Document-based question answering

Hachem Betrouni[1,2]

[1]National Polytechnic school of Algiers, Industrial Engineering Department, Data Science and AI, Algiers, hachem.betrouni@g.enp.edu.dz
[2]BIGmama technology, France, hb@big-mama.io

June 15, 2023

Abstract

In this thesis, we present a comprehensive analysis of various open-source embedding models and large language models (LLMs) for document-based question answering (DBQA). The goal of DBQA is to extract relevant information from a given document and answer user queries in natural language. We evaluate the performance of different embedding models, including BERT, E5, and MiniLM, in combination with open source LLMs such as Vicuna, Falcon, and OpenLlama. Our experimental setup involves a retriever-generator framework, where a retrieval system retrieves the most relevant contexts from a document using embedding models. The conditioned generative system then generates a response to the user query based on the retrieved contexts using LLMs. We benchmark the performance of these systems using the SQuAD dataset. We find that the choice of embedding model significantly impacts the retrieval performance. Moreover, we observe that the choice of LLM also plays a crucial role in generating accurate and informative answers.unsrt This comparative analysis provides insights into the strengths and weaknesses of various open-source embedding models and LLMs for DBQA tasks. The findings of this study can guide researchers and practitioners in selecting suitable models for their specific DBQA applications and contribute to the advancement of question-answering systems.

To my parents,
to my family, mentors, friends, enemies, and every person who shaped who I am,
to every rock that broke my bones, to every sea waving on the shores,
to Algiers's seagulls waking me up morning, reminded of Allah's boons,
to Collo's men who rescued me,
to you all I dedicate not this thesis but the years that led to its maketh.

Contents

List of Figures

List of Tables

1 General introduction

For this BIGmama confined the development of a document-based question answering model to us.

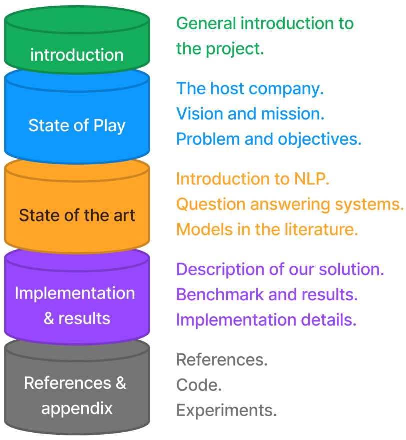

Figure 1: General structure of this document.

2 State of play

2.1 Introduction

Creating value in the market is usually about solving a complex problem or answering a question. When it comes to the Data science and AI sector this used to be tackled with off-the-shelf solutions/applications/answers. However the arrival of LLMs and other fundamental models upset the level of requirements. Creating value now requires expensive, bespoke AI applications. The requirements are no longer to predict a number, but to explain the prediction, it's no longer to recommend a blog title but to generate its whole content.

2.2 Asset-based consulting

This evolvement of the business landscape, driven by rapid technological advancements, changing market dynamics, and increasing competition created a fast-paced environment, where organizations seek innovative ways to optimize their operations, unlock hidden potentials, and achieve sustainable growth.

Today we are well aware that answering these organizations needs for innovation has shifted from writing consulting reports (or any of the traditional approaches) to building tangible assets that concretize the added value. Today, clients are expecting specialised solutions to their problems, rather than the boilerplate solutions consultants traditionally provided.

This has given rise to a thriving industry known as asset-based consulting (ABC).

Asset-based consulting is a specialized field within consulting that focuses on identifying, leveraging, and maximizing the value of a company's assets, AI can be incorporated in the schema of consulting by leveraging the data/knowledge/expertise of the company, incorporating it with intelligent systems, automatize tasks and improve the productivity of its users in general.

In the ABC context, assets can include tangible resources such as infrastructure, technology, and inventory, as well as intangible assets like intellectual property, brand equity, and human capital. By harnessing the full potential of these assets, asset-based consultants help organizations drive operational efficiency, enhance performance, and create a competitive advantage.

Unlike traditional consulting approaches that often emphasize external factors and strategies, asset-based consulting takes a more holistic and internal perspective. It recognizes that companies possess unique assets and capabilities that, when strategically managed, or in our case, combined with AI, can serve as the foundation for sustainable competitive advantage and resilience. Here consultants work closely with clients to understand their expertise, identify untapped questions and needs, and design tailored solutions.

Asset-based consulting is known for its multidisciplinary nature, drawing expertise from various fields including finance, operations, technology, marketing, and human resources. Consultants blend their industry knowledge, analytical skills, and business acumen to assess, diagnose, and unlock the hidden value with innovative assets. They often conduct thorough assessments, employ data-driven methodologies, and collaborate closely with expert teams to align the asset usefulness with the overall business objectives.

As industries become increasingly complex and competitive, the demand for asset-based consulting continues to rise. Organizations of all sizes and sectors recognize the need to unlock the value within their assets to stay ahead of the curve. From manufacturing firms seeking to optimize their supply chains to technology companies aiming to capitalize on their intellectual property, asset-based consulting provides a strategic framework and expertise to achieve these objectives.

2.3 BIGmama technology

BIGmama is a French registered startup with an Algerian extension, specialized in data science and AI. They have been developing bespoke predictive applications for more than 8 years (as of the year 2023).

The board of directors counts former CEOs of global groups (Danone, Safran) and the team is assembled from a dozen of high-level data scientists and software engineers.

2.4 Mission

BIGmama's mission is to democratize the access to bespoke AI applications, and to transform companies and individuals by introducing AI in their jobs however possible.

From an original method based on the know-how developed while working with their partners for nearly a decade that we automate or increase with algorithms (Hybrid AI).

What is important at BIGmama today is to be able to industrialize this methodology by proposing a software that encapsulates and hybridizes this expertise with ML models. In this way, this tool will allows them to build state of the art predictive applications, relatively quickly and at an affordable cost.

They believe that a large part of the future of the AI sector will be played out in the ability to articulate human expertise with the countless possibilities offered by AI models.

The last version of the software they are developing (HYKO), should be released by this June.

2.5 Vision

Data science will soon become a commodity, the arrival of large language models (which are considered foundational models in NLP) mark but a start of a new wave of foundational models. In the near future, we can expect the emergence of generalist models that are equivalent to ChatGPT but specialized in computer vision tasks, forecasting, pattern recognition, and other domains.

