Semantic Search and Question-Answering Systems

A Thesis

submitted to
Indian Institute of Science Education and Research Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

by

Purva Parmar

Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

April, 2023

Supervisor: Dr. Aniruddha Pant

Certificate

This is to certify that this dissertation entitled Semantic Search and Question-Answering Systems towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by Purva Parmar at Indian Institute of Science Education and Research under the supervision of Dr. Aniruddha Pant, CEO, AlgoAnalytics, during the academic year 2022-2023.

Dr. Aniruddha Pant

Committee:

Dr. Aniruddha Pant

Dr. Leelavati Narlikar

Declaration

I hereby declare that the matter embodied in the report entitled Semantic Search and Question-Answering Systems are the results of the work carried out by me at the Department of Data Science, Indian Institute of Science Education and Research, Pune, under the supervision of Dr. Aniruddha Pant and the same has not been submitted elsewhere for any other degree.

Purva Parmar

Acknowledgments

I would like to thank IISER Pune and the Data Science Department for the opportunity to conduct this work in collaboration with AlgoAnalytics Pvt. Ltd. I express my deepest appreciation to Dr. Anindya Goswami, the Chair of Data Science MS Projects, Dr. Aniruddha Pant, my Supervisor and the CEO of AlgoAnalytics and Dr. Leelavati Narlikar, my Expert and an Associate Professor in the Data Science Department. I would also like to thank AlgoAnalytics Pvt. Ltd, where this project was carried out as part of an internship. I would also like to express my gratitude to Kalash Gandhi, Piyush Sar and Abhishek Pawar, Data Scientists and my mentors at AlgoAnalytics. I would also like to thank Anoushka Rao, a Cloud Engineer at AlgoAnalytics, for always being there to guide me on the realm of Cloud Services and the actual deployment of my work. I'd like to thank Goirik Chakrabarty, my friend, for all the discussions across various fields, and also for proofreading this work. Finally, I'd like to thank my parents and friends who have always been there as a support.

Abstract

Keyword Search Systems have been in wide use since the availability of computers themselves. Keyword search returns the relevant results which contain the exact keywords used in a search query. But in today's world of extensive information, a simple keyword search is bound to miss a lot of other relevant results which are phrased or written differently. This leads us to develop Semantic Search systems, which can also consider the context and meaning behind a search query, and return results which may not have the exact keywords, but still are relevant because of the intended meaning. This can be taken one-step further by introducing (Extractive) Question-Answering Systems, which if given a question-like query, can directly extract the short answer from the massive wealth of existing information. We explore the development and implementation of such a Semantic Search and Question-Answering system in the domain of finance for a financial product, AlgoFabric, at AlgoAnalytics Pvt. Ltd.

Contents

Introduction

The advent of computers and the internet has revolutionised the way we access information. Keyword search systems have been widely used to retrieve relevant information based on the exact keywords used in a search query. However, the increasing volume of information available today makes it challenging for simple keyword search systems to retrieve all the relevant information. Systems like BM25 and TF-IDF work fast but ultimately lose out on possibly relevant results which do not have the exact keywords. This limitation has led to the development of semantic search systems that can account for the context and meaning behind a search query and return relevant results that may not contain the exact keywords used in the query.

To make Semantic Search systems a reality, we need to first find a way to represent words as vectors in a high dimensional space, so that they can be compared with each other and operations can be performed computationally. This leads to development of the idea of Word Embeddings and the many methods associated with it, like word2vec[1, 2], GloVE[3], BERT[4], etc. The Semantic Search itself is then accomplished by the usage of Dense Passage Retrievers[5].

We look at the relevant background, starting with Keyword and Semantic Search Systems in Chapter 1, followed by Word Embeddings and different kinds of deep learning models. More information about them can be found in the book *Speech Language Processing*, 3rd Edition Draft, by Dan Jurafsky and James Martin[6].

Semantic search systems can be taken a step further by introducing question-answering systems that can directly extract short answers from a vast pool of information. Such systems can significantly benefit various domains, including finance, where quick and accurate retrieval of information is crucial. This is then implemented by the usage of Reader Mod-

els[7].

The purpose of this thesis project is to explore the development and implementation of a semantic search and question-answering system in the domain of finance at AlgoAnalytics Pvt. Ltd. The system will be designed to retrieve relevant information based on the context and meaning behind a search query rather than just the exact keywords used. The system will also be able to answer question-like queries by extracting short answers from a vast pool of information. We look at a sample architecture of the implementation in Chapter 2.

In Chapter 3, for evaluation, we look at existing benchmarks, and also the creation of a custom dataset that is used for comparing different models in the context of the domain of the work. Finally, we look at the results and conclude with the chosen models for the application.

Chapter 1

Background

Searching for relevant information has always been a need in systems with a lot of data. Manually sorting and looking for important information is a thing of the past now. With the advent of the internet and computers, searching has become more relevant than ever, with troves of online databases, websites, blogs, articles, and so much more. Google, the Search Engine, became popular due to its efficient PageRank algorithm.

A *document* shall refer to any unit piece of information. For example, a document may be a website, a news article, a blog piece, a tweet, a PDF file from some organisation, etc.

It is not necessary for a document to have a fixed or specified length or size in terms of text characters or words. It may be of any size. Documents can be processed to be split into smaller chunks if required.

A *corpus* is a collection of all the documents that we have, in the context of our domain or system.

Corpus can be loosely thought of as the Universal set. For example, in the context of a company's internal product, the corpus can refer to all the pages of documentation and code-base written for the product. For someone searching for something relating to the product, only this corpus matters.

The problem statement for searching is simple: Given a set of query keywords, find the relevant documents which contain these keywords and rank them in the order from most

relevant to the least relevant. As such, ranking the documents becomes an important part of the search system. This is where ranking algorithms come in.

This field is also referred to as *Information Retrieval*, as we are retrieving the relevant information from the vast amount of data that may exist in the system.

1.1 Keyword Search and Ranking Functions

Keyword search refers to the procedure of finding documents containing the *exact* keywords mentioned in the query.

One way to accomplish this might be to look for how many times each keyword occurs in each document, and rank the documents in order of which contains the most amount of keywords the most times. This can easily lead to problems where larger documents have more words, and can easily be ranked higher up. There might also be keywords in the query which are articles or conjunctions like *the*, *and*, *a*, etc., which naturally occur in sentences and hence in most documents. So, simply ranking documents by frequency is not going to work.

1.1.1 TF-IDF

TF-IDF stands for Term Frequency - Inverse Document Frequency. The tf-idf value of a document increases in proportion to how often a keyword occurs in a document, but also takes into account how many documents contain the keyword. This takes care of the problem mentioned earlier about how some words occur more frequently, like articles, conjunctions, etc.

The tf-idf value of a document contains two terms, as the name suggests: the term frequency (tf) and the inverse document frequency (idf). This first term is due to Luhn, 1957[8].

Definition 1.1.1. *The number of times a keyword occurs in a document is called its term frequency.*

However, larger documents simply may have a higher term frequency owing to their size. We need to weight the term frequency appropriately. One simple way is to normalise by the total number of words present in the document.

With this, the term frequency $\mathrm{tf}(t, d)$ of a term t within a document d becomes:

$$\mathrm{tf}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \tag{1.1}$$

where $f_{t,d}$ is the raw count or frequency of the word t in document d.

Another way to handle the size disparity among documents is to take a logarithm.[6]

$$\mathrm{tf}(t, d) = \log(f_{t,d} + 1) \tag{1.2}$$

Where 1 is added to avoid singularities with $f_{t,d} = 0$. One way to think about this is that a term appearing 1000 more times in a document does not make it 1000 times more relevant semantically than other documents. Taking a logarithm compresses the range of values.

The second term, the inverse document frequency is due to Sparck Jones, 1972[9]. It is a measure of how important and rare a keyword is. If a keyword only occurs in a few documents, we give more weightage to ranking by this keyword as compared to other keywords which occur more frequently everywhere. For example, a proper noun like IISER Pune is rarer than a word like *research*, which may occur more generally in the corpus.

