Wiki QA

Mohamed Zakaria, Ali Amr, Sarah Hella

May 26, 2023

1 Introduction & Motivation

Wikipedia QA is a Wikipedia Question Answering model that serves to answers users question through the knowledge available on Wikipedia. Wikipedia contains tons and tons of information on almost all topics. However, due to the availability of this large amount of information, it can be hard to find the exact piece of information you are looking for. This is exactly the problem that Wikipedia QA aims to solve. It is designed to help searching for information easier by giving your the exact piece of information that you are looking for.

2 Challenges

With a system of this size, there are a lot of challenges to be faced. Here's a list of examples

1. Gathering information from wikipedia
2. Large number of documents to search through
3. Retrieved document may not contain the answer

Correct and sound architectural decisions will help overcome such obstacles. The plan to overcome these hurdles is as follows:

1. Use of Wikipedia API to gather information
2. Identifying if a dense method of retrieval can be used if the search space is too big with high similarity
3. Use of Top K document retrieval to extract answers from multiple docs and rank them. Highest ranking answer will be returned.

3 Dataset

For this project we will use the SQuAD2.0 dataset.

3.1 What is the SQuAD2.0 dataset?

Squad2.0 (Standford Question Answering Dataset) [RJL18] is a popular dataset used with question answering models. It is an extension of its predecessor, the SQuAD1.1 dataset [RZLL16]. Both the 1.1 and 2.0 versions are used in bench-marking of question answering models. The dataset extends on its predecessor by adding unanswerable questions to the existing answerable questions from SQuAD1.1. This was done to challenge the models performance and test its ability of not only identifying and extracting a correct answer, but also identifying when no correct answer is available and to refrain from answering. This extension serves the purpose of elevating the standards of question answering models and increase model reliability.

3.2 Data Analysis

We analyzed the SQuAD dataset in order to identify its features and structure and how the data will be processed in order to be used for training.

3.2.1 Dataset Structure

The dataset is presented in the form of a JSON file. A file is available for each of the DEV and TRAIN datasets, both of which share the same structure with some minor differences.

The dataset heirarchy starts with a set of articles. Each article is split into paragraphs where each paragraph has a context and a questions & answers object. The context represents the paragraph text which may or may not contain the answer. The questions & answers objects contains a set of questions for that particular context. Each question has:

1. Question: the question text
2. Id: question id
3. Answers: list of answers to the question from the context
    - Text: the answer text form
    - Answer Start: the index representing the start of the answer in the context
4. Is Possible: boolean representing whether this is an impossible question or no
5. Plausible Answers: Only present if the question is impossible and contains answers which may be partially correct but are considered false. It takes the same form as the Answers feature.

Its important to note that the Answer feature is one area of difference between the DEV and TRAIN sets. The TRAIN set only contains a single answer for each question in the dataset while the DEV set contains multiple answers and their respective start indices. The multiple answers represent answers selected but different annotators. Most of these multiple answers appear to have very minor differences between them.

3.2.2 Data features

We analyzed each of the available set to identify the following features about each of them:

- Dataset Size
- Number of articles
- Number of unique contexts
- Number of answerable and unanswerable questions and their respective ratios
- Vocab size
- Number of stop words

Dataset	Size	Articles	Contexts	Answerable,%	Unanswerable,%	Vocab Size	Stop Words
TRAIN	130319	442	19020	86821 , 66.6%	43498 , 33.37%	89982	623
DEV	11873	35	1204	5928 , 49.9%	5945 , 50.07%	18770	461

Table 1: Features of the SQuAD2.0 Train & Dev sets

The results of this analysis can be found in 1.

3.3 Data Processing

In order to perform the previous analysis step as well as prepare the data for training, several preprocessing steps needed to be taken.

3.3.1 Generating Data Examples

The data is originally provided in JSON format. Data must be extracted and repackaged into individual training examples in order to perform both analysis and data preprocessing for training. Each training example consists of the following:

- id : question id
- article title : name of article to which context belongs
- context : paragraph in question
- question
- answer which contains:
  - text : answer text form
  - answer start : answer start index in context

Two method variants exist for generating examples for analysis and training. The key difference is that the training examples variant adds the answer part as is. The analysis examples variant places a null value in the answers part if no answer exists. This is done to simplify parts of analysis process.

