítor Mangaravite, Arian Pasquali, Alípio Jorge, Célia Nunes, and Adam Jatowt. 2020. Yake! keyword extraction from single documents using multiple local features. volume 509, pages 257–289.

Ying-Hong Chan and Yao-Chung Fan. 2019. A recurrent BERT-based model for question generation. In *Proceedings of the 2nd Workshop on Machine Reading for Question Answering*, pages 154–162, Hong Kong, China. Association for Computational Linguistics.

Adam Roberts Katherine Lee Sharan Narang Michael Matena Yanqi Zhou Wei Li Colin Raffel, Noam Shazeer and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. volume 21, pages 1–67.

Manuel Romero. 2021. T5 (base) fine-tuned on squad for qg via ap.