Definition 1.1.2. *Let D be the corpus containing N documents. The inverse document frequency (idf) of a term t in a corpus D is the logarithmic fraction of the total number of documents N and the number of documents in which the term t occurs, say df_t.*

$$\mathrm{idf}(t, D) = \log\left(\frac{N}{\mathrm{df}_t}\right) = \log\left(\frac{N}{|\{d \in D : t \in d\}|}\right) \tag{1.3}$$

.

Definition 1.1.3. *The tf-idf score of a term/keyword t in a document d within a corpus D is a product of the two statistics, the term frequency (tf) and the inverse document frequency*

(idf).

$$\text{tf-idf}(t, d, D) = \text{tf}(t, d) \ \cdot \ \text{idf}(t, D) \tag{1.4}$$

So, keywords which are rarer and frequently occur in only a few documents result in a higher tf-idf score. Common words are filtered out because of low idf scores.

1.1.2 BM25

The BM stands for Best Matching. This ranking function is also referred to as Okapi BM25, in reference to where it was first implemented in practice (the Okapi Information Retrieval System). It was conceived by Jones et al. around 2000[10].

Introduction to Information Retrieval by Manning et al.[11] presents the BM25 function along with some history.

Suppose the query Q contains the keywords $q_1, q_2, \ldots, q_n$. Then, the BM25 score of a document D is:

$$\text{BM25-Score}(D, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{(k_1 + 1) f(q_i, D)}{k_1 \cdot \left(1 - b + b \cdot \frac{L_D}{L_{avg}}\right) + f(q_i, D)} \tag{1.5}$$

where

- IDF is the Inverse Document Frequency
- $f(q_i, D)$ is the count of the keyword q_i in document D
- L_D is the length of document D in words
- L_{avg} is the average document length of the corpus
- k_1 and b are chosen parameters, usually set to $k_1 \in [1.2, 2.0]$ and $b = 0.75$

k_1 is a parameter which scales the term frequency $f(q_i, D)$. A larger value of k_1 leads to more focus on the raw counts of the keyword in a document. $k_1 = 0$ corresponds to a binary model, i.e. the document D contains the keyword q_i or not.

$b(0 \leqslant b \leqslant 1)$ determines scaling by the document length. $b = 0$ is no length normalisation, while $b = 1$ represents fully scaling the term weight by the document length.

1.2 Semantic Search

Search systems described so far have only considered the literal keywords, and searched and ranked documents based on the presence of the keywords. However, natural languages are more complex, and simply looking at keywords and not their surrounding context and intent can result in unintended and not-so-relevant output.

For example, consider the search query: "How tall is the tallest mountain on Mars?" A keyword search would focus on the relevant keywords of *tallest*, *mountain* and *Mars*, but will likely return documents containing the tallest mountains on Earth, reading which would then answer the question as it being Mt. Everest. However, the context of the mountain being on Mars is still understood. While there may be a document or two which present the actual statistics of mountains on Mars, it would be among the vast quantity of others detailing the mountains on Earth.

For Semantic Search to work, however, we would need a way to vectorise words so as to compare them, and also somehow consider their word order so that some semblance of context and intent is achieved. This is where we get to Word Embeddings and Transformer models.

1.3 Word Embeddings

A *word embedding* refers to the vectorised representation of a word in a natural language. We want to assign vectors to words in such a way that many features of a natural human language are preserved. Speech and Language Processing by Jurafsky et al.[6] presents many

of these, termed as *Lexical Semantics*:

- Words may have different meanings in different contexts. For example, "orange" may refer to a colour, or may also refer to a fruit.
- Different words can have similar meanings, also known as *synonyms*. For example, *lift/elevator*, *fridge/refrigerator*, etc.
- Different words can also have opposite meanings, also known as *antonyms*. For example, *good/bad*, *import/export*, etc. We would like our vector representations of words to also reflect these synonyms and antonyms in some way.
- The *principle of contrast*[12] states that *whenever there is a difference in the word form in a language, there is also a different in the associated meaning*. To give an example, consider the word pair $H_2O/Water$. One is used more in a scientific context, and the other more in a general context. However, both of them refer to the same thing. These can also be considered as synonyms, but the point to take away from here is that no two words have the exact same, identical meanings.
- Words may not be synonyms/antonyms, but they can still share some kind of *similarity*. For example, Asia and Europe are two continents and are similar in that way. One way of quantifying word similarity is by having humans judge their similarity and score them. One such dataset is the SimLex-999 by Hill et al.[13] There the similarity between pairs of words is quantified in a scale range of 0-10.
- Words can also be *related*. For example, consider the words *student* and *books*. They are not synonyms, nor are they similar. However, they are *related* to each other. Students use books in their studies. Words can be grouped into what we refer to as *Semantic Fields* based on their relatedness. For example, the semantic field of an Tech Company may include a group of words like (departments, computers, keyboards, monitors, cubicles, cafeteria, furniture) etc.
- The perspective and role can matter in a sentence. Consider the activity of me lending my friend Goirik some money for lunch when he forgot to bring his wallet and his digital transactions are not working due to the bank server being down. This can be paraphrased in two ways as "Purva lent Goirik some money for lunch", and as "Goirik borrowed some money from Purva for lunch". Understanding these perspectives and

shifting from one to another becomes essential in multi-lingual cases, where natural languages differ in sentence construction.

These are some of the features of natural languages, and it would be nice if these features are preserved when we do eventually consider assigning vector representations to words.

1.3.1 Vectorizing words

Word Embeddings are vector representations of words in a high-dimensional space. There are various methods to generate such embeddings.

Word2vec

The word2vec model presented by Mikolov et al.[1, 2] is trained on a large corpus of data and generates a high-dimensional vector representation for each word in the vocabulary. There are two variants of the word2vec model:

- Continuous Bag-of-Words (CBOW): This model takes a context of surrounding words in a chosen window-size and tries to predict the target word in the middle of the context.
- Skip-grams: This model takes a single word and tries to predict the surrounding context words in a chosen window-size.

GloVe

GloVe (Global Vectors for Word Representation)[3] is a word embedding technique built on the idea that we can derive semantic relationships between words from their co-occurrences (i.e. how often two words occur together).

It builds a co-occurrence matrix which tabulates how often pairs of words appear together in a corpus of data (GloVe uses Wikipedia), and then uses it to train a neural network

to generate word embeddings. It also balances the importance of rare and common co-occurrences.

1.4 Transformers

Transformers are a type of deep learning model that were introduced by Vaswani et al. in 2017 in a paper titled "Attention Is All You Need"[14], where they eliminate the need for Recurrent Neural Networks (RNNs) and instead use a self-attention mechanism to capture dependencies between input and output tokens.

Self-attention is a mechanism that allows the model to focus on different parts of the input sequence while processing each element of the sequence. In each self-attention layer, the model calculates attention weights for each element in the input sequence based on its relationship with all other elements in the sequence. This allows the model to capture long-range dependencies in the input sequence without requiring recurrent connections.

There are two parts to the Transformer architecture, the encoder and the decoder. In Figure 1.1, the left side is the encoder, and the right side is the decoder. The encoder takes the input sequence and produces a sequence of hidden representations which are passed to the decoder. The decoder generates the output sequence based on the encoder's hidden representations and the previous output tokens. There are also multiple attention heads, which allow the model to focus on different parts of the sequences as needed.

One of the key advantages of the Transformer model is its ability to parallelise computation, which speeds up training and inference. This is due to the fact that the self-attention mechanism allows the model to process each element in the input sequence independently.