AI projects that take 6 months in the making, require extensive "hyper-parameter tuning", hundreds of experiments, task specific architectural modeling, and that come at an expensive price will soon become obsolete and outdated.

The future is going to become in the hands of those who are capable of harnessing the power of these generalist models. Algeria and the whole African continent is far behind, and its no longer a question of closing this gap step by step, a "quantum jump" is required. BIGmama will lead this quantum jump, and will prove that excellence can beam from within the African continent.

2.6 Unique methodology

One of the valuable heritages that BIGmama acquired during the 8 years of actively developing bespoke AI applications to its clients is a the unique methodology of work at BIGmama. it's centered around the idea is

2.6.1 AI starts with problematization.

Data science does not start with data science but with a re-framing and problematization work. Our clients most often arrive with a subject (and not problems). A subject in it self is composed of many "hidden" problems (often hidden even to the client), a good consultant at BIGmama knows that this process of problematization (that of decomposing the subject into multiple problems) requires multiple iterations and attempts, and its success is deeply rooted to that of understanding the expertise of the client, their needs and to the skills of the consultant to translate these problems into something addressable by the current AI models.

2.6.2 AI is a tool, not an end-goal

At BIGmama, we firmly believe that AI should be seen as a powerful tool rather than an ultimate goal in itself. While AI technology continues to advance rapidly, its true value lies in its ability to address and solve specific problems or challenges faced by individuals, businesses, and society as a whole. By recognizing AI as a tool, we emphasize its role in enabling and enhancing human capabilities rather than replacing them. And we emphasis that the current approaches in AI are not always the best to solve *every* problem.

2.6.3 Hybridization

We believe that the future of AI lies in what is commonly referred to today as hybrid AI. This is a set of approaches and methodologies aimed at combining the potential of models with purely human knowledge. We believe that this hybridization allows us to put humans at the heart of technological development and to produce tools that are more efficient, easier to maintain, explain and that are far less expensive.

2.7 Software as a service (SaaS)

BIGmama is currently working on a software that intrinsically embeds its mission of democratizing AI and its unique methodology of how to do so. Hyko is an iterative software that help users formulate their problems properly using AI conversational systems powered by LLMs. It then automatically generate a working applications prototypes that combines and orchestrate multiple AI models predisposed in the user's model-base.

Hyko can be described as a three steps iterative process :

Diagnosis and problematization (Scoping)

Automatic prototyping Hyko leverages the ever increasing availability of open-source AI models (Huggingface [43], Github, etc.) and the zero-shot reasoning capabilities [20] of today's LLMs to solve the following tasks :

1. task planning : designing how the tasks should be addressed, setting up the inputs, outputs and the dependencies of each task in the pipeline.

2. model selection : selecting the right model in each step of the pipeline.

Similar to HuggingGPT [36], by solving these tasks Hyko is capable of going from a description of the user's problems to a working AI prototype solution.

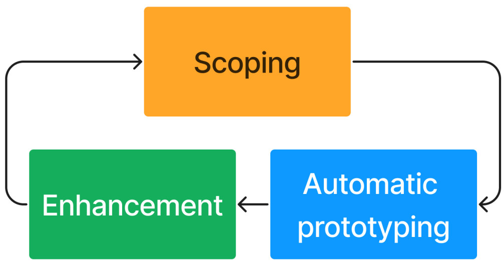

Figure 2: Iterative process of developing an AI asset in Hyko.

Enhancement of the prototype

2.8 Knowledge representation

A very important aspect of the first step (Scoping) is the ability to represent already existing knowledge (technical documents for example) and make it digestible by the LLMs behind the scoping phase.

2.8.1 Document-based question answering

Document-based question answering (DBQA) is a machine learning task that focuses on extracting accurate and relevant answers from a given document and responding in natural language to a specific query.

It involves representing and manipulating content within textual documents, such as articles, technical documentation in order to provide precise answers to user queries.

DBQA systems typically involve a two stages process :

1. context retrieval : employing information retrieval (IR) techniques to extract relating contextual passages from the document.

2. question answering (QA) : using natural language processing (NLP) approaches to answer a query using the retrieved context.

2.9 Conclusion

In this paper we will dive deeper into the different frameworks, models and techniques used in DBQA systems (and similar QA tasks).

In this project we experimented with different DBQA systems, we prepared a small benchmark for few open-source LLMs used (in many combinations) with simple context retrieval systems.

3 State of the art

3.1 Introduction

3.1.1 NLP Tasks

3.1.2 Embeddings

A word embedding is a technique used in NLP to represent words as dense vectors in a high-dimensional space (an embedding). It aims to capture the semantic and syntactic relationships between words, enabling algorithms to process language more effectively.

Word embeddings can be created using two main approaches (or a combination of both): neural and statistical. Neural approaches, use neural networks to learn word representations by capturing semantic and syntactic relationships. Statistical approaches, rely on statistical features of words and their occurrences.

TF-IDF TF-IDF (Term Frequency-Inverse Document Frequency) is a commonly used weighting scheme in information retrieval and text mining tasks, designed to capture the importance of terms in a document collection. It combines two fundamental concepts: term frequency (TF) and inverse document frequency (IDF). TF refers to the number of times a term appears in a document, normalized by the total number of terms in that document. IDF, on the other hand, quantifies the rarity of a term across the entire document collection by taking the logarithm of the inverse of its document frequency.

Document frequency refers to the number of documents in the collection that contain a given term. The TF-IDF weight for a term in a document is obtained by multiplying its TF value by its IDF value. This approach assigns higher weights to terms that appear frequently in a specific document but are relatively rare across the entire collection, effectively capturing their discriminative power. the TF-IDF weight of a term t in a document d can be computed as follows:

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

$$\mathrm{idf}(t,D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

$$\mathrm{tfidf}(t,d,D) = \mathrm{tf}(t,d) \cdot \mathrm{idf}(t,D)$$

where $f_{t,d}$ is the raw count of a term in a document, i.e., the number of times that term t occurs in document d, N the total number of documents in the corpus $N = |D|$ and $|\{d \in D : t \in d\}|$ is the number of documents where term t apears.

The resulting TF-IDF weights can be used for various purposes, such as document ranking, text classification, or keyword extraction, enabling the identification of important terms that characterize the content of documents in a collection.

