Datenanalyse im Unternehmen

Christopher Keibel
30341906
South Westphalia University of Applied Sciences

March 4, 2024

Abstract

This work is being done in the context of module *Datenanylse im Unternehmen* at the *South Westphalia University of Applied Sciences*. The goal of the project is to design a *retrieval augmented generation* pipeline in which the models for retrieving documents and those for generating answers can be easily swapped. The pipeline evaluates the individual components and their interaction as a closed system. In addition, an optional keyword retriever is provided for the retrieval of documents in order to implement and evaluate a hybrid search. At the beginning of the work, the individual components are explained in a theoretical section. This is followed by the methodology part, in which the implementation and evaluation are described.

Contents

1 Introduction

Since the arrival of *ChatGPT*, generative AI algorithms have made their way into the mainstream media and the everyday lives of many[1] people. Every day, numerous people use the chatbot *ChatGPT* developed by *OpenAI* to make everyday tasks easier. Behind its technological success is an algorithm called *Transformer* [21], an *encoder-decoder* network that can efficiently process long text passages. To be precise, *ChatGPT* utilizes only the *decoder*[14] component of the original *Transformer*.

In [14, 15], *OpenAI* shows that *language models* can learn different text-based tasks through *generative pre-training (causal language modelling)*. The *generative pre-training* is done on an unlabelled dataset and the goal of the *language model* during *language modelling* is to predict the next *token* given a sequence of *tokens* as input. In [9], *OpenAI* shows that scaling these *language models* has a significant impact on the capabilities and performance. The capabilities of these models can be improved by further a so-called *instruction fine-tuning* [23] for specific tasks and by using *reinforcement learning from human feedback* [12] to more closely match the output to human intent.

Although models trained in this way may have impressive capabilities, they still have a serious problem. The knowledge acquired during the training of these models is stored in the model parameters[7]. Therefore, it is not possible to draw conclusions about the accuracy of the generated outputs. Here we often speak of model *hallucination*, as it may generate outputs with a high confidence that may not be factually correct.

Another issue is the knowledge cutoff problem. This refers to the limitation of models to learn only the information present in their training data. Despite being trained on a vast corpus of data, their knowledge is restricted to the date on which the data was last collected. To acquire new knowledge, these models have to be re-trained at some expense.

In [7], a framework called *retrieval augmented generation (RAG)* is proposed, which is intended to solve these problems. By adding a non-parameterized memory in the of a vector database, the knowledge of the *language model* is augmented.

The workflow of this framework is defined by three main components. In addition to the generative model, another *language model* which is called *retriever*, is used to search the additional knowledge that is stored in the vector database on the basis of a query. The found information and the query are then passed to the third component, the generative *language model*, which then answers the query based on the extracted information.

Unlike online web search engines such as *Google* or *Bing*, this setup also allows users to perform searches on their own data by indexing it using a vector database.

This work discusses and evaluates the various components and the workflow of the *retrieval augmented generation* framework. The capabilities of this technology are illustrated and possible drawbacks and problems are outlined. Section 2 provides a theoretical basis for the technology, which is a prerequisite for understanding and implementing the project described in section 3.

2 Retrieval Augmented Generation (RAG)

As we have learned in section 1, the *retrieval augmented generation* framework augments the context of our *prompts* with relevant information to address the knowledge limitations of *generative models*.

[1] Reuters: *ChatGPT* sets record for fastest-growing user base (visited 04.03.2024)

In this chapter, we will take a detailed look at the components of the *RAG* framework and how they interact with each other to overcome the limitations of *generative models*.

2.1 Retrieving relevant information

A critical part of making this framework work well, is finding relevant information to a user query. If the information passed to the *generative language model* is not relevant to the query, the *generative model* will not be able to return a qualified answer. The search for relevant information therefore seems to be one of the most important components of the framework for it to be successful. In a *RAG* setup, the system receives a user query in the form of a *string* and must find relevant information from a *data store*[2] based on this *string*. Since the query is delivered in *string* form, it is not easily possible to perform a classic *SQL* database query to find the desired information. To be able to easily find and store data in the form of a *string*, we make use of *natural language processing (NLP)* algorithms to convert given texts into vectors.

Transforming texts and documents[3] into vectors and thus into a *Euclidean vector space* with same dimension helps to perform meaningful retrieval operations [18]. A *Euclidean vector space* has the property that the similarity between vectors within it can be calculated. We make use of this property in *information retrieval* to map documents and user queries in the same *Euclidean space* and calculate the similarity between them. A similarity function like *cosine similarity* indicates how close a query vector is to a document vector in the *vector space*. The closer together these vectors are, the more likely we assume that they have a similar meaning. It is therefore important to find a good method that maps similar documents into a similar *vector representation*, so that they are close to each other in the in the *vector space*.

