**Policy Q&A System**

**Approach**

**1. BM25+Extractive Q&A**

We first read all the pdfs page wise and extract the text from the same. Now we create chunks of size max size 1000 characters for each pdf since it way more efficient to retrieve relevant chunks rather than the complete pdfs later down the pipeline for question answer. Chunking increases performance as well since it allows us to focus only on the relevant sections and avoids confusing the Q&A model with garbage context. For chunking, we first tokenize the document text into sentences and then iteratively create chunks from these sentences maintaining the constraint of 1000 characters. We then feed the word tokenized documents into the bm25Okapi. It does the required preprocessing (inverted index computation) of the source documents for bm25 scoring/retrieval. All this is done for the preprocessing part.

In order to answer a new query, we tokenize the query and find the bm25 score wrt each document and sort them in descending order. We generate a list of chunks for the top3 documents and again use bm25 retrieval to fetch the most relevant chunks. Top 3 chunks are retained for usage as context to answer the query. We use a pre-trained transformer model (roberta-large-squad2) to answer the query by analyzing and extracting answer from the combined relevant passages/chunks from the documents.

We’ve defined the main flask endpoint as “**/answer”**. It takes the input as jsonified “query” and uses the strategy discussed above to return the “response” and the “source” chunks retrieved. It can also handle and process inputs from the web url api. The flask app is made accessible on the port 5000 of the localhost address.

All relevant code lies in bm25\_app.py

**2. Elastic Search RAG**

Similar to previous approach, we first read all the pdfs page wise and extract the text from the same. Then we create chunks of size max size 1000 characters from each document. For chunking, we first tokenize the document text into sentences and then iteratively create chunks from these sentences maintaining the constraint of 1000 characters. In order to maintain continuity, we also include an overlap of 2 sentences between successive chunks.

Then we initialize our elastic search index with parameters such embedding dimension, meta data for chunks such as chunk\_id, doc\_id etc. We then index all the chunks using the metadata and the corresponding sentence embedding. The embedding is generated using sentence-transformer library. We use a relatively smaller model “all-MiniLM-L6-v2” for quick embedding computations. All this is done as part of preprocessing.

In order to answer a new query, we find the embedding of the query use elastic search to find top k nearest /most similar chunks from the vector database. We use cosine similarity for to compare 2 embeddings. Top 5 chunks are retained for usage as context to answer the query. We use a gpt4o-mini to answer the query. We augment the prompt with the relevant chunk data and instruct the model to answer the query based on the context provided.

We’ve defined the main flask endpoint as “**/answer”**. It takes the input as jsonified “query” and uses the strategy discussed above to return the “response” and the “source” chunks retrieved. It can also handle and process inputs from the web url api. The flask app is made accessible on the port 8080 of the localhost address.

All relevant code lies in elasticRag\_app.py

**3. BM25 RAG**

This is a hybrid approach which employs BM25 retrieval for fetch the relevant chunks similar to approach 1. Then, we augment the prompt/query with relevant context and instruct a generative model (gpt40-mini) to answer the question based on the context provided.

All relevant code lies in bm25\_generative\_app.py

**Unit Testing**

In order to test the validity of the Q&A model/code, we form basic unit tests during docker build process itself so that in case of failure, the process stops there itself. We’ve used pytest to configure the unit tests. We have 2 tests for both the Q&A models:

1. Retrieval Test: Invokes the '/retrieve' flask endpoint to retrieve top “k” relevant document chunks based on a dummy query. We’ve added assert checks to ensure that retrieval call to the endpoint is successful. Further check the number of documents retrieved as the as the “k” set for the model.
2. End to End Test: Invokes the '/answer' flask endpoint to retrieve the response for a dummy query. Here retrieval as well as Q&A happens hence being a more complete check of correctness. We’ve added assert checks to ensure that answer call to the endpoint is successful. Further check that the output is a json and has the “response” and “sources” as keys to check the correctness of the output.

Tests are defined in test.py

**Evaluation Strategy**

**Test** **Set** **Generation**

In order to evaluate the model, we need to generate synthetic question answer pairs from the pdfs. We’ve used 2 strategies for the same:

**a) Using Generative Model:** We pass the complete document text as context along with instructions to generate 10 question answer pairs from the excerpt. We do this for all documents so we can generate 100 QA pairs in total. Here we avoid chunking, because we want questions randomly sampled from different parts of the document and possibly having answers in different chunks as well. For generating QA pairs, we use GPT4-O, a superior model relative to the one we used for Q&A in RAG approach. This ensures that the test set quality is high. We instruct the model to generate QA pairs in json format for easy automated extraction. These question answer pairs are present in syntheticTest.jsonl

Now using the QA pairs generated, we also create “complex” QA pairs by randomly selecting any 2 pairs from 100 and concatenating the question and answer parts respectively to generate the new QA pair. This helps us create complex unrelated questions from possibly different documents and hence tests the retrieval as well as Q&A efficiency of the model. These question answer pairs are present in syntheticTestComplex.jsonl