1.5 BERT

BERT (Bidirectional Encoder Representations from Transformers)[4] is a transformer-based model that is trained on a large corpus of text data to learn the context and meaning of words. It is bi-directional, meaning it can process text in both directions, which allows it to

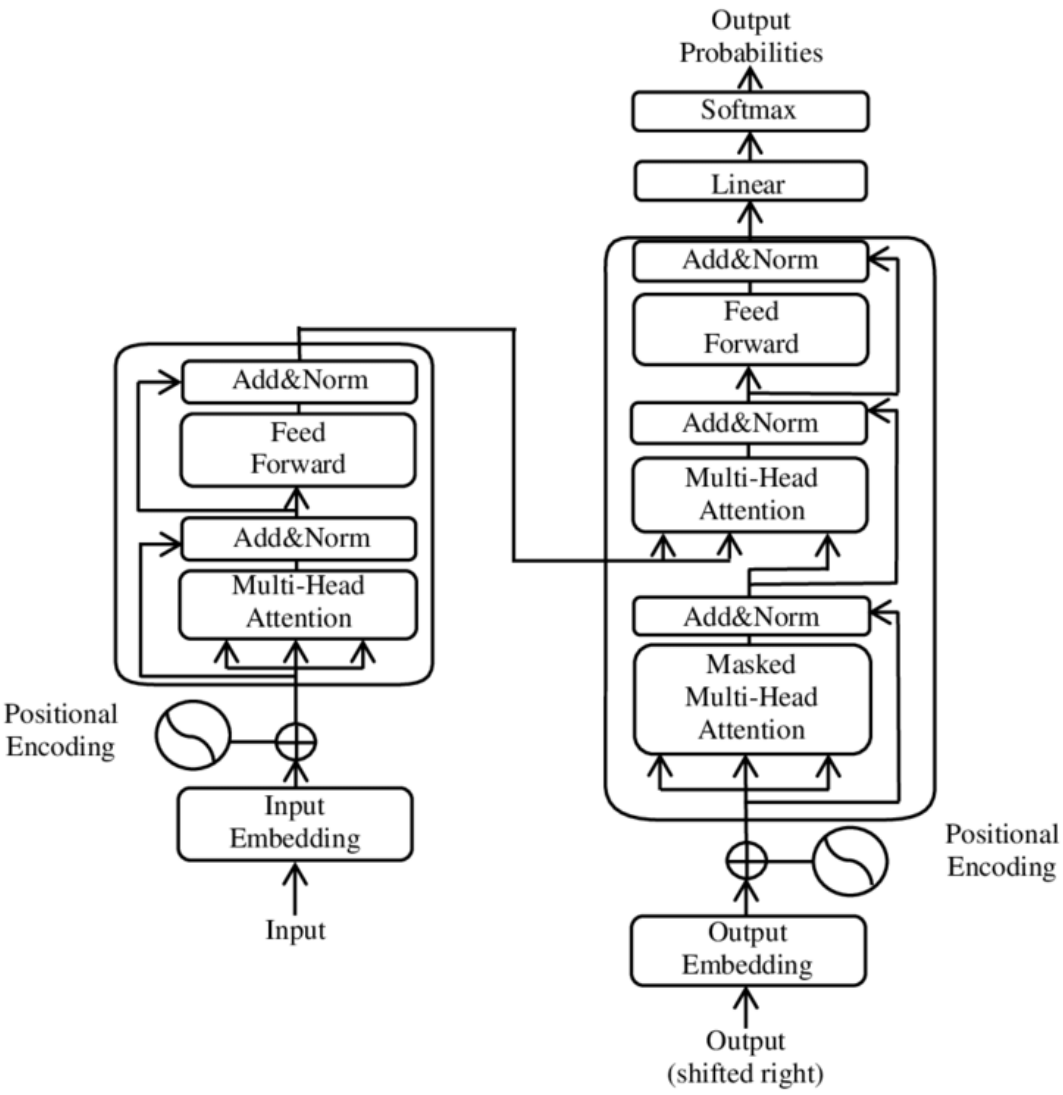

Figure 1.1: The Transformer Architecture[15]

learn the context and meaning of words more accurately than traditional language models.

Language Models are probability distributions over sequence of words[6]. Given a sequence of words, they can assign a probability to each word. These models can also hence be used to generate the next word given the previous context based on probabilities.

BERT uses the encoder layers of a Transformer, as it only has to generate a language model. The encoder reads the entire sequence of words at once. BERT also uses Masked Language Modelling (MLM). There are special tokens introduced: [CLS] for classification tasks, [SEP] for separating tokens, [MASK] for masking (hiding) certain words. 15% of the tokens are masked, and the training of BERT involves predicting these words.

After this pre-training, BERT can be fine-tuned to specific tasks.

BERT can be used to generate word embeddings. Traditional embedding methods such as word2vec and GloVe generate static embeddings that do not change based on the context of the input text. In contrast, BERT generates dynamic embeddings that can adapt to the context of the input text, allowing it to capture more nuanced and complex meanings of words.

Another advantage of BERT word embeddings is their ability to handle out-of-vocabulary (OOV) words. In traditional embedding methods, OOV words are either mapped to a special token or ignored entirely, which can lead to information loss. BERT can generate embeddings for OOV words by leveraging the subword information learned during its MLM training. This process allows BERT to handle rare and complex words that may not be present in the training corpus.

1.6 Dense Passage Retrievers

Dense Passage Retrievers (DPRs)[5] are neural network models for retrieving relevant passages from large text data. DPR uses dense vector representations to represent the context of a given query and passages in a dense vector space, allowing for efficient similarity-based retrieval.

The DPR is composed of two main components: a bi-encoder and a retriever. The

bi-encoder is responsible for encoding the query and the passage into dense vectors, while the retriever retrieves relevant passages by calculating the similarity between the query and the encoded passages in the dense vector space. The bi-encoder is typically a large pre-trained language model, such as BERT[4], while the retriever is usually implemented using approximate nearest neighbour search algorithms, such as FAISS[16, 17].

One of the main advantages of DPR is its ability to handle long and complex queries, which is a challenging task for traditional information retrieval systems. Furthermore, DPR is able to capture the semantic relationships between words and phrases, allowing for more accurate retrieval of relevant passages.

1.7 Question-Answering

The goal of Question-Answering (QnA or QA) is to generate or extract an answer to a query question. These systems can be Open Domain (where the entire knowledge base, say Wikipedia, is available), or Closed Domain (where the system is restricted to a domain-specific corpus of data).

Document Reader by Chen et al.[7] uses a Reader model for QnA tasks. These are neural networks, where the word embeddings are passed in as input. The output layer generates a probability distribution over the possible answers to the query based on the passages identified by the model.

The Stanford Question Answering Dataset (SQuAD) v2.0[18] is a QnA evaluation dataset consisting of 150,000 questions over a set of 500+ Wikipedia articles. This has become one of the benchmarks for evaluating QnA models.

Chapter 2

The Application

This project's work involved with developing a Search application for the AlgoFabric project at AlgoAnalytics Pvt. Ltd., which would essentially be the backend behind the Search button on the product. While it started with a standard keyword search, it went on to be developed into a Semantic Search system and extended with a Question-Answering component.

2.1 Architecture

2.1.1 Considerations and Requirements

There are several requirements of a search system:

- It returns results which actually contain or relate to the keywords.
- The results must be ranked in a meaningful order, from *most relevant* to *least relevant*.
- The system should be a Semantic Search type, so that it can find meaningful results even when the exact keywords may not be present in the results.
- The search should not take too long. The end-user will simply not wait for the search to finish and move on to other things. Speed and user experience are of priority.

- The documents are primarily of two types: News and Tweets. Results for them must be presented separately, where applicable.
- The system should tie in well with the other tech stack being used throughout AlgoFabric.
- There must be appropriate documentation so that application maintenance responsibilities can be passed on easily, and that there is a reference available.
- There should exist some evaluation metric so that various semantic search models can be compared against each other.

2.1.2 Tech Stack

The tech stack was chosen as follows:

- Python as the primary programming language, as it is the most widespread in the community for developing AI Applications
- Haystack[19], an open-source NLP framework in Python for developing systems which use Transformers and other deep learning models
- FastAPI[20], a web framework for building APIs in Python
- Git, a Version Control System for managing different versions of the source code
- GitHub, a cloud-based repository for hosting source-code which is version controlled with git
- Hugging Face Model Hub, a model repository for pre-trained models for various tasks in NLP
- Docker, a platform as well as a software tool for containerising applications, isolating them in their special environment, called containers, so they do not clash with any other applications' dependencies
- A Cloud Platform (Amazon Web Services / Microsoft Azure) for deploying the containerised application

2.1.3 Planning Phase

The documents to be processed within the application are primarily of two types: News and Tweets. Results for these must be presented separately, wherever applicable.

These documents reside in a central database which is hosted on a Cloud Platform (Amazon Web Services / Microsoft Azure). There also exists a lot of metadata (creation date, title, source, etc.) and some information from other tools in the AlgoFabric ecosystem, like sentiment analysis scores for the document.