2.1.1 Vector representations of text

There are different ways to represent texts as vectors, which can be divided into two fundamental categories, which will be used in section 3 to perform *information retrieval*. The first category are so called *sparse vectors*, which are commonly used for keyword searches. The second category are *dense vectors*. *Dense vectors* aim to encode the *semantic meaning* of a document without being limited to exact keywords.

Sparse vector representation

To represent documents as sparse vectors, extensions of a *bag-of-words* model are often used. In a *bag-of-words* model, unique words are considered independently of each other, and their respective occurrences are counted. [1, pp. 13ff]. When splitting documents into smaller units such as words or sub words, we also speak of *tokens*[4]. The dimension of the resulting vectors is equal to the number of our unique words in our entire *corpus*, which is the size of the *vocabulary* used by the *bag-of-words* model. Each component of a vector is clearly assigned to a word from the *vocabulary*. The exact values

[2] A *data store* is a collection of information that is searched by the *RAG* framework. The information can be stored in a database or even in a simple list.

[3] I will use the words document and text interchangeably in this work, both mean the same thing in the context of this work.

[4] In the context of *bag-of-words* models, I continue to write about words or terms instead of *tokens*, as these work on a word/ term basis.

of the vector components are created when a document is *vectorized*. For each *word*, the occurrence frequency in the document is counted and stored in the associated vector component as shown in fig. 1.

	Term 1	Term 2	...	Term N
Document 1	3	0	...	2
Document 2	0	5	...	1
...	...	...	...	...
Document N	2	1	...	0

Figure 1: *Bag-of-words model* - The column headers contain the unique words from our *vocabulary*. The table has a separate row for each document. Each column stores the frequency with which a term occurs in each document.

Using the resulting row vectors (document vectors) from fig. 1, we can construct our *vector space* for the search. User queries need to be *vectorized* in the same way (word counts per term over the entire vocabulary) so that we can then use *cosine similarity* to find matching documents in the *vector space*.

As can be seen in fig. 1, a small number of documents can result in *sparse vectors* (vectors with many zero components). *Sparse vectors* take up an unnecessary amount of memory. To implement a *bag-of-words* model efficiently, it is often done in the form of an *inverted index*. An *inverted index* stores a tuple consisting of a *document id* and the occurrence count for each term. If a term does not occur in a document, it is not stored for the current term.

However, simply counting the words in a document for *information retrieval* can be problematic. The vectors obtained in this way do not take into account the *importance* or *rarity* of a *word* in the entire document. Texts often consist of many words that do not contribute much to the information content of the document; these words are referred to as *stopwords*. A first extension of the *bag-of-words* model is *tf-idf*, short for *Term Frequency Inverse Document Frequency* [3, pp. 7ff]. The goal of *tf-idf* is to calculate some relevance for document with respect to a user query. *Tf-idf* adds a weighting to each word of the *vector representation* to capture the meaning of a word in a document relative to the frequency of the word in the entire corpus (all documents).

$$
\begin{aligned}
tfidf_{t,d} &= tf_{t,d} \cdot idf_{t,D} \\
&\text{where:} \\
t &= \text{Term (word,)} \\
d &= \text{document,} \\
D &= \text{corpus (all documents.)}
\end{aligned}
\tag{1}
$$

Term Frequency counts the number of times a word appears in a document, with higher occurrences being more significant. The *term frequency* is then normalised to obtain the relative frequency, considering the length of the document.

$$tf_{t,d} = \frac{\sum_{t:t\in d} 1}{\sum_{i=1}^{|d|} 1}$$

where: (2)

$t : t \in d$ = term (word) in document,

$|d|$ = length of the document (all terms).

Inverse Document Frequency counts how often the word occurs in the entire corpus.

$$idf_{t,D} = log\frac{N}{\sum_{D:t\in D} 1}$$

where: (3)

N = number of documents,

$D : t \in D$ = term (word) in document.

If a word occurs frequently in the entire corpus, it is possibly a *stopword* and must be weighted lower. *Tf-idf* considers the length of a document, as longer documents contain more words, resulting in certain words appearing more frequently than in shorter documents. If a long document and a short document have the same *Tf-idf* score, the long document is weighted lower.

One problem with *tf-idf* is that *term frequency (tf)* has a very strong influence on the resulting *relevance scores*. If a word from a user query occurs very often in two different documents from the corpus, we do not know whether the document in which it occurs more often is actually more relevant to the query in the end. *BM25 (best match 25)* is an improved ranking model that takes this into account by calculating the log of *tf*. The $\log tf$ in *BM25* rewards term frequency at low values like *tf-idf*, and as the *tf* increases, this reward flattens out.

$$BM25(Q,D) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{TF(q_i,D)\cdot(k+1)}{TF(q_{i,D}) + k\cdot(1-b+b\cdot\frac{|D|}{avgdl})}$$

where: (4)

Q = user query,

D = document,

n = number of terms in query,

q = actual term,

$|D|$ = document length,

$avdgl$ = average document length,

k and b = are hyperparameters.