**b) Masking Document Chunks:** A cheaper way to create a test without using generative models is to create chunks of document and randomly mask 50% of the words. The masked chunk + an instruction to fill in the missing words forms the query. The original chunk forms the ground truth answer. We’ve used chunks of max size 500 characters; hence total 107 QA pairs were created for this scenario. These question answer pairs are present in maskedTest.jsonl

All relevant code lies in generateQA.py

**Results**

We’ve used the following metrics to evaluate our approaches:

1. BLEU Score
2. Word Error Rate
3. Cosine Similarity with a superior model than the one used for embedding generation
4. BERT Score

All relevant code lies in evaluate.py

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Metric | Test Set | BM25+Extr. Q&A | ElasticSearch RAG | BM25 RAG |
| Bleu Score | Synthetic Test | 0.0647 | 0.30 | **0.31** |
| Synthetic Complex | 0.021 | 0.231 | **0.307** |
| Masked Set | 0.0057 | 0.382 | **0.389** |
| Word Error Rate | Synthetic Test | **0.8409** | 1.9 | 1.9 |
|  | Synthetic Complex | **0.9372** | 1.52 | 1.26 |
|  | Masked Set | 0.967 | 0.563 | **0.557** |
| Cosine Similarity | Synthetic Test | 0.47 | 0.836 | **0.838** |
|  | Synthetic Complex | 0.308 | 0.83 | **0.85** |
|  | Masked Set | 0.24 | 0.841 | **0.849** |
| BERT Score F1 | Synthetic Test | 0.85 | 0.924 | **0.925** |
|  | Synthetic Complex | 0.83 | 0.9 | **0.917** |
|  | Masked Set | 0.8 | 0.91 | **0.91** |
| BERT Score Precision | Synthetic Test | 0.88 | 0.901 | **0.903** |
|  | Synthetic Complex | 0.86 | 0.88 | **0.898** |
|  | Masked Set | 0.83 | 0.886 | **0.887** |
| BERT Score Recall | Synthetic Test | 0.83 | 0.949 | **0.949** |
|  | Synthetic Complex | 0.8 | 0.92 | **0.937** |
|  | Masked Set | 0.77 | 0.935 | **0.938** |

**Observations:**

1. The BM25 + Extractive Q&A gives poor results mainly due to the output length. It gives very short yet precise answer and hence word matching metrics like BLEU and WER show poor numbers. The cosine similarity is about 0.47 for synthetic test but drops to 0.3 for complex queries with multiple unrelated questions highlighting the models incapability to answer such questions. It also doesn’t well on masked set since it is not a exactly a question answer problem and it hasn’t such examples during it’s finetuning.
2. Elastic Search RAG gives much better results relative to 1st approach. WER > 1 is due to output length of generative model being more than reference text. It can partially be solved by decreasing max tokens. We’ve currently used a max\_token of 192 so as to cover all examples in masked test set. For masked set, the WER also drops down relative to other two sets due reference length of chunk being higher. Cosine similarity has also increased from ~0.3 to 0.8+ range portraying that answers are indeed similar to ground truth.
3. BM25 RAG gives slightly better results than elastic RAG for synthetic test queries. The improvement is higher for complex queries highlighting the model’s capability to answer multiple unrelated questions (possibly from different docs) at once. This is probably because of the simplicity and scale of the documents (10), which is helping BM25 retrieval to show better results over elastic search. It can also be because we’re using a very small model for embedding the chunks/query for quick responses. So, using a better model should help with the performance.

**Containerization**

We containerize the complete code along with pdfs using docker.

The dockerfile for bm25\_app/bm25\_generative\_app has commands to do the following:

1. Create a base python 3.9 image.
2. Set your working directory and copy all the files there.
3. Install the dependencies
4. During the build itself, start the app, trigger sleep until app starts, run the unit tests. In case the tests are passed, evaluate the model on test sets and write the results in txt.
5. CMD to run the application when docker run is invoked.

The dockerfile for elasticRag\_app has commands to do the following:

1. Create a base python 3.9 image.
2. Set your working directory and copy all the files there.
3. Install the dependencies
4. Install elastic search which is later used for document chunk indexing and retrieval.
5. Create a new user (other than root) and give the ownerships to required directory(s). This is required to run elastic search.
6. Install the dependencies
7. During the build itself, start elastic search, trigger sleep until it starts, start the app, trigger sleep until app starts, run the unit tests. In case the tests are passed, evaluate the model on test sets and write the results in txt.
8. CMD to run elastic search and the application when docker run is invoked.

**Scope for Improvement**

1. We can test different embedding models such as openai embeddings in RAG approach and choose whichever gives best results
2. We can test better generative models in RAG approach such as GPT4-o
3. For the RAG approach, we can index documents using a knowledge graph and use that for additional context.
4. We can further tune k for the top k documents, max size of chunks and other hyperparameters
5. We can enhance our unit tests to cover a more diverse range of correctness checks
6. We can do a more detailed evaluation by evaluating solely on the retrieval capability as well using ndcg etc. We should perform latency computations for both retrieval and answering for all models. We can also brainstorm better test set creation strategies such as concatenation of 3 QA pairs etc.