Haystack, an open-source NLP framework, was chosen as the framework to develop with. Developing a Search system in Haystack requires three components: Indexing Pipeline, Document Store and Search Pipeline as shown in Figure 2.1.

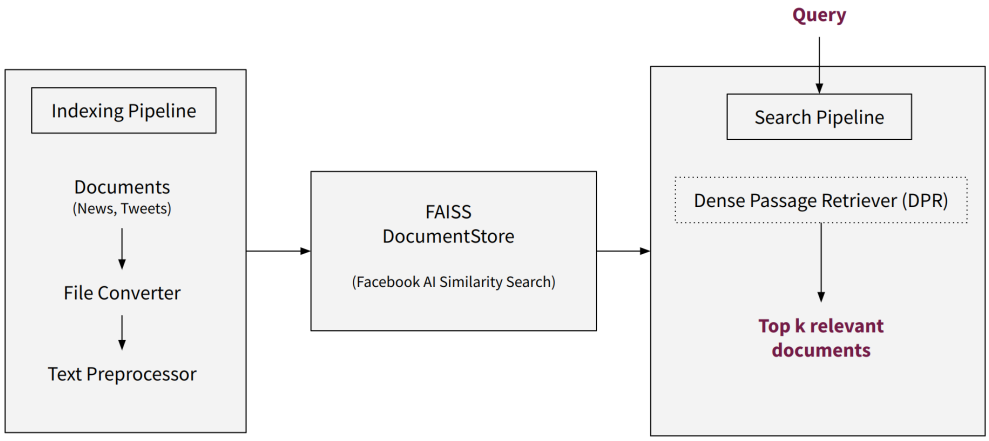

Figure 2.1: A basic Semantic Search pipeline in Haystack.

The Indexing Pipeline has two parts:

- The File Convertors to first bring all documents into a common format. This is usually a plain text file.
- The PreProcessor cleans whitespaces, removes header/footers and splits the documents into fixed-size chunks. It can be configured to respect sentence boundary so that a document is not split mid-sentence.

The Document Store contains all the documents which have been processed appropriately

through the Indexing Pipeline, and also their metadata. The Document Store is usually an sqlite database which is stored as a file on the system. If the Search Pipeline uses a Retriever which needs Word Embeddings, then they can be stored in the Document Store in an index file alongside the sqlite database file.

In the Search Pipeline, we take in the query, and search for relevant documents in the document store. Word Embeddings that might be needed are stored in the Document Store as well. The top k documents are returned as the search results, where k is set as per use case, usually 5 or 10.

The Application Development was also planned with the idea in mind that the models should be plug-and-play. The application logic should not depend on what model is used, and simply providing the path of any downloaded model should work out-of-the-box.

2.1.4 The FAISS Document Store

Document Stores that work with Dense Passage Retrievers (DPRs) must store the Word Embeddings for the documents so that these vectors can be compared and searched for.

Facebook AI Similarity Search (FAISS)[16, 17] is a library for efficient similarity search of vectors.

Within the Haystack framework, we use the sqlite database for storing the documents, while storing the embeddings in a FAISS Index file. Haystack provides functions to achieve the task of calculating and updating embeddings for documents based on the Retriever used.

2.1.5 Implementation

The exact implementation details and the source code is the property of AlgoAnalytics Pvt. Ltd. and hence cannot be shared, however, the general structure is elaborated in this section.

The application can be thought of as having a few separate programs which connect together in some way.

Creating Resources

We first create any resources that the application may need for a smooth operation.

- Creating necessary directories
- Empty Document Store and FAISS Index files
- Downloading the chosen models from Hugging Face and storing them locally, like the Dense Passage Retriever and Reader

This program would only need to be run once when starting a fresh instance.

Fetching Data from Central Database

The actual news and tweets reside in the central database, which itself is on a Cloud Platform. The entries have a database ID which looks like *Title+URL*, where the URL is the original source link from where the news/tweet content was obtained from. For writing documents to Haystack's Document Store, we need to first have the documents present in plaintext files in a folder. Since we need to know the original ID, that information has to be stored as well. However, since this ID also contains special characters, we cannot simply name the plaintext files after this ID.

Here, we introduce a mapping dictionary and a mapping file. Whenever we fetch a document from the central AlgoFabric database on the cloud, we assign a randomly generated 128-bit label, called a Universally Unique IDentifier (UUID), in accordance with RFC4122[21]. An example of a UUID is `bb055e14-8bde-4703-a346-13ca0488f0b3`. So, we take this UUID, and name the plaintext file as the "GeneratedUUID.txt", and keep track of the corresponding actual Database ID in a dictionary structure, which we call the mapping dictionary. We store this mapping dictionary in a JSON[22] file, which we call the Mapping File.

Using the Mapping File, we can also track which database entries have also been already written to the Document Store, so future fetches will be able to remove any duplicates.

So, in the fetch program, we

- Load the Mapping File, if it exists.
- Fetch news and tweets from the central database.
- Check if any of these documents have already been written, using the IDs present in Mapping File. Remove already written documents. Now, only new documents are left.
- Assign each document a randomly generated UUID, and store the contents in plaintext files whose names are these UUIDs.
- Keep track of all the UUID → Database ID mappings in a mapping dictionary.
- Invoke the Haystack's Indexing Pipeline on the folder where all the plaintext files are saved and write them to the Document Store.
- Write the mapping dictionary onto a mapping file in JSON format.
- Compute and writes the embeddings in the FAISS Index. This step takes the longest time, and can be significantly sped up with GPUs.

This program must be run on a schedule, every workday, every few hours so that the search application always has access to the latest data, up to a certain time lag.

Databases hosted on Cloud Platforms also usually have rate limits (how often and how much data that can be fetched in a given time range), or may need other optimisations.

On Amazon Web Services (AWS), the database product is DynamoDB. Querying the database has a restriction. A single fetch query can only return data up to a size limit of 1 MB. If we are to fetch data which is going to be larger than that, we need to split our queries so that we *paginate* them[23].

AWS DynamoDB Docs[23] present the following logic to paginate the queries:

1. Perform a query. Check if the result contains a `LastEvaluatedKey` present. If it is, then there are additional items to be fetched as the limit was reached. If it is not, then the query returned all the results that were there and we can exit.
2. Perform a new query with the same conditions as the first one, but this time, also include an `ExclusiveStartKey` parameter containing the `LastEvaluatedKey` value from the last query.

3. Run the new query request.

4. Go to step 1.

This whole process is similar to an Iterator object in multiple programming languages which yields its entries until none is left.

On Microsoft Azure, pagination is not required, but requests are optimised by dividing the queries based on a partition key. Azure's Database Product: CosmosDB, divides the database entries into distinct subsets called *logical partitions*[24]. These logical partitions are created based on a value called *partition key*. Entries that have the same partition key are stored in the same partition. In the AlgoFabric database, the partition key is set to *month*, meaning news/tweets which were created in a given month reside in the same partition.

If we query CosmosDB without specifying a partition key, for example, by specifying a date range spanning over multiple months, it will have to look into each of the logical partitions and fetch data matching the query condition. However, we can split our single query over multiple queries so that each one fetches data only for a single month, and hence specifies a partition key (month) value for each query. This results in faster response times from CosmosDB and the overall data is obtained in a more efficient manner.

Processing Search Results

The Semantic Search component takes in a query, uses the Dense Passage Retriever's Query Embedding Model to compute the vector embeddings for the query, and then the most similar embeddings of documents are found with FAISS. The results contain top k documents, ordered in the rank of most relevant first. In the results, each one has a metadata consisting of the original filename where the document was written from, which is of the form of "UUID.txt". We take this UUID, and use the Mapping File to find the corresponding Database ID. We then query the AlgoFabric database for the whole information on this document.

We query the central database again because it contains some of the other semantic information, like Events Analysis, Named Entity Recognition and Sentiment Analysis as processed by other tasks running in AlgoFabric. We then process the entire result so that it

can be displayed properly on the front-end User Interface, and return it in a JSON format, a standard for APIs.

Handling Older Documents

The search results must not be too old. One way to implement this is by introducing some weights to each document which decay with time, so that their importance reduces as they become older.