Dense vector representation

In *sparse vectors* as described above, we look for an exact *term match* and describe documents by terms and how important they are to the document. However, these *sparse vectors* do not encode any *semantic characteristics*. This can quickly lead to problems if keywords are not included in a query to find the relevant document we are looking for. *Sparse vector search* relies heavily on a exact term match to find relevant documents.

To represent *semantic characteristics* without relying too heavily on keywords, we utilise machine learning models to learn and encode *dense vectors*. In order to learn these vectors through machine learning models, we make use of the *distributional semantics theory* [4]. According to the *distributional semantics theory*, words that appear in similar contexts tend to have a similar meaning.

The *Word2vec* models help to explain how the *distributional semantics theory* was used to obtain *dense vectors* encoded with *semantic meaning* [10]. *Word2vec* utilises two simple artificial neural network architectures: *Continuous bag of words (CBOW)* and the *skip-gram* model. *CBOW* takes a sentence with a masked word as input and predicts the masked word based on the context. The *skip-gram* model, on the other hand, predicts the context words given a word. Both models learn to represent words in relation to their context. However, the word vectors learned in this way have a limitation. Each word has only one representation. In *natural language*, however, the same words can have different meanings, which also depend on the context and word order. *Transformer* models, as used in this work, take advantage of the position of a word (*positional encoding*) in a sentence to better understand the meaning of the word in a given context and to allow different representations of the same word [21]. This section explains how the *encoder* part of the *transformers* learns its *embeddings*[5], which will be used for *information retrieval* later.

In order for the *encoder* to process text in string form, it is necessary to convert it into numerical values. This is achieved by *tokenizing*[6] the documents at the start. *Tokenization* is carried out by a *tokenizer* that learns how to split texts. The rules that the *tokenizer* learns to split the text determine the *vocabulary*. Texts are often divided into *sub-words*. In the original *BERT* paper, *WordPiece* was used for *tokenization*, which generated a *vocabulary* of 30,000 *tokens* [6]. Each *token* is assigned a unique, nurmeric id. After *tokenization*, the resulting ids are transformed into vectors using an *embedding layer* [20, p. 65]. This layer acts as a *lookup table*, assigning a vector to each token id. The *embedding layer* is essentially a matrix with dimensions equal to the *vocabulary size* and *target size* of our vectors. Next, *positional encoding* is added, a mechanism to better understand the word order of the input. There are different ways to add *positional encoding* to the *embedded vectors*. The original *transformer* used *sine* and *cosine functions* of different frequencies. To add *positional encodings* to the *embedded vectors*, we create for each input word another vector of the same dimension as the *embedded vectors*. The vector components are defined by a formula (5) that takes into account the alternation of *sine* and *cosine curves*. Each component is represented by a separate *sinusoid*, with even-indexed components represented by a *sine function* and odd-indexed components represented by a *cosine function*. The position of the word in the input determines the part of the *sinusoid* from which the values are taken.

[5] The term *embeddings* or *word embeddings* is frequently used when referring to data that has been encoded in lower-dimensional *vector space*.

[6] *Tokenization* is the process of splitting text into smaller units called *tokens*.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d}) \tag{5}$$
where d = dimension of embedded vector.

The vectors generated represent the *position* of the words in the input relative to the rest of the input. These vectors are then added to the *embedded vectors*.

In [21], the generated vectors are fed into a *multi-head attention* block comprising several *self-attention mechanisms*. The *self-attention mechanism* involves creating three representations of each *embedded vector* by passing them through a *linear layer*. These *triplets* are referred to as *Key (K)*, *Query (Q)*, and *Value (V)*. The *attention weights* are then calculated using the *scaled dot-product* formula based on this new representation. The *scaled dot-product* formula takes the *dot-product* of the *query* and *key* representations, producing *attention scores*, which are then scaled by the dimension of the vectors to avoid large values during training [20, pp. 62ff]. Finally, the *scaled attention scores* are normalised using a *softmax function*. The *attention weights* obtained are then multiplied by the corresponding *value representation* of the *triplet*.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d}})V \tag{6}$$
where d = dimension of vector.

This process defines a single *attention head*. The advantage of *multiple attention heads* is, to be able to focus on various *semantic characteristics* of the input. The mechanism is comparable to that of *convolutional neural networks*, where the filters learn to detect different types of edges in images.

When multiple *attention heads* are used, their outputs are concatenated so that we again obtain a single representation. To prevent this from becoming too large, the output dimension of the *attention heads* can be a factor of the dimension of the input vectors. *BERT*, for example, uses 12 *attention heads* with a vector dimension of 768. The partial output dimensions of the *attention heads* then have the dimension $768/12 = 64$ [21].