Bert The BERT model [9] is pre-trained on large-scale corpora using unsupervised learning, followed by fine-tuning on specific downstream tasks. The resulting contextual word embeddings capture intricate semantic and syntactic relationships, enabling BERT to excel in various NLP tasks However BERT [9] uses a cross-encoder (i.e. two sentences are passed to the transformer network and the target value is predicted). This makes it unsuitable for various pair regression tasks due to too many possible combinations (finding most similar pair in a collection of n sentences requires $\frac{n*(n-1)}{2}$ inferences). Sentence-BERT (SBERT) [33], is a modification of the BERT network using siamese and triplet networks, SBERT is able to derive semantically meaningful sentence embeddings (semantically similar sentences are close in vector space).

E5 According to Huggingface embedding models leaderboard [27], the E5 [40] is the current stat-of-the-art (average performance across 62 datasets and more than 7 tasks).

MiniLM MiniLM [41] uses a simple and effective knowledge distillation method to compress large pre-trained Transformer based language models. The student model (MiniLM) is trained by deeply mimicking the teacher's self-attention [39] modules, which are the vital components of the Transformer networks.

The authors of [41] propose using the self-attention distributions and value relation of the teacher's last Transformer layer to guide the training of the student, which is effective and flexible for the student models.

The embeddings that come from this model are of high quality, cheap to compute and can be used in a number of down stream tasks.

3.1.3 Transformers

Transformers are a type of deep learning model that have revolutionized natural language processing (NLP) tasks. They were introduced by [39] and have since become the state-of-the-art approach for various NLP applications.

Attention Mechanism:

Attention mechanisms are a key component of transformers. They allow the model to focus on different parts of the input sequence when processing each element. The attention mechanism assigns weights to each element in the input sequence based on its relevance to the current element being processed.

Given an input sequence $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$, the attention mechanism computes a weighted sum of all elements in $\mathbf{X}$, where the weights are determined by the relevance of each element to the current element. This can be mathematically represented as:

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

where $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are the query, key, and value matrices, respectively, and d_k represents the dimension of the key.

Multi-Head Attention:

To capture different types of information, self-attention is performed multiple times in parallel, with different learned linear projections. The outputs of these attention heads are concatenated and linearly transformed:

$$MultiHead(X) = Concat(head_1, head_2, ..., head_h)W_O$$

where $head_i = SelfAttention(XW_{Qi}, XW_{Ki}, XW_{Vi})$, W_{Qi}, W_{Ki}, and W_{Vi} are learnable weight matrices, and W_O is the output weight matrix.

Encoder Transformer:

In a transformer model, the encoder is responsible for encoding the input sequence. It consists of a stack of identical layers, each comprising a multi-head self-attention mechanism and a feed-forward neural network. The encoder processes the input sequence in parallel and captures the dependencies between different elements.

Let $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$ be the input sequence. The encoder takes $\mathbf{X}$ and produces a sequence of encoded representations $\mathbf{Z} = \{z_1, z_2, \ldots, z_n\}$. The encoding process can be expressed as:

$$\mathbf{Z} = Encoder(\mathbf{X}) = EncoderLayer(\ldots(EncoderLayer(\mathbf{X})))$$

where $EncoderLayer$ represents a single layer in the encoder, and the ellipsis denotes the stacking of multiple layers.

Encoder-Decoder Architecture:

The encoder-decoder architecture extends the transformer model to tasks such as machine translation, where the input and output sequences have different lengths. The encoder processes the input sequence, while the decoder generates the output sequence.

Let $\mathbf{X} = \{x_1, x_2, \ldots, x_m\}$ and $\mathbf{Y} = \{y_1, y_2, \ldots, y_n\}$ be the input and output sequences, respectively. The encoder-decoder architecture can be summarized as:

$$\mathbf{Z} = Encoder(\mathbf{X})$$
$$\mathbf{Y}' = Decoder(\mathbf{Z}, \mathbf{Y}) = DecoderLayer(\ldots(DecoderLayer(\mathbf{Z}, \mathbf{Y})))$$

where $Decoder$ represents the decoder, $\mathbf{Z}$ is the encoded representation of $\mathbf{X}$ obtained from the encoder, and $\mathbf{Y}'$ is the predicted output sequence.

The decoder processes the output sequence $\mathbf{Y}$ and attends to the encoded representation $\mathbf{Z}$ to generate the output. The decoder also employs self-attention, allowing it to attend to previously generated elements in the output sequence.

3.1.4 Similarity measures

Similarity measures are used to quantify the degree of similarity or dissimilarity between two objects or vectors. They are widely used in various domains, including information retrieval, recommendation systems, and clustering. Different similarity measures capture different aspects of similarity based on the specific requirements of the task.

Here we mention a few examples of similarity measures commonly used in the litterateur. However the selection of a suitable similarity measure is contingent upon the characteristics of the data and the particular objectives of the task being performed.

Cosine Similarity:

Cosine similarity is a commonly used similarity measure that computes the cosine of the angle between two vectors. It is particularly useful when comparing the similarity between documents, text embeddings, or high-dimensional data. The cosine similarity between two vectors $\mathbf{A}$ and $\mathbf{B}$ can be calculated as:

$$cosine\_similarity(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

where $\cdot$ denotes the dot product and $\| \cdot \|$ represents the Euclidean norm.

Euclidean Distance:

Euclidean distance is a commonly used dissimilarity measure that calculates the straight-line distance between two points in Euclidean space. It is widely used in clustering algorithms, such as k-means. The Euclidean distance between two vectors $\mathbf{A}$ and $\mathbf{B}$ of the same dimension can be computed as:

$$euclidean\_distance(\mathbf{A}, \mathbf{B}) = \sqrt{\sum_{i=1}^{n} (A_i - B_i)^2}$$

where A_i and B_i represent the i-th elements of vectors $\mathbf{A}$ and $\mathbf{B}$, respectively.

Jaccard Similarity:

Jaccard similarity [16] is a measure commonly used for comparing the similarity between sets. It is particularly useful in text mining and recommendation systems. The Jaccard similarity between two sets A and B is calculated as the size of their intersection divided by the size of their union:

$$jaccard\_similarity(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where $|A|$ and $|B|$ denote the cardinalities of sets A and B, respectively.

Hamming Distance:

Hamming distance [12] is a similarity measure used for comparing binary vectors of the same length. It calculates the number of positions at which the corresponding elements of two vectors are different. The Hamming distance between two binary vectors $\mathbf{A}$ and $\mathbf{B}$ can be computed as:

$$hamming\_distance(\mathbf{A}, \mathbf{B}) = \sum_{i=1}^{n} (A_i \oplus B_i)$$

where A_i and B_i represent the i-th elements of vectors $\mathbf{A}$ and $\mathbf{B}$, respectively, and $\oplus$ denotes the bitwise XOR operation.