However, a much simpler method which tends to work in practice is simply removing documents older than a threshold, say, 30 days. This is what we implement here. This program is also run on a schedule, once every day.

App

The actual application implements an API using FastAPI. This runs a Server which serves the `/search/{query-string}` endpoint as an API. The front-end of AlgoFabric can then interact with this endpoint and get the necessary results.

All the models and the document store are loaded, and the results from the retriever are processed as per the needs of the front-end and returned in a JSON format, standard for an API.

We run the search pipeline twice, once for news and once for tweets. There are also two Document Stores, one each for news and tweets. The Semantic Search part is handled by a Dense Passage Retriever, which has two models: a query encoder model and a passage encoder model, which encode queries and passages (documents) to their vector embeddings. The search and ranking of the results is handled by FAISS, which implements some approximate nearest neighbour algorithms.

The whole application is containerised with Docker and the web application serving the `/search/` API is then deployed to a Cloud Service.

2.2 Extending with QnA

If the query is a question, we can go one step further and introduce a Question-Answering (QnA) component. QnA tries to find a short answer to the specific question.

We introduce a Question vs Statement Classifier, which is a simple neural network trained for the task, in the architecture. If the query is a question, the question along with the search results from the Dense Passage Retriever are passed onto a Reader model.

The Reader model reads through the search results and returns answers to the question along with a confidence score.

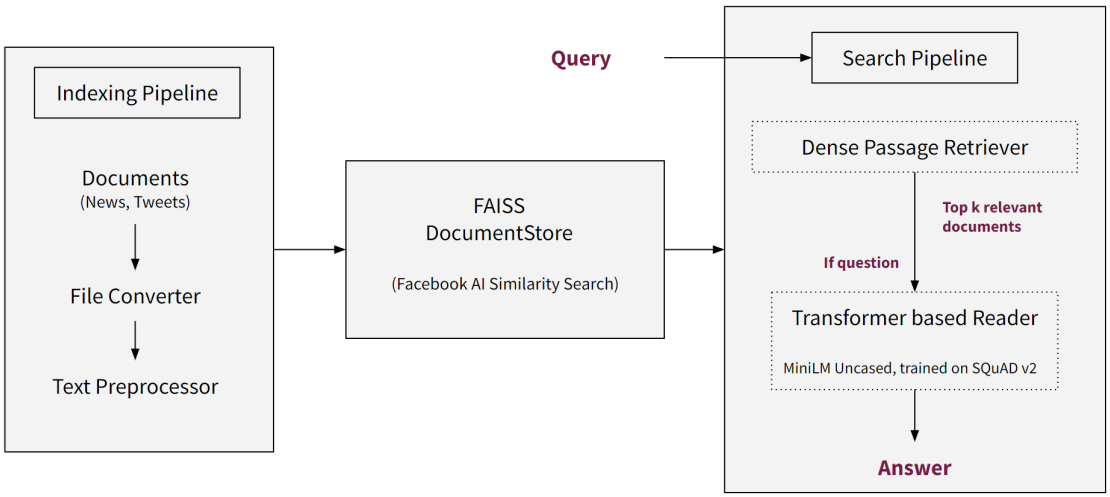

Figure 2.2: Semantic Search + QnA Architecture

A QnA system can be of two types:

- Extractive QnA: Tries to find a specific start and end point for a short answer by reading through the top k documents returned from the Search Pipeline / Retriever.
- Generative QnA: Tries to generate a short answer word by word.

Generative QnA tends to work better with larger data and and larger and better trained models. However, when the scale is not too big, as is the case with AlgoFabric, Extractive QnA works just fine, and that is what we use. This is accomplished by using a *Reader* model.

Since we want the QnA to only be executed when the query is a question, we also introduce a small Query Classifier model, which is again a Transformer which classifies a query between a Statement or a Question. If the query is a question, we combine the top k results from news and tweets search pipelines and pass them to the Reader model, which reads through them and returns the top few answers. We combine the news and tweets sources because when looking for an answer, it might be present anywhere and now the distinction of the source being a tweet or a news does not matter anymore.

2.3 Deployment

The whole application is containerised using Docker[25]. This involves isolating the project's runtime and dependencies into a single container so that it does not clash with anything else, and can be reliably reproduced the same way in every machine which can run Docker containers.

The first step is building a Docker image, which is like a blueprint for a container. The image can then be used to create containers which can be run to use the application.

The isolation also means that each container has its own data, and the data will not persist between multiple container runs. Every time the container is stopped, it loses any new data. To solve this issue, we use Docker Volumes. These are external directories which can be mounted onto a container, so that it has access to persistent data that will remain even after restarts. This also means that multiple containers which are built for running separate programs can access the same data using Docker Volumes.

Docker Images can then be uploaded (better known as *pushed*) onto a Container Registry, like Azure Container Registry (ACR) or Amazon Elastic Container Registry (AWS ECR). After that, we can create a container from the image and run it using any application service on the Cloud Platforms. This completes the deployment procedure and the API is served from the cloud.

Chapter 3

Evaluating Models

Haystack provides some internal functions and tools for evaluating, and metrics and their information are detailed on the Evaluation documentation page[26].

3.1 Retrieval Metrics

We look at some of the metrics for evaluating Retrievers, and hence essentially the Semantic Search part of the whole system.

3.1.1 Recall

Recall is the fraction of relevant/correct documents that were present in the search result.

For a single query and a single correct document, the search result either has the relevant document or not, meaning the evaluation can be done in binary terms. For the entire evaluation dataset, this would a score between 0 and 1. If no query result contained the correct document, then the score would be 0. If all query results contained the correct document, then the score would be 1.

There can be multiple correct documents which are relevant to the search. There are two metrics in Haystack for this:

- `recall_single_hit` is about whether at least one of the correct documents was returned.

- `recall_multi_hit` is about how many of the multiple correct documents were returned in the result.

These metrics are affected by how many documents the retriever actually returns, which is a configurable parameter: top k documents. The value of k is set depending on the use case, usually 10 in AlgoFabric.

3.1.2 Mean Reciprocal Rank (MRR)

The Mean Reciprocal Rank (MRR) score takes into account the position of the first correctly returned document (the "rank") in the search result. This helps to also consider how *relevant* the results further up top are.

For a set of Q queries, the MRR rank is computed as:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i} \tag{3.1}$$

where rank_i is the position, or rank, of the first relevant document in the i^{th} query.

The MRR can be a value between 0 (no correct matches) and 1 (a correct document was returned as the top result in all queries).

3.1.3 Mean Average Precision (mAP)

MRR only considers the position of the first correctly retrieved document in the search result. Mean Average Precision (mAP) takes into account the position of every correctly returned document in the search result.

3.2 QnA Metrics

Extractive QnA works by returning the start and end point within a document, which may be the answer to the question asked.

3.2.1 Exact Match

Exact Match is the proportion of queries where the predicted answer matches exactly with the expected answer.

For a single query, the exact match score is either 0 or 1, i.e. the answers do not match exactly or they do. For a set of queries, this simply becomes the number of exact matches.

This is not a very good measure for Semantic Search, as the predicted answer by the Reader model may include or exclude smaller words like articles (a, an, the), and the answer may occur phrased differently in multiple areas in a document.

3.2.2 F1 Score

F1 score measures the word overlap between the predicted answer and the expected answer.

In the case of evaluating QnA, Precision would be the fraction of words in the answer that are correct. Recall would be the fraction of correct words in the answer as compared to the expected answer.

Precision is the fraction of retrieved documents that are correct. Recall is the fraction of relevant documents that were successfully retrieved.

$$\text{Precision} = \frac{\text{Number of correct words in the predicted answer}}{\text{Total number of words in the predicted answer}} \tag{3.2}$$

$$\text{Recall} = \frac{\text{Number of correct words in the predicted answer}}{\text{Total number of words in the expected answer}} \tag{3.3}$$

F_1 score is the harmonic mean of Precision and Recall.

$$F_1 = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} \tag{3.4}$$

The F1 score is more lenient than the Exact Match, and hence a better indicator. This is also a score between 0 and 1. A score of 0 means an absolutely incorrect answer that does not have anything in common with the expected answer. A score of 1 means an Exact Match with the expected answer.