After our *embeddings* have been processed by the *multi-head attention* Block, we receive so-called *contextualised embeddings* for each *token* of the input sequence. The *contextualized embeddings* are then processed independently by a normal *feed forward network*.

The output is an *embedded vector* for each *token* in the input sequence. As the input sequences can vary in length, the output will have a corresponding number of vectors. In order to be able to perform the search in *vector space*, however, we want to represent our input (documents and queries) with just a single vector. To obtain a single vector representation, a pooling operation, such as *mean pooling*, can be performed over the vectors. This results in single values for each dimension in the final vector representation.

BERT is a model that is suitable for representing natural language. *BERT* is often extended with task-specific heads, e.g. for *classification* or *sentiment analysis*. Another use case is to compare a pair of inputs for *similarity*, as required in *information retrieval*. To achieve this, both inputs must be fed into the model simultaneously, allowing the *attention heads* to work on them in parallel (*cross-attention*). However, this process has proven to be computationally intensive [17]. In [17], an

architectural modification called *SBERT (Sentence BERT)* is proposed that reduces this computational overhead while maintaining the same accuracy. *SBERT* uses a *siamese network structure*, in which weights are shared to generate input *embeddings* separately. These embedded inputs can then be compared using *cosine similarity*.

In addition, *SBERT* was *fine-tuned* with a *multiple negative ranking loss*. It ensures that the *vector representations* are optimized for *semantic similarity*. With the *multiple negative ranking loss*, we have a triplets consisting of three input sentences. One is called the *anchor*. In addition, we have a similar sentence (*positive sentence*) to the *anchor* and an opposite sentence (*negative sentence*) to the anchor. The objective of the *multiple negative ranking loss* is to produce *vector representations* that bring the *anchor* and the *positive sentence* closer together in the *vector space* while pushing the *negative sentence* further away.

2.1.2 Retrieving relevant document vectors

In section 2.1.1 we learnt how to convert vectors into meaningful representations in order to search for them in a *vector space*. In this section, we will look at how to retrieve relevant document vectors that are close to a query vector, to achieve satisfactory results.

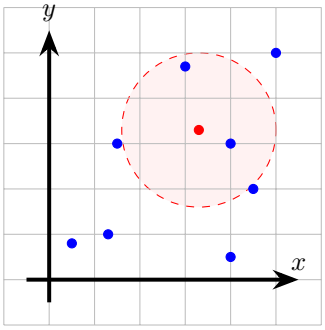

Figure 2: Here we can see a simplified 2-dimensional search space. The blue dots represent our documents, the red dot represents a user query. The goal of a search in the *vector space* is to find documents that are close to a user query.

In fig. 2 we see a simplified representation of *vector search* in a 2-dimensional *vector space*. The blue dots represent *vectorized* documents and the red dot represents a user query. The goal of a *vector search* is to retrieve nearby documents, shown here as a red search radius.

Distance measures

There are several different metrics to compute a score for vector similarity. *Chroma*, the *vector store* used in this project, offers three different metrics: *cosine similarity*, the *inner product* of two vectors and the *euclidean norm* (ℓ^2*-norm*). In order to use and evaluate these correctly, it is important to understand how they work.

To calculate the ***cosine similarity*** between two given vectors $\vec{a}$ and $\vec{b}$, we calculate the angle θ between $\vec{a}$ and $\vec{b}$ as we can see in fig. 3. The cosine of θ is then calculated to obtain an *similarity score* for the two vectors $\vec{a}$ and $\vec{b}$.

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{||\vec{a}|| \cdot ||\vec{b}||} \tag{7}$$

Cosine has the property that the cosine of equal vectors is 1 and 0 if they are orthogonal [19]. Therefore, if we want to obtain matching documents for our query, we look for *cosine similarity scores* that are close to 1.

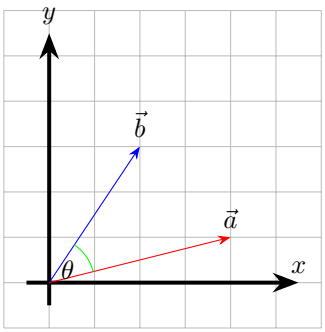

Figure 3: In this figure we have the two vectors $\vec{a}$ (red) and $\vec{b}$ (blue). Between them we have the angle θ (green). We calculate the cosine of θ to get our *similarity score* of $\vec{a}$ and $\vec{b}$.

To illustrate this in an example, let's look at the three sentences *"Hello World"*, *"Hello Mars"* and *"Transformers are awesome"* and calculate the *similarity score* between them using *cosine similarity*. To obtain vectors for these three sentences, we embed them with an *encoder transformer model*. We then use the *cosine similarity* formula to calculate the *similarity score* between them. As we can see in fig. 4, the *cosine similarity* for identical sentences is 1, for similar sentences it is less than 1 and the more dissimilar they are, the closer the score is to 0.