3.3.2 Data Preprocessing for Training

This is an additional step only applied to the training examples generated from the previous sub section. Getting SQuAD ready for training requires an input format goes as follows

- Input ids: appended question + context processed by the tokenizer (each word/token represented by its id)
- Attention mask: a mask the represents the words that the model should focus on using the attention mechanism
- Start position: the index of the first token in the answer in the
- End position: the index of the last token in the answer in the tokenized context

To achieve this, the question and context were both passed to the tokenizer. The output of the tokenizer includes the input ids as well as the sequence ids & offset mappings. The sequence ids are used to identify the context start and end. The offset mappings are used to identify the start and end positions of the answer in the tokenized input. Contexts which don't contain the answer partially or fully have a zero for both the start and end positions indicating that the answer is not indicated in the context.

4 System Architecture

In this section, we will discuss the model architecture used for this task. The architecture consists of 4 modules. A query processor, a Wiki Doc Fetcher & Splitter, a retriever and finally a model fine tuned for the extractive question answering task. Figure 1 shows the proposed system architecture.

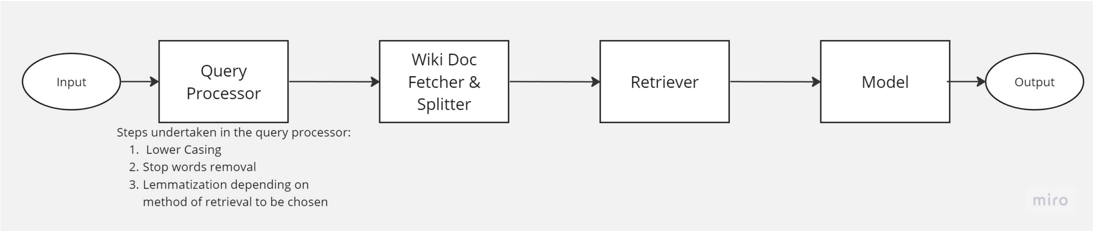
Input
Query Processor
Wiki Doc Fetcher & Splitter
Retriever
Model
Output
Steps undertaken in the query processor:
1. Lower Casing
2. Stop words removal
3. Lemmatization depending on method of retrieval to be chosen
miro

Figure 1: System Architecture

4.1 Query Processor

This is the module responsible for the processing the question or query received from the user. The query processing steps are the following, first the entire query is lower-cased. This will be followed by the removal of stop words such that only relevant words are included in the query. Finally, lemmatization of the query may be performed. This step is yet to decided as it depends on other architectural decisions yet be decided and discussed in the retriever sub section.

Removal of stop words will be done through part of speech (POS) tagging. This means that each word in the query will be tagged with its part of speech tag that indicates whether its a verb, noun, adjective or any other POS. Words that are not tagged as nouns, proper nouns, verbs, adjectives or numbers will be discarded. The result of this is filtered query only containing relevant terms.

4.2 Wiki Doc Fetcher & Splitter

This module performs 2 key important tasks for the system to function. The first task is using the processed query, the Wikipedia API will be used in order to fetch potential pages that may relate to the topic. Once the links are retrieved, each page's contents will be obtained from the API. For every page we retrieve, the module will perform a processing splitting step on the page and divide it into multiple short passages. This is done to optimize the answer extraction process as extracting from longer passages will take much longer. The preprocessing steps applied to each page's contents are

- removing empty white lines
- removing white space
- removing header and footer
- Split into passages of 200 words
- Sliding window approach used when splitting with window size 10
- Splitting respects sentence boundaries

Once each of the pages are processed into passages, the system will move on to the retrieval step. It is also worth mentioning that we have capped the number of pages that can be retrieved for a query by the API is 10.