3.2.3 Semantic Answer Similarity (SAS)

The Semantic Answer Similarity (SAS) metric uses a Transformer model to evaluate how semantically (meaning and intent-wise) similar the predicted answer is to the expected answer[27].

Risch et al., who introduced the SAS metric, provide a good example in their work[27]:

Question: How many plant species are estimated to be in the Amazon region?
Context: The region is home to about 2.5 million insect species, tens of thousands of plants, and some 2,000 birds and mammals. To date, at least 40,000 plant species [. . .]
Ground Truth / Expected Answer: "40,000"
Predicted Answer: "tens of thousands"
Exact Match: 0
F1 Score: 0.00
SAS Score: 0.55
Human Judgement: 0.50

This example demonstrates how simply taking F1 scores may not be a very good way to evaluate QnA models.

3.3 Evaluation Dataset

One benchmark for evaluating QnA systems is the Stanford Question Answering Dataset v2.0 (SQuAD2.0)[18]. It consists of over 150,000 questions posed by crowdworkers on a set of 500+ Wikipedia articles. The expected answers are annotated in the articles, an example of which is shown in Figure 3.1.

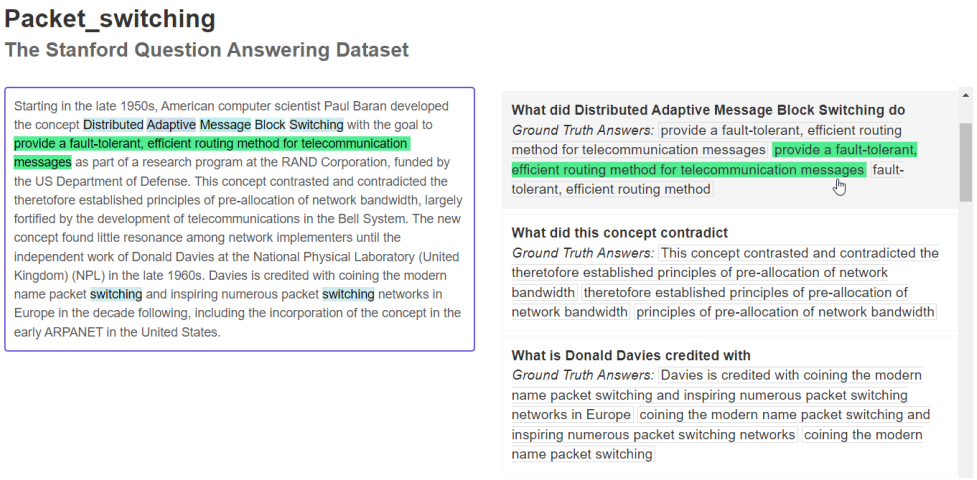

Figure 3.1: A screenshot of some example questions and ground truth answers in the SQuAD2.0 dataset.

AlgoFabric works in the domain of Finance, where financial documents, news, tweets, etc. are collected and analysed. SQuAD2.0 is a general dataset, so I went about creating my own dataset tuned to Finance for evaluating the QnA models in the applicable domain.

The creation of this evaluation dataset was done by taking around 500 news articles from a period of time, from the AlgoFabric's central database, and creating questions and annotating their answers in the Haystack Annotation Tool[28]. Not every article had relevant news, and some articles could have multiple questions. We call this custom dataset the **FinQuAD (Financial Question Answering Dataset)**.

The FinQuAD has 258 questions over 171 documents. An example screenshot is shown in Figure 3.2.

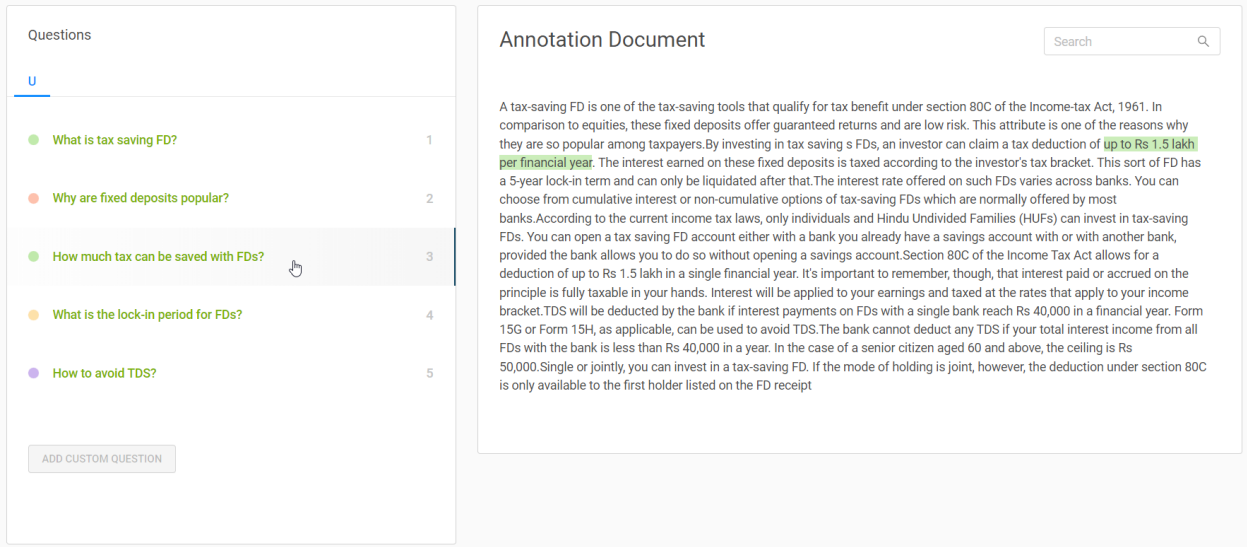

Figure 3.2: Screenshot of a question and its annotated answer in FinQuAD, through the Haystack Annotation Tool.

3.4 Results

All of the models have been evaluated on the custom-made FinQUaD dataset using computational resources in Google Colaboratory (better known as Colab). The models are named in the Hugging Face's Model Hub naming convention, which follows a `username/model-name` style. The username may be a person or an organisation, but is ultimately an online account providing the models on Hugging Face's Model Hub.

Sparse Retrievers perform simple Keyword Search. We use them as simple and fast retrievers to test out many Reader models, so that we may select a few for testing with Dense Passage Retrievers.

3.4.1 Sparse Retriever + Reader

Constant Parameters:

- Device
  - Tesla T4 GPU

- PreProcessor
  - Split by word, meaning a document split is done at the word level.
- Retriever
  - `top_k` = 5, meaning the Retriever would return the top 5 results.
- Reader
  - `top_k` = 3, meaning the Reader would return the top 3 answers.

Information			**Retriever Metrics**		**Reader Metrics**			**Pipeline**
split len	**Retriever**	**Reader**	**recall multi hit**	**map**	**f1**	**exact match**	**sas**	**time**
300	BM25	mfeb/albert-xxlarge-v2-squad2	94.51%	82.58%	84.01%	65.89%	84.18%	00:22:40
300	BM25	deepset/minilm-uncased-squad2	94.38%	81.69%	83.75%	67.44%	84.00%	00:01:48
300	BM25	ktrapeznikov/albert-xlarge-v2-squad-v2	94.51%	82.58%	83.63%	66.28%	83.69%	00:11:33
400	BM25	deepset/minilm-uncased-squad2	93.67%	83.82%	82.44%	67.05%	83.22%	00:01:57

Table 3.1 continued from previous page

Information			Retriever Metrics		Reader Metrics			Pipeline
split len	**Retriever**	**Reader**	**recall multi hit**	**map**	**f1**	**exact match**	**sas**	**time**
200	BM25	mfeb/ albert-xxlarge-v2-squad2	93.09%	78.52%	82.51%	64.73%	83.20%	00:15:34
200	BM25	deepset/ minilm-uncased-squad2	93.09%	78.52%	82.02%	66.67%	82.90%	00:01:39
300	BM25	deepset/ roberta-base-squad2	94.51%	82.58%	82.30%	67.05%	82.53%	00:03:15
400	BM25	deepset/ electra-base-squad2	94.57%	85.70%	79.87%	64.73%	82.35%	00:03:05
300	BM25	deepset/ electra-base-squad2	94.51%	82.58%	80.00%	64.34%	82.27%	00:02:47
200	BM25	deepset/ electra-base-squad2	93.09%	79.00%	79.63%	65.12%	82.17%	00:02:17
200	BM25	ktrapeznikov/ albert-xlarge-v2-squad-v2	93.09%	78.52%	81.93%	65.12%	82.10%	00:08:48
400	BM25	deepset/ roberta-base-squad2	94.57%	85.70%	81.84%	67.83%	81.88%	00:03:28