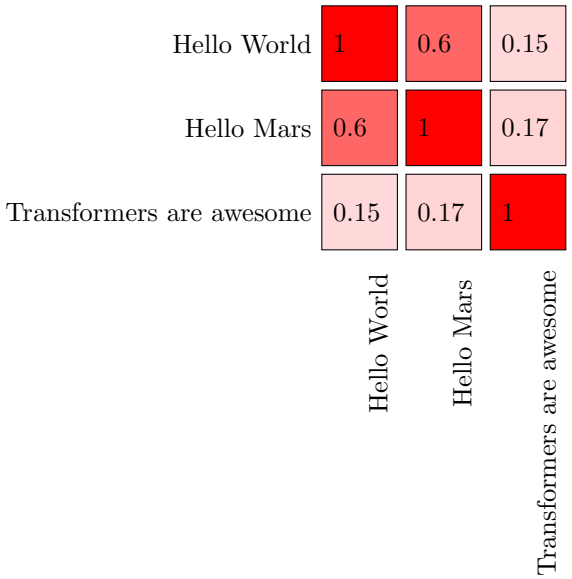

Figure 4: In this *heatmap (confusion matrix)*, we can see that identical sentences receive a score of 1. If the sentences are slightly different, such as *"Hello World"* and *"Hello Mars"*, the score decreases. The less similar the sentences are, the closer the score gets to 0. So we know that *cosine similarity scores* can be between 1 and 0.

The ***inner product*** of two vectors $\vec{a}$ and $\vec{b}$ is a generalised form of the *dot product*. It is calculated by multiplying the components of vectors $\vec{a}$ and $\vec{b}$ and then summing the result.

$$ip = \sum_{i=1}^{n} a_i \cdot b_i, \text{ with } a_i \in \vec{a} \text{ and } b_i \in \vec{b} \tag{8}$$

fig. 5 demonstrates how scores are derived from the *inner product* geometrically. In fig. 5, the similarity with vector $\vec{a}$ is calculated using vectors $\vec{b}$ and $\vec{c}$. The vectors $\vec{b}$ and $\vec{c}$ are projected onto vector $\vec{a}$, and the length of the resulting projection determines the *similarity score* of the *inner product* [5, pp. 10ff]. It is evident that vectors that are farther apart produce smaller projections and, therefore, have a lower score. For vectors of equal length, the *inner product* results in identical values as the *cosine similarity*.

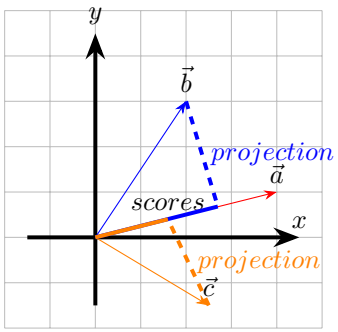

Figure 5: This figure shows geometrically how the *inner product similarity scores* are calculated. The similarities between vectors $\vec{a}$ and $\vec{b}$ and vectors $\vec{a}$ and $\vec{c}$ are calculated. Vectors $\vec{b}$ and $\vec{c}$ are projected onto $\vec{a}$. The length of the respective projection on the vector $\vec{a}$ then determines the *inner product similarity score*.

The ***Euclidean distance***, or *Euclidean norm (ℓ^2-norm)*, can be used to measure the distance between vectors in a *vector space*. To calculate the *Euclidean distance* between the vectors $\vec{a}$ and $\vec{b}$, the length of a vector $\vec{c}$ is calculated. Vector $\vec{c}$ starts from vector $\vec{a}$ and points to the end of vector $\vec{b}$ as shown in fig. 6. This gives the following formula:

$$d = \sqrt{\sum_{i=1}^{n} a_i - b_i}, \text{ with } a_i \in \vec{a} \text{ and } b_i \in \vec{b}. \tag{9}$$

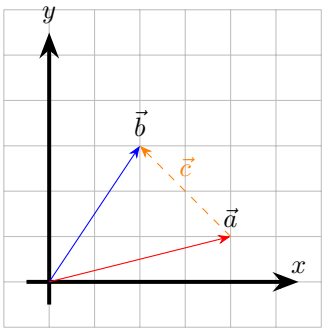

Figure 6: The *Euclidean distance* measures the distance between two vectors $\vec{a}$ and $\vec{b}$. The distance can be represented by a vector $\vec{c}$ whose length must be calculated to determine the distance between $\vec{a}$ and $\vec{b}$.

Unlike *cosine similarity* or the *inner product*, smaller values for *Euclidean distance* mean a better *similarity score* because it measures the actual distance between two vectors.