4.3 Retriever

The retriever is a critical module for this system's success. This is because the retriever is responsible for finding the candidate documents that may contain the answer to the user's question. There are 2 types of retrievers that may be used. Sparse retrievers use metrics such as term frequency to retrieve correct documents while dense retrievers utilize embeddings to create a deep representation of documents and questions and retrieve nearest neighbors. Sparse retrievers are characterized by being effective and easy to implement, require no training as well as having the ability to work with any language. Dense retrievers are characterized by being powerful yet computationally demanding, requiring training and being language specific. We have attempted to use both types of retrievers in this system to identify which would provide us with the best results. Before we move on to discussing the retriever approach used, we first must take a look at a key component for the retrieval to work.

4.3.1 Document Stores

The document store is data structure that is used to store the processed passages and their respective representations for efficient retrieval. The notion of a document store and retriever is used by the NLP framework Haystack by Deepset [1] and it is the core behind the retrieval performed in this project. There are different types of document stores, each have their respective advantages, disadvantages and use cases. For this system, we have experimented with 2 types of document stores each for one of the retrieval methods. These types are the FAISS Document Store and the In-Memory Document Store.

[1] https://haystack.deepset.ai/

4.3.2 Dense Retriever

Dense retriever are regarded to be very powerful in the field of information retrieval. This is due to their ability to generate vector representations of documents and a query and use these representations to obtain the documents that are most similar to the provided query. One key feature of dense retrievers is their ability to find candidate documents without relying on exact word matches. This is dependant on the fact that similar words will have similar embeddings in the vector space and thus should be close together. This gives the dense retrievers an edge over sparse retrievers.

The dense retriever originally appeared to be the stronger candidate, however, this appearance did not persist for long. We used the *multi-qa-mpnet-base-dot-v1* model. This is the default embedding retriever used by haystack, however, you can use any of the pretrained models provided by Scentence Transformers [2].We picked this model specifically as it was fine tuned on retrieval tasks that use short queries to return the most similar documents which is identical to our use case (Recall that the processed query can be very short when cleaning is performed).

The main issue observed with this method was the creating of the embeddings. The embeddings creation is a very slow process that takes time. This is even slower when you consider the number of 200 word passages that you can obtain from 10 Wikipedia pages. This was a huge disadvantage for the embedding retrieval in our use as it would slow down the response time significantly (We observed the embedding creation process to take up to 5 mins!). While the embedding retriever is powerful, it is more suited for system where the documents are not frequently updated. In our case, with every question, new documents are stored.

4.3.3 Sparse Retriever

We settled on the use of a sparse method of retrieval. The retriever used is the BM25 retriever that is provided by Haystack and leverages the elastic implementation of BM25. The figure below shows the equation for BM25.

$$\sum_{i}^{n} IDF(q_i) \frac{f(q_i, D) \, * \, (k1 + 1)}{f(q_i, D) \, + \, k1 \, * \, (1 \, - \, b \, + \, b \, * \, \frac{fieldLen}{avgFieldLen})} \tag{11}$$

Figure 2: Equation for the Elastic implementation of the BM25 algorithm

where :

- q_i is the ith term in the retriever input
- IDF(q_i) is the Inverse Document Frequency of the ith term
- $\frac{fieldLen}{avgFieldLen}$ is a measure of how long the document is relative to the average document length
- b is a factor that determines the importance of the document relative length to the average (the greater this factor, the more relevant is the relative document length). The default value is 0.75 in ElasticSearch
- f(q_i,D) is the number of times the ith term appears in the document D
- k1 is a variable that controls term frequency saturation and thus the effect of a single term in the query on the document score.

BM25 is a variant of TF-IDF which improves upon the original algorithm. This is due to the fact that the BM25 algorithm takes into account a couple factors such as relative document length and term frequency saturation. It also provides us with the the ability to manipulate how important are these factors for our implementation by altering the *k1* and *b* values.

Since sparse retrievers are dependant on exact word matches rather than word similarity, a document may have a lower score than expected be because of subtle differences in word structure that essentially have the same meaning. An example of this would be words like *founded* and *founder*. While

[2] https://www.sbert.net/docs/pretrained_models.html#

these 2 words essentially have the same meaning but are used in different grammatical structures, the algorithm would not detect that these are the same word.