3.2 Many LLMs

Plethora of large language models (LLMs) are being released week upon week. Some models are made open-source : OpenLlama [11], GPT-2 [31], others are proprietary : GPT-3[4] and GPT-4 [28], some are published for commercial use such as Falcon [1], others are only for research purposes : Llama [38] and Vicuna [6].

Its hard to benchmark the performance of these models due to the wide range of applicable evaluation tasks and the difficulty of filtering out the genuine progress from grandiose claims as well as problems with current QA datasets and unrigorous evaluations [23].

Huggingface recently published an LLM leaderboard [2], where they evaluate publicly available LLMs on 4 key benchmarks :

1. AI2 Reasoning Challenge (25-shot) - a set of grade-school science questions [8].
2. HellaSwag (10-shot) - a test of commonsense inference, which is easy for humans (95%) but challenging for SOTA models [45].
3. MMLU (5-shot) - a test to measure a text model's multitask accuracy. The test covers 57 tasks including elementary mathematics, US history, computer science, law, and more [13].
4. TruthfulQA (0-shot) - a benchmark to measure whether a language model is truthful in generating answers to questions [25].

Most of the top performing models in this benchmark are either fine-tuned Falcon [1] or fine-tuned LLama [38].

Falcon Large language models are typically trained using a combination of carefully selected high-quality data sources and filtered web data. The curation process aims to create models that perform well across a wide range of tasks. However, as the size of models increases and more training data is required, concerns arise regarding the scalability of the curation process and the availability of unique high-quality data.

The refined web dataset for Falcon llm [29] demonstrates that properly filtered and deduplicated web data alone can yield powerful language models, even outperform state-of-the-art models trained on curated datasets like The Pile [10]. The authors of this dataset[29] have released the Falcon [1] language models with 1.3/7.5 billion parameters trained on this dataset, which serve as valuable resources for this project.

LLaMa The authors of [38] present LLaMA, a collection of foundation language models with parameters ranging from 7 billion to 65 billion. They demonstrate that it is possible to train state-of-the-art models using publicly available datasets alone, without relying on proprietary or inaccessible data. In particular, LLaMA-13B surpasses the performance of GPT-3 (175B) [4] on most benchmarks, and LLaMA-65B is on par with other top models like Chinchilla-70B [15] and PaLM-540B [7]. By making open-sourcing the LLaMA architecture and training code and by making the weights available for researchers, plenty of other models appeared as a result of fine-tuning the original LLaMA models on various tasks and datasets, these fine-tuned models include Alpaca [37] and Vicuna [6].

OpenLLaMa [11] provides a totally open-source replication (weights and code, no research-only restriction) of the original LLaMA [38] models and which surprisingly in some cases outperforms the original models according to their benchmark [11]

3.3 Question answering

3.3.1 Visual QA

3.3.2 Open-domain QA

Open-domain Question Answering (ODQA) is a task examining the ability of models to produce answers to natural language factoid questions drawn from an open set of domains.

DrQA [5] implemented Wikipedia as its knowledge source and this choice has became a default setting for many ODQA studies since then.

However it's relevant in some cases (such as ours) to limit the knowledge source to a signal (or multiple) pre-defined documents, there is a slight difference between the two tasks, to avoid confusion we call the latter "document-based question answering".

Nevertheless approaches in ODQA are applicable for DBQA.

3.4 Methods for document-based question answering

3.4.1 Open-book QA

Open-book models in the field of Open-Domain Question Answering (ODQA) initially retrieve relevant documents then either extract or generate answers based on the information contained in the retrieved documents. We can distinguish mainly two approaches to tackle the ODQA problem :

1. Retriever-reader : this model works toward finding the related context in the documentation than process the retrieved context to **extract** the start/end positions of an answer. The output of the model is the selected context and the identified span of the answer in the context.

2. Retriever-generator : unlike the reader model, the generator model generate free text conditioned with the retrieved context to answer the question.

Retriever-reader Dense Passage Retrieval (DPR) [18], follows a pipeline approach, where documents are retrieved using dense embeddings. These retrieved documents are then passed to a conventional reader-re-ranker, which extracts specific spans of text as answers.

Retriever-generator Retrieval-Augmented Generation [22] is a seq2seq model that jointly learns to retrieve and generate answers. It utilizes dense retrieval and BART [21] for this purpose.

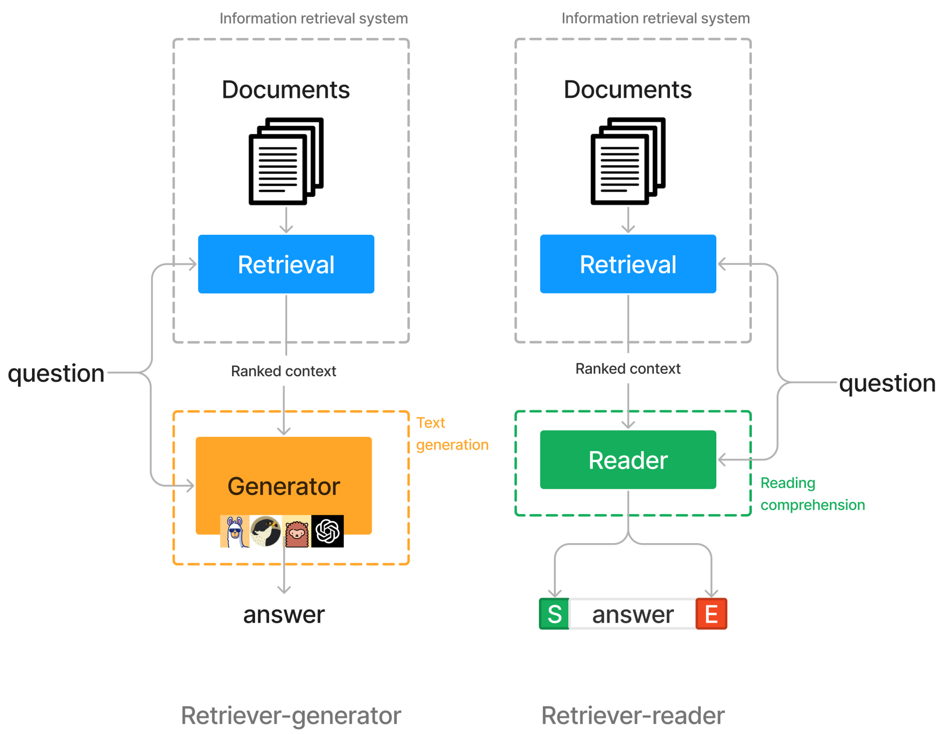

Figure 3: Overview of open-book question answering approaches.