2.2 Generating text

After our *retriever* (model that embeds documents and queries) has retrieved relevant documents with the help of a *similarity function*, it is now time to generate text based on these documents that answers the user query. This is done by another *language model*. The model receives the documents and the user query in a structured (*prompt*) form in order to answer the question. The *language model* uses the *decoder* architecture of the original *transformer* [14][7] and, as indicated in section 1, the goal of the *language model* is to predict the next most probable *token* given the input sequence.

$$p(x) = \Pi_{i=1}^{n} p(s_n | s_1, ..., s_{n-1}) \tag{10}$$

This *language model* can also be seen as a document completion engine, it receives a sequence of tokens as input and generates new tokens until it predicts an *end-of-sequence token (EOS)*. This is an interative task because until the model predicts the *eos-token*, all previous generated *tokens* are appended to the input sequence and the next *token* in the sequence is predicted. Through special *task-specific fine-tuning*, the model learns a *probability distribution* over the *token vocabulary* that matches more likely our task.

3 Methodology

This section of the report describes the practical part of the project. As presented in section 2, several components work together to implement a complete *RAG system*. Only the most important parts of the implemented code are described.

3.1 Components

Python classes have been implemented for the *retrievers (encoder and bm25)* and for the *causal language model (decoder)*. For components working on the basis of a neural algorithm, a `__call__` method was implemented to allow simple inference calls. In addition, a separate *vector store* has been implemented from scratch to make *hybrid search* easier to handle.

3.1.1 Encoder and BM25/ Retriever

Encoder

The `EncoderModel` class is responsible for *embedding* and *searching* in the *dense vector space*. It is initialized with a `model_id` and an optionally `device` parameter which determines on what *device* (`cpu` or `cuda:<Index>`) the model should work on.

```
class EncoderModel:
    def __init__(self, model_id, device=None):
        self.model_id = model_id
        fallback_device = "cuda" if torch.cuda.is_available() else device
        self.device = device if device else fallback_device
```

[7] This does not always have to be the case. *T5* [16] from *Google*, for example, is a complete *sequence-to-sequence transfomer*.

The `__call__` function implements the inference call. The input document is *tokenized* and the model creates the *token embeddings*. Mean pooling is performed on the *token embeddings* to obtain the *document embeddings*. The *document embeddings* are then *normalized*.

```
def __call__(self, document):
    tokenized_document = self.tokenize(document)
    outputs = self.model_inference(tokenized_document)
    embeddings = self.mean_pooling(
        outputs, tokenized_document["attention_mask"]
    )
    torch.cuda.empty_cache()
    return F.normalize(embeddings, p=2, dim=1).cpu().numpy()
```

BM25

The `BM25Model` uses the `BM25Okapi` algorithm from the `rank_bm25` package to search and embed in the *sparse vector space*. The class is initialised without specifying any parameters. The important methods of this class include the `add_documents` and `search` functions. The `add_documents` function initializes the `BM25Okapi` algorithm. Before this, the function cleans the document using a `WordNetLemmatiser` from the `nltk` package and tokenizes the document into words using the `word_tokenize` function from the `nltk` package.

```
def add_documents(self, documents):
    cleaned_docs = self.clean_documents(documents)
    self.bm25 = BM25Okapi(cleaned_docs)
```

The `search` function of the `BM25Model` class receives a query as input and *cleans* and *tokenizes* it in the same way as the `add_documents` function. The *bm25 scores* are then calculated on the *tokenized corpus* and returned.

```
def search(self, query):
    cleaned_query = self.clean_string(query)
    scores = self.bm25.get_scores(cleaned_query)
    return scores
```

3.1.2 Decoder/ Autoregressive model

The `DecoderModel` class is responsible for the *causal language model* and therefore for *generating answers*. The class is initialised via the `model_id` parameter and the optional `generation_config` and `device` parameters.

```
class DecoderModel:
    def __init__(self, model_id, generation_config=None, device=None,
        **kwargs):
        assert isinstance(
        generation_config,
        GenerationConfig) or generation_config is None
```

```
        self.model_id = model_id
        self.device = device if device else "auto"
        ... # Initialize model and tokenizer
        self.tokenizer.chat_template = self.get_jinja_tempalte()
        self.generation_config=generation_config if generation_config
            else self.default_generation_config()
```

The `tokenizer` is applied a *jinja-template* in the *constructor*, which configures the *prompt structure* for the model to perform the task. After applying the *template*, *prompts* will be structured like this:

```
"""
<|system|>
Use the following pieces of context to answer the question at the end.
If you can not answer the question from the given context, just say so.
Don't make any things up.
<|context|>
{context}
<|question|>
{question}
<|assistant|>
"""
```

NOTE: In a first experiment, the model was provided with explicit instructions on how to respond if it was unable to generate an answer based on the given context. It was observed that the models struggled to replicate this exact phrasing and made grammatical errors. The prompt template was adjusted to allow the model more freedom in its generated answers.