Table 3.1 continued from previous page

Information			Retriever Metrics		Reader Metrics			Pipeline
split len	**Retriever**	**Reader**	**recall multi hit**	**map**	**f1**	**exact match**	**sas**	**time**
200	BM25	deepset/ roberta-base-squad2	93.09%	78.52%	81.46%	67.44%	81.31%	00:02:57
300	BM25	mrm8488/ spanbert-finetuned-squadv2	94.51%	82.58%	80.11%	63.95%	81.21%	00:02:00
300	BM25	deepset/ tinyroberta-squad2	94.51%	82.61%	80.67%	64.34%	81.02%	00:03:12
400	BM25	mrm8488/ spanbert-finetuned-squadv2	94.57%	85.70%	80.43%	64.73%	80.95%	00:02:19
400	BM25	deepset/ tinyroberta-squad2	94.57%	85.70%	80.38%	64.34%	80.90%	00:03:25
200	BM25	deepset/ tinyroberta-squad2	93.09%	78.52%	79.92%	62.40%	80.76%	00:02:50
200	BM25	mrm8488/ spanbert-finetuned-squadv2	93.09%	78.52%	79.51%	62.02%	80.63%	00:01:44
200	BM25	squirro/ albert-base-v2-squad_v2	93.09%	78.52%	78.70%	62.79%	79.47%	00:02:20

Table 3.1 continued from previous page

Information			Retriever Metrics		Reader Metrics			Pipeline
split len	**Retriever**	**Reader**	**recall multi hit**	**map**	**f1**	**exact match**	**sas**	**time**
300	BM25	squirro/albert-base-v2-squad_v2	94.38%	82.02%	78.96%	63.18%	79.11%	00:02:41
400	BM25	squirro/albert-base-v2-squad_v2	94.44%	84.86%	78.41%	63.18%	79.08%	00:02:59
200	BM25	anablasi/qa_financial_v2	93.09%	78.52%	67.14%	50.78%	69.65%	00:01:44
200	BM25	distilbert-base-cased-distilled-squad	93.09%	78.52%	68.41%	50.78%	68.82%	00:01:33
300	BM25	anablasi/qa_financial_v2	94.51%	82.58%	65.06%	48.06%	67.41%	02:04.10
400	BM25	anablasi/qa_financial_v2	94.57%	85.70%	64.11%	48.45%	66.10%	00:02:06
400	BM25	distilbert-base-cased-distilled-squad	94.57%	85.70%	64.99%	49.22%	65.45%	00:02:04

Table 3.1 continued from previous page

Information			Retriever Metrics		Reader Metrics			Pipeline
split len	**Retriever**	**Reader**	**recall multi hit**	**map**	**f1**	**exact match**	**sas**	**time**
300	BM25	distilbert-base-cased-distilled-squad	94.51%	82.58%	63.62%	47.67%	64.39%	00:01:48
300	TF-IDF	deepset/roberta-base-squad2	123.90%	93.63%	54.68%	43.02%	58.84%	00:02:45
300	TF-IDF	deepset/tinyroberta-squad2	123.90%	93.63%	54.71%	43.41%	57.86%	00:02:47
300	TF-IDF	deepset/minilm-uncased-squad2	123.90%	93.63%	54.59%	42.64%	57.66%	00:01:33
300	TF-IDF	squirro/albert-base-v2-squad_v2	123.90%	93.63%	53.67%	40.31%	57.11%	00:02:32
300	TF-IDF	deepset/electra-base-squad2	123.90%	93.63%	49.64%	33.33%	53.28%	00:01:58

Table 3.1: Evaluation Metrics for Sparse Retriever + Reader pipeline. Results are arranged from highest SAS (Semantic Answer Similarity) score to lowest.

In the case of Sparse Retrievers, BM25 seems to provide better Recall and mAP than TF-IDF.

Testing a few different split lengths against multiple Reader models shows some interesting results. While the state-of-the-art is maintained by really large models like `mfeb/albert-xxlarge-v2-squad2`, the `deepset/minilm-uncased-squad2` model gives similar results but in a much lesser evaluation time. The `deepset/roberta-base-squad2` is a somewhat worse performing model but also with a small execution time. We take up these two Reader models for further testing with different Dense Passage Retrievers.

3.4.2 Dense Passage Retriever + Reader

Dense Passage Retrievers have the potential to perform a Semantic Search rather than a simple keyword search.

Constant Parameters:

- Device
  - Tesla T4 GPU
- PreProcessor
  - Split by word, meaning a document split is done at the word level.
- Retriever
  - `top_k` = 10, meaning the Retriever would return the top 10 results.
  - `embedding_dim`, the dimension of Word Embeddings, depends on the Query Encoding and Passage Encoding Models. This value is 768, except for the Deepset Bert-Small-MM-Retriever's Models where the dimension is 512.
  - Maximum Length of Query = 100 Characters
  - Maximum Length of Passage = 400 Characters
- Reader
  - `top_k` = 3, meaning the Reader would return the top 3 answers.

Looking at Table 3.2, the top performing model comes from Sohee Yang et al.[29], where they "distill the reader into the retriever so that the retriever absorbs the strength of the reader while keeping its own benefit". Combined with the MiniLM-Uncased Reader model, we get our best models in the pipeline for the AlgoFabric use case.

3.4.3 Chosen Models

- Dense Passage Retriever
  - Query Embedding Model: `soheeyang/rdr-question_encoder-single-nq-base`
  - Passage Embedding Model: `soheeyang/rdr-ctx_encoder-single-nq-base`
- Query Classifier: `shahrukhx01/question-vs-statement-classifier`
- Reader: `deepset/minilm-uncased-squad2`

These models have been chosen as they gave good results (very close to the top performing larger models) on the custom created FinQuAD evaluation set, and they were reasonably fast. A Semantic Search query on a CPU can finish in around 1-3 seconds, while one a question takes around 3-6 seconds. With GPUs it can come down to under a second.