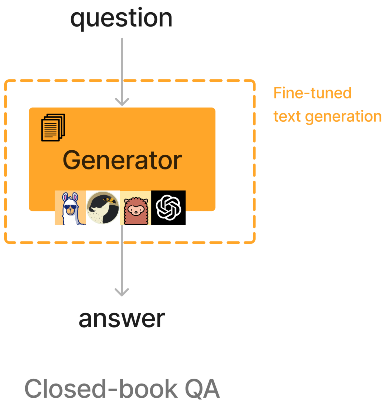

Figure 4: Overview of closed-book fine-tuned QA.

3.4.2 Closed-book QA

Large language models undergo extensive unsupervised pre-training phase using large amount of textual data. With a substantial number of parameters, these models possess the ability to memorize factual information within their weight parameters. Consequently, one is able to employ this property for question-answering tasks without relying on explicit context (shown in figure 4).

The pre-trained language models generate free-form text responses to questions, without explicitly employing reading comprehension techniques. Closed-book models can encode the given documentation within the parameters of the model itself to answer queries, rather than using a retrieval model.

Authors of [34] fine-tune a pre-trained T5 [32] model to answer questions (without access to any external context or knowledge) and were able (at the time) to compete with open-domain systems that explicitly retrieve answers from an external knowledge source when answering questions.

GPT-3 [4] has been evaluated on the closed book question answering task using the TriviaQA dataset [17] without any gradient updates or fine-tuning, the evaluation (figure 5) shows that GPT-3 match/exceed the performance of state of the art (at that time).

3.5 Retrieval models

When it comes to implementing a retriever for a retriever-generator/retriever-reader models, there is mainly two systems :

1. using classic non-learning-based TF-IDF features (“classic IR”).
2. or using dense embedding vectors of text produced by neural networks (“neural IR”).

classic IR For example DrQA [5] adopts an efficient non-learning-based search engine based on bigram hashing and TF-IDF matching.

Another approach used by BERTserini [44] consists of ranking retrieved text segments using BM25 [35], a classic TF-IDF-based retrieval scoring function.

In terms of the effect of text granularity on performance, [44] found that paragraph retrieval ¿ sentence retrieval ¿ article retrieval.

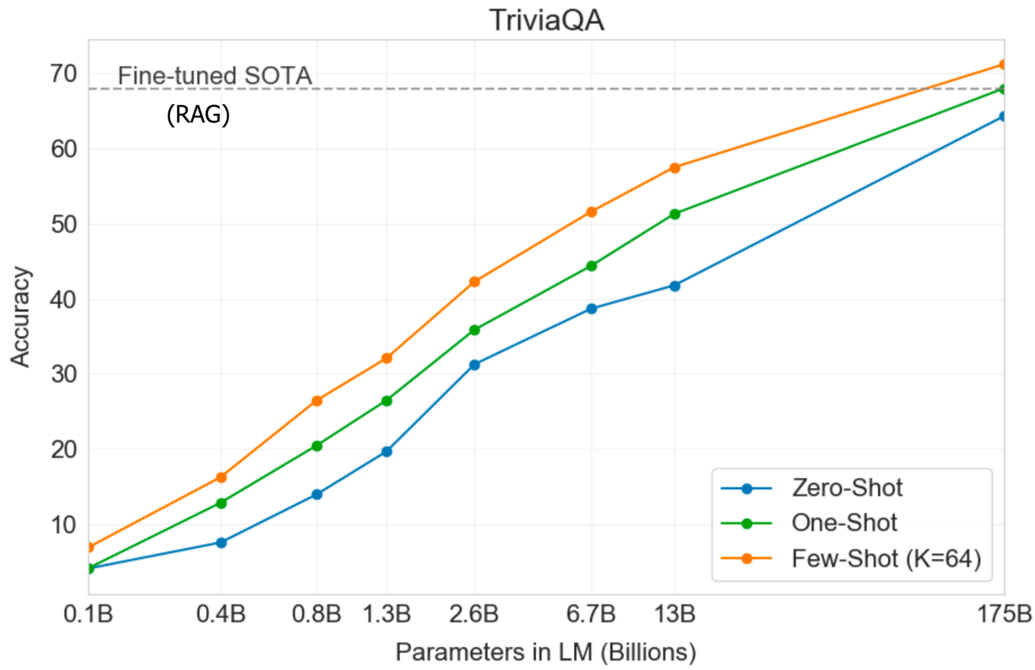

Figure 5: GPT3's performance on TriviaQA [17]. [4]

Multi-passage BERT QA model [42] uses elasticSearch with BM25 [35]. They found that splitting articles into passages with the length of 100 words by sliding window brings 4% improvements, and that splitting documents into passages without overlap causes some near-boundary evidence to lose useful contexts thus decreasing the performance.

neural IR Neural IR is a new category of methods for retrieval problems, it mainly uses dense representations of some neural network architectures (e.g. LSTM [14], BERT [9], etc).

After the arrival of many "general language models" (to do cite llms) many IR for QA systems follow this (or a slightly different) approach :

1. Extract the dense representations of a question and a context passage by feeding them into a language model.

2. Use the dot-product of these two representations as the retrieval score to rank and select most relevant passages.

$$h_q = E_q(q)$$
$$h_z = E_z(z)$$
$$score = h_q^\intercal h_z$$

(however it is not necessary that neural IR out-performs classic IR [24])

3.6 Reader Models

The reader model objective is to learn solve reading comprehension tasks, extract an answer for a question from a retrieved context. Here we only discuss approaches for machine comprehension using neural networks since it yields best performing results.

4 Implementation and results

4.1 Introduction

In our Implementation we chose a retriever-generator approach, we experimented with different combinations of embedding and Language models, as well as different similarity measures and text splitting strategies.

We evaluated the different systems using a sample from the SQuAD [30] dataset and GPT-3 [4] as a scoring model, more about that in the sections below.