During initialization, a default `generation_configuration` is also created. The number of new *tokens* to be generated is limited to `150` .The parameter `no_repeat_ngram_size` is used to ensure that no identical *strings* of three words are repeated. This is particularly important as *large language models* tend to repeat themselves.

```
def default_generation_config(self):
    gen_cfg = GenerationConfig.from_pretrained(self.model_id)
    gen_cfg.max_new_tokens = 150
    gen_cfg.pad_token_id = self.tokenizer.pad_token_id
    gen_cfg.begin_suppress_tokensrepetition_penalty = 5
    gen_cfg.no_repeat_ngram_size = 3
    return gen_cfg
```

The function responsible for generating answers is `__call__` . It takes a *question* and a *list* of *contexts* as input and applies the *prompt structure* as described above. The received *prompt* is *tokenized* using the `tokenizer` and then fed into the *language model* to generate an answer. The generated answer is saved in the `answer` variable, which includes the *prompt tokens* and up to 150 *new tokens* (as configured). To *decode* the generated answer, only the *newly generated tokens* are passed to the `tokenizer` by `outputs[0][len(tokenized_prompt.input_ids[0]):]` .

```
def __call__(self, question, context):
```

```
    prompt = self.construct_rag_prompt(question, context)
    tokenized_prompt = self.tokenize(prompt)
    outputs = self.model.generate(
        **tokenized_prompt,
        generation_config=self.generation_config
    )
    answer = self.tokenizer.decode(
        outputs[0][len(tokenized_prompt.input_ids[0]):]
    )
    return answer
```

3.1.3 Vector store

The `VectorStore` class is initialized with a `model_id` for the *encoder*. The *retriever models* are not located as components in the *inference loop* but are called via the `VectorStore` class. In addition to the `model_id`, there are also *optional* parameters for `hybrid`, `weight` and `distance_metric`. The parameters `hybrid` and `weight` relate to the *hybrid search*. The parameter `hybrid` is a *boolean* value that determines whether the `VectorStore` should enable *hybrid search*, `weight` determines how much each *search score* should be considered if `hybrid` is set to `True`. The `distance_metric` determines which metric should be used for the search in the *vector space*; if none is specified, the *cosine similarity* is used by default. Possible values for `distance_metric` are `cosine`, `l2` and `ip`.

The *documents* are stored using a `pandas.DataFrame` with columns *id*, *text* and *vector*.

```
class VectorStore:
    def __init__(self, model_id, hybrid=False, weight=0.5,
            distance_metric=None):
        self.encoder = EncoderModel(model_id)
        self.bm25 = BM25Model() if hybrid else None
        self.documents = pd.DataFrame(columns=["id", "text", "vector"])
        self.distance_metric = distance_metric
        self.weight = weight
```

The function `search` combines the search functions for the two search paradigms which return the `top_n` best scores for each search. The scores of both searches are *normalized* to ensure that they behave the same when combined.

The scores from both searches are combined using the following formula.

$$score_{hybrid} = (1 - weight) \cdot score_{sparse} + weight \cdot score_{dense}$$

The *sparse search* is only executed if the *hybrid search* is activated. The `search` function returns the result as a list of *dictionaries* sorted in descending order by best score. Each *dictionary* contains the respective *score*, as well as the *document text* and the *id*.

```
def search(self, query, top_n=10):
    vectorized_query = self.encoder(query)
    dense_results = self.dense_search(list(vectorized_query), top_n)
```

```
    sparse_results = None
    if self.bm25:
        sparse_results = self.sparse_search(query, top_n)
    hybrid_results = self.merge_results(dense_results, sparse_results)
    result = [
        {
            "score": score,
            "document": self.documents.text.values[key],
            "id": self.documents.id.values[key]
        } for key, score in hybrid_results.items()
    ]
    return sorted(result, key=lambda x: x["score"], reverse=True)[:top_n]
```

It is **important** to note that the calculation for score using ℓ^2 has been adjusted. As mentioned in section 2.1.2, ℓ^2 measures the distance between vectors, so smaller values are preferable to larger ones. The scores have been *normalized* and then subtracted from 1 so that the ℓ^2 *metric* behaves like the other *search metrics*, where higher scores are better.

3.1.4 Inference loop

The *inference loop* enables components to interact through *nesting*. The two outer loops iterate through lists of model ids for *decoder* and *encoder* models to be tested. Another loop tests the *hybrid* and *dense search* for each model combination, followed by iterating through the entire dataset. The search is performed for each *data point* in the dataset using each distance metric. The highest-scoring search result (measured by *ndcg*[8]) from the various distance metrics is then inputted into the *causal language model* to generate an answer.

```
for causal_id in causal_models:
    # Initialize causal lm
    ...
    for retriever_id in retriever_models:
        # Initialize VectorStore with retrievers
        ...
            for hybrid in [True, False]:
                # differentiate between hybrid and dense search
                ...
                for row in dataset.iterrows():
                    # iterate through the dataset
                    ...
                    for distance_metric in ["cosine", "ip", "l2"]:
                        # differentiate between distance metrics
                        # retrieve document with distance
                        # generate answer from best found contexts
```

[8]NDCG is described in section 3.2.3.