In order to overcome this issue we add an optional lemmatization step to all passages as well as the query used for retrieval. This helps returns words to their roots and will provide us with the ability to match the different grammatical structures of the same word. If we go back to the *founder* and *founded* example, this means that both words would be lemmatized to *find*.

4.4 Question Answering Model

The question answering model is ultimately where the user's question will be answered. Each of the candidate documents returned from the retriever will be used in this step. The model will extract an answer to the user's question from each of the documents and the answers will be ranked. The highest ranking answer will be returned to user as the final answer.

The model used to accomplish this task will be the RoBERTa Large model. RoBERTa stands for Robustly Optimized BERT pretraining Approach [LOG+19]. RoBERTa extends on BERT [DCLT19] by modifying and optimizing its pretraining protocol to become a high performing model on the SQuAD2.0 dataset. Both BERT and RoBERTa are based on the transformers neural network [VSP+17] which was a ground breaking change in the world of NLP. The RoBERTa model we obtained was provided by the Hugging Face Hub[3].

4.4.1 Tokenizer

For this project, the tokenizer was not fine-tuned by us. Instead, we use a fine-tuned tokenizer that is available on the Hugging Face Hub and provided the Deepset team [4].

The tokenizer take the following arguments:

- Question
- Context
- Max sequence length = 384
- Truncation Strategy = Only Second
- Stride = 128
- Return Offset Mappings = True
- Return Overflowing Tokens = True

You can notice that the maximum sequence length is 384 but what happens if a sequence is longer than that? This is where the stride and truncation strategy arguments come in where the truncation strategy indicates that the context should only be truncated and the question is left as is. This results in the context being split into multiple contexts all with to fit a maximum sequence length (question + context length) of 384. A sliding window approach is used when truncating with a stride of 128 to workaround having the answer being split in 2 different contexts. The truncated contexts are also returned returned due to the argument *return_overflowing_tokens*.

This results in the tokenizer output to have the following structure

- Input Ids: A tensor of input id tensors for each of the context segments
- Attention Mask: A tensor of attention mask tensors for each of the context segments
- Offset Mapping: A tensor of offset mappings for each of the context segments
- Overflow to Sample mapping: Which maps each of the segments to its original context
- Sequence Ids: A tensor containing the mapping of each token to either the question, context or no. A mapping is presented for each context segment

[3] https://huggingface.co/

[4] https://huggingface.co/deepset/roberta-large-squad2

The *overflow to sample mapping* tensor is discarded as it has no use in the case of inference. The *offset mapping* tensor is popped but is stored as it will be a key part of answer extraction. The offset mapping is simply an a tensor of tuples. Each tuple has the *start char index* and the *end char index* of its corresponding token in the context. So suppose for example that at the first index, the offset mapping contains the tuple *(1,10)*. This means that the first token in the input starts at character 1 and ends at character 10 in the original whole context.

Once the indicated tensors are popped, the tokenizer output is ready for the extraction phase.

4.4.2 Answer Extraction & Ranking

To obtain the answer to the given question, we have to go through a series of steps. The first of which is to pass the output of tokenizer to the model. This provides us with 2 tensors which are the *start_logits* and *end_logits* tensors. Each of them is tensor of tensors, where of the contained tensors contains the logits for each token of the input sequence.

The second step would be convert these logits into probabilities. Before we do that, we need to mask the tokens which were not part of the context. Therefore a mask is created which masks the parts of the input sequence that are the context. This means that the question, SEP and padding tokens are masked. The CLS token is not masked as it used to indicate that an answer is not present. After the logits are masked, they are softmaxed to convert the logits into probabilities.

Since we now have a tensor of start probabilities and end probabilities, we now want to get a single start and end probability. We do that by getting the probability for all start and end probability pairs for single context segment. This paired probability is obtained through simple multiplication. We want to get the start and end pair with the highest probability where $start\_idx \leq end\_idx$. For that we get the upper triangular matrix of the multiplication result as this ensures the established start and end indices relationship then obtain the index of the highest probability in the flattened upper triangular matrix. We convert this 1D index to its corresponding 2D index to obtain the score and the start & end indices. This is done for each of the start and end tensors for each of the context segments. This results in a start index, end index and score for each of the context segments.