LlamaIndex [26] tested out few indexing methods (embeddings, LLMs and retrieval systems) following a similar framework to ours. However their focus was on OpenAI proprietary language models (GPT-3 and GPT-4) [4, 28]. Our focus in this project on the other hand is on using open-source embedding models and LLMs that can fully run on-premise, since model and data sovereignty are a critical aspect for our use-case.

In the following sections we will provide more details on our implementation as well as the evaluation results we got.

4.2 challenges

The following challenges arise when using LLMs for DBQA :

1. Small context window: LLMs typically have a limited context window, which means they can only consider a fixed number of preceding and succeeding tokens when generating a response. In DBQA, where the answer may depend on information scattered throughout a document, this limited context window can make it difficult for LLMs to capture the necessary context and provide accurate answers. Relevant information may fall outside the context window, leading to incomplete or incorrect responses.

2. Hallucination: LLMs can sometimes generate responses that are plausible-sounding but factually incorrect. This phenomenon, known as hallucination, can be a significant issue in DBQA. Since LLMs rely on statistical patterns and language modeling rather than true understanding, they may generate answers that sound reasonable but are not grounded in the actual content of the document. Hallucination can mislead users and compromise the reliability of the DBQA system.

3. Storing the models and computational demand for inference: LLMs, especially large-scale models, require substantial computational resources for both training and inference. Storing these models and performing inference can be challenging due to their size and computational demands. DBQA systems that utilize LLMs need to address issues such as efficient model storage, quick retrieval of relevant models, and the computational infrastructure required to perform real-time inference on large documents. These demands can pose practical challenges for deploying LLM-based DBQA systems at scale.

We were able to workaround the first challenge using IR system based on neural-embedding. For the second challenge we tried using small temperature (a hyperparameter used to control the randomness of the generated text) and presence penalty values (high presence penalty values results in the model being more likely to generate tokens that have not yet been included in the generated text and vice-versa). Using smaller models (7B parameters each) (required around 36GB of memory) and allowed us to run the inferences on a multi-GPU machine.

	Language model		
Embedder	Vicuna	Falcon	OpenLlama
Bert	**0.4**	0.25	0.35
MiniLM	0.35	0.25	0.3
E5	0.35	0.25	0.3

Table 1: Benchmark : score of each system when tested on 20 samples from SQuAD [30] dataset, and evaluated using GPT-3 [4].

4.3 Description

The systems that we experimented with follows the retriever-generator framework which consists mainly of two parts :

a retrieval system that takes as input a document (text) and a query from the user, it splits the document into sentences using a heuristic algorithm developed for the processing the Parallel Corpus for Statistical Machine Translation [19]. Each sentence from the document along the query of the user are going to be embedded by one of the chosen embedding models. We measure the one-to-one similarity (using cosine similarity) between the query embedding and each sentence embedding and we retrieve the top K sentences to be used a context.

a conditioned generative system it takes as input the retrieved contexts and the query to construct a prompt that condition the generation using one of the chosen open-source LLMs. In table 1 we show that we tested all the possible combination of the different embedding models and LMs at hand.

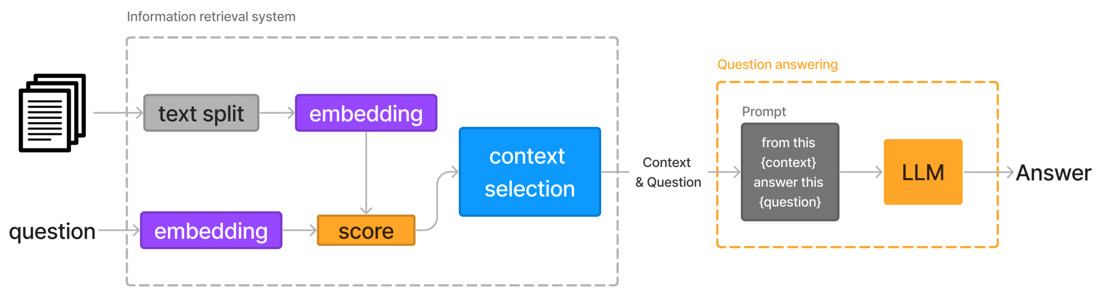

Figure 6: Highlevel overview of the whole system.

4.4 Evaluation

We bench-marked the different systems on the SQuAD dataset [30]. We sampled 20 with there context and answer (examples can be found in this annex 5.2), for each question and context we tested a different embedding and LLM, each response is them evaluated using GPT-3 [4] (a score of +1 is given if the answer is correct 0 otherwise), results are shown in table 1.

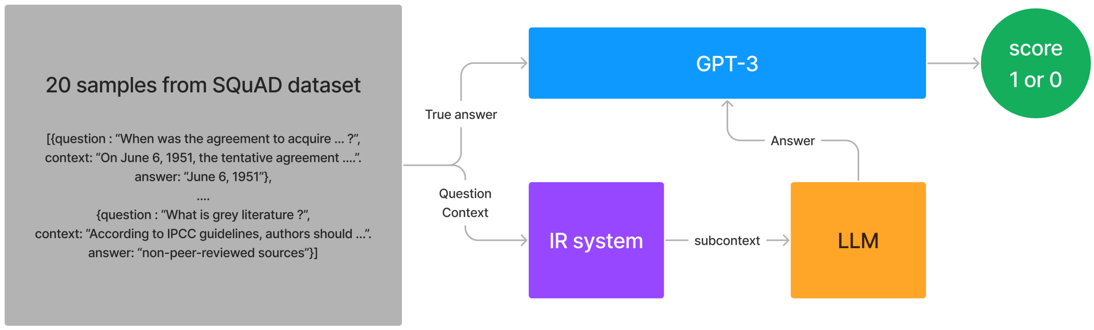

Figure 7: Evaluation schema with GPT-3 [4].

4.5 Implementation details

- general details python, git repos, machine specs.

1. python version : 3.10.11
2. system : Ubuntu 20.04.6 LTS x86_64
3. GPU : 1x NVIDIA GeForce GTX 1080 Ti and 2x NVIDIA GeForce RTX 3060
4. RAM : 126GB

IR system - splitting. - embedding.

Language models - LMs (batching).

Evaluation - evaluation (sampling from dataset) (how to use gpt-3 decoding).

User interface - backend - frontend

References

[1] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40b: an open large language model with state-of-the-art performance. 2023.