Information				Retriever Metrics		Reader Metrics			Pipeline
split len	**DPR query embedding model**	**DPR passage embedding model**	**Reader**	**recall multi hit**	**map**	**f1**	**exact match**	**sas**	**time**
200	soheeyang/ rdr-question_encoder-single-nq-base	soheeyang/ rdr-ctx_encoder-single-nq-base	deepset/minilm-uncased-squad2	69.59%	50.59%	61.50%	48.06%	63.34%	00:03:12
200	soheeyang/ rdr-question_encoder-single-nq-base	soheeyang/ rdr-ctx_encoder-single-nq-base	deepset/tinyroberta-squad2	69.59%	50.59%	60.11%	44.96%	61.89%	00:05:56
300	soheeyang/ rdr-question_encoder-single-nq-base	soheeyang/ rdr-ctx_encoder-single-nq-base	deepset/minilm-uncased-squad2	68.52%	48.08%	60.31%	48.45%	61.92%	00:03:56
300	soheeyang/ rdr-question_encoder-single-nq-base	soheeyang/ rdr-ctx_encoder-single-nq-base	deepset/tinyroberta-squad2	68.52%	48.08%	59.81%	46.51%	61.74%	00:06:53
200	facebook/ dpr-question_encoder-single-nq-base	facebook/ dpr-ctx_encoder-single-nq-base	deepset/tinyroberta-squad2	64.71%	41.68%	55.41%	39.92%	58.99%	00:05:42
200	facebook/ dpr-question_encoder-single-nq-base	facebook/ dpr-ctx_encoder-single-nq-base	deepset/minilm-uncased-squad2	64.71%	41.68%	55.71%	42.25%	57.80%	00:03:05
300	facebook/ dpr-question_encoder-single-nq-base	facebook/ dpr-ctx_encoder-single-nq-base	deepset/minilm-uncased-squad2	63.71%	41.16%	54.82%	42.25%	57.26%	00:03:54
300	facebook/ dpr-question_encoder-single-nq-base	facebook/ dpr-ctx_encoder-single-nq-base	deepset/tinyroberta-squad2	63.71%	41.16%	54.56%	41.09%	57.13%	00:07:04
200	castorini/ mdpr-question-nq	castorini/ mdpr-passage-nq	deepset/tinyroberta-squad2	62.69%	40.33%	55.51%	41.47%	58.35%	00:05:41
200	castorini/ mdpr-question-nq	castorini/ mdpr-passage-nq	deepset/minilm-uncased-squad2	62.69%	40.33%	54.76%	42.64%	56.71%	00:02:59
300	castorini/ mdpr-question-nq	castorini/ mdpr-passage-nq	deepset/tinyroberta-squad2	61.35%	40.04%	54.51%	43.41%	58.01%	00:06:51
300	castorini/ mdpr-question-nq	castorini/ mdpr-passage-nq	deepset/minilm-uncased-squad2	61.35%	40.04%	54.73%	43.41%	56.35%	00:03:50
300	diarsabri/ LaDPR-query-encoder	diarsabri/ LaDPR-context-encoder	deepset/tinyroberta-squad2	61.32%	36.04%	52.65%	40.31%	55.65%	00:06:37
300	diarsabri/ LaDPR-query-encoder	diarsabri/ LaDPR-context-encoder	deepset/minilm-uncased-squad2	61.32%	36.04%	52.25%	40.31%	55.53%	00:03:52
200	diarsabri/ LaDPR-query-encoder	diarsabri/ LaDPR-context-encoder	deepset/minilm-uncased-squad2	55.65%	32.02%	48.21%	37.98%	51.94%	00:03:56
200	diarsabri/ LaDPR-query-encoder	diarsabri/ LaDPR-context-encoder	deepset/tinyroberta-squad2	55.65%	32.02%	48.05%	36.05%	50.90%	00:05:46
200	deepset/bert-small-mm_retrieval-question_encoder	deepset/bert-small-mm_retrieval-passage_encoder	deepset/minilm-uncased-squad2	45.03%	23.35%	42.30%	32.56%	45.65%	00:03:19
200	deepset/bert-small-mm_retrieval-question_encoder	deepset/bert-small-mm_retrieval-passage_encoder	deepset/tinyroberta-squad2	45.03%	23.35%	40.90%	28.68%	43.61%	00:05:56
300	deepset/bert-small-mm_retrieval-question_encoder	deepset/bert-small-mm_retrieval-passage_encoder	deepset/minilm-uncased-squad2	49.05%	22.84%	43.47%	32.17%	46.52%	00:03:52
300	deepset/bert-small-mm_retrieval-question_encoder	deepset/bert-small-mm_retrieval-passage_encoder	deepset/tinyroberta-squad2	49.05%	22.84%	42.56%	28.29%	44.66%	00:07:24

Table 3.2: Evaluation Metrics for Dense Passage Retriever + Reader pipeline. Results are arranged from the highest SAS (Semantic Answer Similarity) score to lowest.

Chapter 4

Conclusion

In conclusion, this thesis project explored the development and implementation of a semantic search and question-answering system in the domain of finance. The goal was to create a system that could retrieve relevant information based on the context and meaning behind a search query rather than just the exact keywords used. The system was also designed to answer question-like queries by extracting short answers from a vast pool of information, and to have a very modular setting where any model could work directly by just plugging it in. To accomplish the goals, the thesis project looked at the development of Word Embeddings, Dense Passage Retrievers, and Reader Models.

The thesis project provided a detailed account of the development and implementation of the system, including a sample architecture of the implementation. To evaluate the effectiveness of the system, the project looked at existing benchmarks and also created a custom dataset for comparing different models in the context of the domain of the work. Finally, the project concluded with the selection of the best models for the application.

Overall, the thesis project demonstrated the potential of semantic search and question-answering systems in the domain of finance, and showed how Dense Passage Retrievers, and Reader Models can be used to develop and implement such systems. The project has practical applications in the financial industry.

References

[1] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[2] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. arXiv: 1310.4546 [cs.CL].

[3] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[4] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[5] Vladimir Karpukhin et al. "Dense Passage Retrieval for Open-Domain Question Answering". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online: Association for Computational Linguistics, Nov. 2020, pp. 6769–6781. DOI: 10.18653/v1/2020.emnlp-main.550. URL: https://aclanthology.org/2020.emnlp-main.550.

[6] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (3rd ed. draft)*. 2023.

[7] Danqi Chen et al. *Reading Wikipedia to Answer Open-Domain Questions*. 2017. arXiv: 1704.00051 [cs.CL].

[8] H. P. Luhn. "A Statistical Approach to Mechanized Encoding and Searching of Literary Information". In: *IBM Journal of Research and Development* 1.4 (1957), pp. 309–317. DOI: 10.1147/rd.14.0309.

[9] K. Spärck Jones. "A Statistical Interpretation of Term Specificity and Its Application in Retrieval". In: *Journal of Documentation* 28.1 (1972), pp. 11–21. DOI: 10.1108/eb026526.

[10] Karen Spärck Jones, S. Walker, and Stephen E. Robertson. "A probabilistic model of information retrieval: Development and comparative experiments". In: *IP&M* 36.6 (2000), pp. 779–808, 809–840.

[11] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, July 2008. URL: https://nlp.stanford.edu/IR-book.

[12] Eve V Clark and Brian MacWhinney. "The principle of contrast: A constraint on language acquisition". In: *Mechanisms of language acquisition* (1987), pp. 1–33.

[13] Felix Hill, Roi Reichart, and Anna Korhonen. "Simlex-999: Evaluating semantic models with (genuine) similarity estimation". In: *Computational Linguistics* 41.4 (2015), pp. 665–695.

[14] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[15] Yuening Jia. "Attention mechanism in machine translation". In: *Journal of physics: conference series*. Vol. 1314. 1. IOP Publishing. 2019, p. 012186.

[16] Jeff Johnson, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs". In: *IEEE Transactions on Big Data* 7.3 (2019), pp. 535–547.

[17] Meta Research. *FAISS, a library for efficient similarity search and clustering of dense vectors.* Version 1.7.3. Nov. 30, 2022. URL: https://github.com/facebookresearch/faiss.

[18] Pranav Rajpurkar, Robin Jia, and Percy Liang. *Know What You Don't Know: Unanswerable Questions for SQuAD*. 2018. arXiv: 1806.03822 [cs.CL].

[19] deepset GmbH. *Haystack, a framework for NLP tools in Python.* Version 1.13.2. Feb. 10, 2023. URL: https://haystack.deepset.ai/.

[20] Sebastián Ramírez. *FastAPI, a web framework for building APIs in Python.* Version 0.92.0. Feb. 14, 2023. URL: https://fastapi.tiangolo.com/.

[21] Paul J. Leach, Rich Salz, and Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. July 2005. DOI: 10.17487/RFC4122. URL: https://www.rfc-editor.org/info/rfc4122.

[22] Felipe Pezoa et al. “Foundations of JSON schema”. In: *Proceedings of the 25th international conference on World Wide Web*. 2016, pp. 263–273.

[23] Amazon Web Services Documentation. *Paginating table query results in DynamoDB*. 2023. URL: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.Pagination.html.

[24] Microsoft Azure Documentation. *Partitioning and horizontal scaling in Azure Cosmos DB*. 2023. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/partitioning-overview.

[25] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[26] Haystack Documentation. *Evaluation in Haystack*. 2023. URL: https://docs.haystack.deepset.ai/docs/evaluation.

[27] Julian Risch et al. *Semantic Answer Similarity for Evaluating Question Answering Models*. 2021. arXiv: 2108.06130 [cs.CL].

[28] deepset GmbH. *Haystack Annotation Tool*. 2023. URL: https://annotate.deepset.ai/.

[29] Sohee Yang and Minjoon Seo. *Is Retriever Merely an Approximator of Reader?* 2020. arXiv: 2010.10999 [cs.CL].