The final step is get the context segment with the highest score and use its start and end indices along with the offset mapping to obtain the final answer. A final natural language output is presented to the user indicating the answer, the model's confidence score, the URL from which the answer was obtained and the specific passage as well for fact checking purposes.

5 Training & Evaluation

5.1 Training

We trained the RoBERTa Large model on the SQuAD Train dataset in order to obtain our fine-tuned model. The training was done through the Hugging Face Trainer API which allows for a smooth and simple training process. We used the following training arguments:

- num_train_epochs=2,
- per_device_train_batch_size=8,
- per_device_eval_batch_size=8,
- gradient_accumulation_steps=12,
- gradient_checkpointing=True,
- learning_rate=2e-5,
- warmup_ratio = 0.15,
- save_strategy="epoch",
- save_total_limit=1

Gradient accumulation is a technique where gradients are calculated for a small batch of data and then are saved and the process is repeated for another batch till all gradients are accumulated. This helps simulate larger batch sizes to work around not having enough computational resources. In this case using a batch size of 8 and gradient accumulation of 12 we simulate a batch size of 96.

Gradient check-pointing is another technique utilized hear in order optimize memory usage and be more computationally efficient. Gradient check-pointing strategically saves some activations during the forward pass to reduce the memory overhead. In this case, only a handful of activations from the forward pass need to be recomputed so the computational resources are optimized.

5.2 Evaluation

To evaluate the model's performance, we used the Evaluator provided by Hugging Face. The evaluator evaluates using the SQuAD2 metric which calculates the following metrics:

- Exact
- F1
- HasAns_Exact
- HasAns_F1
- NoAns_Exact
- NoAns_F1

Table 2 contains the values obtained for each of these metrics during evaluation. The metrics reflect that the model's fine tuning was successful. For reference in the original RoBERTa paper, the model achieves the following:

- Exact: 86.5
- F1: 89.4

The results of our evaluation are very close to that obtained in the paper, indicating that the training process was highly successful.

Exact	F1	HasAns_Exact	HasAns_F1	NoAns_Exact	NoAns_F1
85.78	88.89	83.47	89.70	88.09	88.09

Table 2: RoBERTa Fine-tuning Results

6 Experiments & Results

To test the system's performance, we gathered 40 general question and answer pairs. The goal was to see whether the system would be able to get the correct answers to the question set. We tested the system performance with and without lemmatization to observe the effect of lemmatization on the system's performance as well. We took a look at the task success rate and the time till completion. Task success rate is a measure of whether the system got the correct answer or not. The answer doesn't have to be an exact match for the task to be successful. The purpose of the completion time metric was to evaluate the effect of the lemmazation on inference time. Table 3 shows the results of our experiments.

System Type	Task Success Rate	Completion Time in mins
With Lemmatization	65%	42.3
Without Lemmatization	65%	39.8

Table 3: System Performance Results

As we can from the table, there is no difference in the task success rate between the both approaches. However, adding lemmatization obviously will slow down the process. Lemmatization slows down the inference process for the 40 questions by 6% which is a 0.15% slow down per question.

7 Limitations

While the system appears to have good performance, there are a couple of limitations to this system which include the following:

- Data on Wikipedia may not be the most accurate
- Not all data is available on Wikipedia
- Sparse representations of document may not be optimal for retrieval (Is there a way for dense representations to work efficiently?)
- Slow inference time
- Model is unable to infer answer from context but rather extracts it from the context (Model is not able to aggregate information from different parts of the context to provide an answer)

8 Conclusion

The purpose of this project was to create a system that can answer general questions from Wikipedia. As the results of the training, our question answering model has achieved very high quality results that are comparable to that achieved in the original RoBERTa paper. To improve this system, future work could include aggregating data from multiple sources for retrieval as well as fine-tuning the model further for better answer extraction.

References

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[RJL18] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad, 2018.

[RZLL16] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text, 2016.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.