[2] Edward Beeching, Sheon Han, Nathan Lambert, Nazneen Rajani, Omar Sanseviero, Lewis Tunstall, and Thomas Wolf. Open llm leaderboard, 2023.

[3] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *PLDI '23*, 2022.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[5] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. Reading wikipedia to answer open-domain questions, 2017.

[6] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[8] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge, 2018.

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

[10] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2020.

[11] Xinyang Geng and Hao Liu. Openllama: An open reproduction of llama, May 2023.

[12] Richard W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[13] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.

[16] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[17] Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension, 2017.

[18] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering, 2020.

[19] Philipp Koehn. Europarl: A parallel corpus for statistical machine translation, 2005.

[20] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023.

[21] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019.

[22] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[23] Patrick Lewis, Pontus Stenetorp, and Sebastian Riedel. Question and answer test-train overlap in open-domain question answering datasets, 2020.

[24] Jimmy Lin. The neural hype and comparisons against weak baselines, 2018.

[25] Stephanie Lin, Jacob Hilton, and Owain Evans. Truthfulqa: Measuring how models mimic human falsehoods, 2022.

[26] Jerry Liu. Llamaindex, 11 2022.

[27] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*, 2022.

[28] OpenAI. Gpt-4 technical report, 2023.

[29] Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: Outperforming curated corpora with web data, and web data only, 2023.

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

[31] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

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

[33] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks, 2019.

[34] Adam Roberts, Colin Raffel, and Noam Shazeer. How much knowledge can you pack into the parameters of a language model?, 2020.

[35] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In Bruce W. Croft and C. J. van Rijsbergen, editors, *SIGIR '94*, pages 232–241, London, 1994. Springer London.

[36] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face, 2023.

[37] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model, 2023.

[38] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

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

[40] Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training, 2022.

[41] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers, 2020.

[42] Zhiguo Wang, Patrick Ng, Xiaofei Ma, Ramesh Nallapati, and Bing Xiang. Multi-passage bert: A globally normalized bert model for open-domain question answering, 2019.

[43] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[44] Wei Yang, Yuqing Xie, Aileen Lin, Xingyu Li, Luchen Tan, Kun Xiong, Ming Li, and Jimmy Lin. End-to-end open-domain question answering with. In *Proceedings of the 2019 Conference of the North*. Association for Computational Linguistics, 2019.

[45] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019.

5 Annexes

5.1 Code

Initializing the different LMs

```python
from abc import ABC, abstractmethod

import gc

import torch
import transformers
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    LlamaForCausalLM,
    LlamaTokenizer
)


class LM(ABC):
    @abstractmethod
    def __call__(self, requests: list[str], **kwargs: any) -> list[str]:
        pass


class Llama(LM):
    def __init__(self, model_path: str = "models/openllama/7B") -> None:
        super().__init__()
        self.model_path = model_path
        self.tokenizer = LlamaTokenizer.from_pretrained(model_path)
        self.model = LlamaForCausalLM.from_pretrained(
            model_path,
            torch_dtype=torch.float16,
            device_map="auto",
        )

    def __call__(self, requests: list[str]) -> list[str]:
        outputs = []
        for prompt in requests:
            input_ids = self.tokenizer(prompt, return_tensors="pt").input_ids
            generation_output = self.model.generate(
                input_ids=input_ids,
                max_new_tokens=30,
                temperature=0.0,
            )
            output = self.tokenizer.decode(
                generation_output[0], skip_special_tokens=True
            )
            output = output.replace(prompt, "")  # eq : return_full_sequence=False
            outputs.append(output)
            print(output)
        return outputs


class Falcon(LM):
    def __init__(self, model_path: str = "models/falcon/7B/snapshots/falcon") -> None
```

```
    :
        super().__init__()
        gc.collect()
        torch.cuda.empty_cache()
        self.model_name = model_path

    def __call__(self, requests: list[str]) -> list[str]:
        tokenizer = AutoTokenizer.from_pretrained(self.model_name)
        pipeline_ = transformers.pipeline(
            "text-generation",
            model=self.model_name,
            tokenizer=tokenizer,
            torch_dtype=torch.bfloat16,
            trust_remote_code=True,
            device_map="auto",
        )
        sequences = pipeline_(
            requests,
            max_new_tokens=30,
            do_sample=True,
            top_k=10,
            temperature=3e-4,
            num_return_sequences=1,
            eos_token_id=tokenizer.eos_token_id,
            return_full_text=False,
        )
        outputs = [seq[0]["generated_text"] for seq in sequences]
        return outputs


# Vicuna.py
import torch
from fastchat.model.chatglm_model import chatglm_generate_stream
from fastchat.model.model_adapter import get_conversation_template, load_model
from fastchat.serve.inference import generate_stream

from src.lm import LM


class SimpleChatIO:
    """this is a workaround to use fastchat for batched inference"""

    def __init__(self, requests: list[str]) -> None:
        self.requests = requests

    def prompt_for_input(self) -> str:
        return self.requests.pop(0) if self.requests else ""

    def stream_output(self, output_stream):
        pre = 0
        for outputs in output_stream:
            output_text = outputs["text"]
            output_text = output_text.strip().split(" ")
            now = len(output_text) - 1
            if now > pre:
                print(" ".join(output_text[pre:now]), end=" ", flush=True)
                pre = now
        print(" ".join(output_text[pre:]), flush=True)
```

```
        return " ".join(output_text)


def chat_loop(
    model_path: str,
    device: str,
    num_gpus: int,
    max_gpu_memory: str,
    load_8bit: bool,
    cpu_offloading: bool,
    temperature: float,
    repetition_penalty: float,
    max_new_tokens: int,
    chatio: SimpleChatIO,
    debug: bool,
):
    # Model
    model, tokenizer = load_model(
        model_path, device, num_gpus, max_gpu_memory, load_8bit, cpu_offloading,
   debug
    )
    is_chatglm = "chatglm" in str(type(model)).lower()
    is_fastchat_t5 = "t5" in str(type(model)).lower()

    # Hardcode T5 repetition penalty to be 1.2
    if is_fastchat_t5 and repetition_penalty == 1.0:
        repetition_penalty = 1.2

    while True:
        conv = get_conversation_template(model_path)  # reset conversation

        try:
            inp = chatio.prompt_for_input()
        except EOFError:
            inp = ""
        if not inp:
            print("exit...")
            break

        conv.append_message(conv.roles[0], inp)
        conv.append_message(conv.roles[1], None)

        if is_chatglm:
            generate_stream_func = chatglm_generate_stream
            prompt = conv.messages[conv.offset :]
        else:
            generate_stream_func = generate_stream
            prompt = conv.get_prompt()

        gen_params = {
            "model": model_path,
            "prompt": prompt,
            "temperature": temperature,
            "repetition_penalty": repetition_penalty,
            "max_new_tokens": max_new_tokens,
            "stop": conv.stop_str,
            "stop_token_ids": conv.stop_token_ids,
            "echo": False,
```

```
        }

        output_stream = generate_stream_func(model, tokenizer, gen_params, device)
        outputs = chatio.stream_output(output_stream)

        yield outputs

        if debug:
            print("\n", {"prompt": prompt, "outputs": outputs}, "\n")


class Vicuna(LM):
    def __init__(self, model_path: str = "models/vicuna/7B") -> None:
        super().__init__()
        self.model_path = model_path
        self.device = "cuda" if torch.cuda.is_available() else "cpu"
        self.num_gpus = 3
        self.max_gpu_memory = None
        self.load_8bit = False
        self.cpu_offloading = False
        self.max_new_tokens = 20

    def __call__(
        self,
        requests: list[str],
    ) -> list[str]:
        chatio = SimpleChatIO(requests=requests)
        outputs = [
            output
            for output in chat_loop(
                self.model_path,
                self.device,
                num_gpus=self.num_gpus,
                load_8bit=self.load_8bit,
                temperature=0.0,
                repetition_penalty=1.0,
                max_new_tokens=20,
                debug=False,
                chatio=chatio,
                cpu_offloading=self.cpu_offloading,
                conv_template=None,
                max_gpu_memory=None,
            )
        ]
        return outputs
```

Initializing the different embedding models

```
from abc import ABC, abstractmethod

import torch
import torch.nn.functional as F
from sentence_transformers import SentenceTransformer
from torch import Tensor
from transformers import AutoModel, AutoTokenizer, BertForQuestionAnswering

from src.similarity_measure import CosineSimilarity
```

```python


class Embedder(ABC):
    @abstractmethod
    def embed(self, texts: list[str]) -> list[list[float]]:
        """Embed a list of contexts"""
        pass

    def embed_query(self, texts: list[str]) -> list[list[float]]:
        """Embed a list of queries"""
        return self.embed(texts)

    def embed_context(self, texts: list[str]) -> list[list[float]]:
        """Embed a list of contexts"""
        return self.embed(texts)


class MiniLM(Embedder):
    def __init__(self) -> None:
        super().__init__()
        self.model = SentenceTransformer("all-MiniLM-L6-v2", device="cpu")

    def embed(self, texts: list[str]) -> list[list[float]]:
        embeddings = self.model.encode(texts)
        return embeddings.tolist()


class Bert(Embedder):
    def __init__(self) -> None:
        super().__init__()
        self.tokenizer = AutoTokenizer.from_pretrained("deepset/bert-base-cased-squad2")
        self.model = BertForQuestionAnswering.from_pretrained(
            "deepset/bert-base-cased-squad2"
        )

    def embed(self, texts: list[str]) -> list[list[float]]:
        inputs = [self.tokenizer(text, return_tensors="pt") for text in texts]
        with torch.no_grad():
            outputs = [
                self.model(**input, output_hidden_states=True) for input in inputs
            ]
            # take the average of the last hidden-state of each token to represent the sentence
            outputs = [
                output.hidden_states[-1].mean(dim=1).flatten().tolist()
                for output in outputs
            ]

        return outputs


class E5(Embedder):
    def __init__(self) -> None:
        super().__init__()
        self.tokenizer = AutoTokenizer.from_pretrained("intfloat/e5-small")
        self.model = AutoModel.from_pretrained("intfloat/e5-small")

```

```python
    def average_pool(
        self, last_hidden_states: Tensor, attention_mask: Tensor
    ) -> Tensor:
        last_hidden = last_hidden_states.masked_fill(
            ~attention_mask[..., None].bool(), 0.0
        )
        return last_hidden.sum(dim=1) / attention_mask.sum(dim=1)[..., None]

    def embed(self, texts: list[str]) -> list[list[float]]:
        batch_dict = self.tokenizer(texts, padding=True, return_tensors="pt")

        outputs = self.model(**batch_dict)
        embeddings = self.average_pool(
            outputs.last_hidden_state, batch_dict["attention_mask"]
        )

        # (Optionally) normalize embeddings
        embeddings = F.normalize(embeddings, p=2, dim=1)
        return embeddings.detach().tolist()

    def embed_query(self, texts: list[str]) -> list[list[float]]:
        return self.embed(["query :" + text for text in texts])

    def embed_context(self, texts: list[str]) -> list[list[float]]:
        return self.embed(["passage :" + text for text in texts])
```

Evaluation using GPT-3 [4] and the lmql library [3]

```python
import os

import dotenv
import lmql
import openai

dotenv.load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")


@lmql.query
async def evaluate_gpt3(prediction: str, ground_truth: str):
    '''argmax
        """output 1 if the student answer is similar to true answer, 0 otherwise
        ignore small differences
        student answer: {prediction}
        true answer: {ground_truth}
        evaluation : [EVALUATION]"""
    from
        "openai/text-davinci-003"
    distribution
        EVALUATION in ["1", "0"]
    '''
```

Index construction

```python
from src.embedding import Embedder
from src.text_splitter import Splitter
```

```python
from src.utils.utils import dump_pickle, read_pickle


class Index:
    def __init__(
        self,
        embedder: Embedder,
        splitter: Splitter,
    ) -> None:
        super().__init__()

        self.splitter = splitter
        self.embedder = embedder

    def __call__(self, document: str, index_path: str | None = None) -> list[str]:
        """returns an indexed document"""
        if document:
            self.index = index = self.index_document(document)
        elif index_path:
            self.index = index = self.load_index(index_path)
        else:
            raise ValueError("Either document or an index_path must be provided")

        return index

    def index_document(self, document: str) -> list[tuple]:
        chunks = self.splitter.split(document)
        embeddings = self.embedder.embed_context(chunks)
        return list(zip(chunks, embeddings))

    def save_index(self, path: str) -> None:
        assert self.index, "Index is empty"
        dump_pickle(self.index, path=path)

    def load_index(self, path: str) -> list[tuple]:
        return read_pickle(path=path)
```

5.2 Experiments