3.2 Results

3.2.1 Dataset

SQuADv2 *Squadv2*[9] is a dataset that extends *SQuADv1*, which comprises 100,000 data points containing an *id*, a *title*, a *context*, a *question*, and *answers*. *SQuADv2* adds 50,000 examples without *answers*. *SQuADv1* is appropriate for testing a system's ability to answer questions based on a given *context*. *SQuADv2* requires the system to determine whether the question can be answered based on the provided context. It is important to note that the *answers* provided only consist of bullet points/text snippets. However, the task in this work is to generate *full-text answers* with a *causal language model*. This makes it difficult to evaluate the model based on the answers provided.

Machine Learning by Zhi-Hua Zhou (Book) The book *Machine Learning* by *Zhi-Hua Zhou*[10] [25] is used as a second dataset. The book was read using the *Python package* `unstructured` and split into semantic chunks using and `langchain-experimental`. The `SemanticChunker`[11] uses a *embedding model* to calculate similarity between consecutive sentences.

```
# read file
elements = partition(str(file.resolve()))
    # chunk by title
    text_elements = chunk_by_title(elements)
    chunks = []

    for element in text_elements:
        element.apply(
            partial(
                clean,
                bullets=True,
                extra_whitespace=True,
                dashes=True,
                trailing_punctuation=True
            )
        )
        chunks.append(element.text)
        ...
text_splitter = SemanticChunker(model)
docs = text_splitter.create_documents([book])
```

The sections for the *table of contents* and *index* were sorted out, resulting in 2513 chunks for the *RAG system* to work on.

The *RAG system* was evaluated by answering 28 *machine learning* questions related to the books context. A list of these questions can be found in appendix A.

[9] SQuADv2 on huggingface (visited 04.03.2024)
[10] The book can be found online in the FH SWF library. (visited 04.03.2024)
[11] Langchain documentation and youtube presentation of the idea of semantic chunking. (visited 04.03.2024)

3.2.2 Models

Each combination of the models listed below was tested. In addition, each combination was tested in a *hybrid setup* with a *score weight* of 0.5 for each.

Retriever

The selected *embedding models* were taken from *MTEB (Massive Text Embedding Benchmark)*[12]. *MTEB* [11] is a benchmark for measuring text embeddings models in various tasks. All models used are open source.

"sentence-transformers/all-MiniLM-L6-v2"
"BAAI/bge-base-en-v1.5"
"WhereIsAI/UAE-Large-V1"
"BAAI/bge-m3"

Causal models

The *causal models* used are all *instruction fine-tuned* models, identifiable by their suffix (*it* or *instruct*). The list of *causal models* include the open source models from *Mistral.AI*, the `mistralai/Mistral-7B-Instruct-v0.2` and the `mistralai/Mixtral-8x7B-Instruct-v0.1`. The `HuggingFaceH4/zephyr-7b-beta` is a variant of `mistralai/Mistral-7B-v0.1` that has been further *fine-tuned* by *HuggingFace* and therefore has the same architecture. In contrast to the other models, `mistralai/Mixtral-8x7B-Instruct-v0.1` is a significantly larger model with 46.7 billion parameters[13].

In addition to the model variants from *Mistral*, the new open source models from *Google* called *Gemma* were also tested. There are two variants, one model with approx. 2 billion parameters and another with 7 billion. A *7B Zephyr* variant of *Gemma* fine-tuned by *HuggingFace* was also tested.

The *causal language models* were all loaded in `torch.float16` format.

"HuggingFaceH4/zephyr-7b-beta"
"mistralai/Mistral-7B-Instruct-v0.2"
"HuggingFaceH4/zephyr-7b-gemma-v0.1"
"mistralai/Mixtral-8x7B-Instruct-v0.1"
"google/gemma-2b-it"
"google/gemma-7b-it"

NOTE: Model `google/gemma-7b-it` was excluded from the evaluation because it only generated the sequence `<pad><pad><pad><eos>` for all pairs. However, the small variant `google/gemma-2b-it` was able to generate good answers.

[12]HuggingFace MTEB (visited 04.03.2024)
[13]Mixtral of experts (visited 04.03.2024)

3.2.3 Metrics

First of all, it must be said that evaluating an entire *RAG system* is difficult. This is because evaluating the responses of a *causal language model* alone is difficult, as the answers generated can vary but may have the same meaning. This means that there is no single correct answer. Another problem with evaluating the *causal language model* is that it relies on the performance of the *retriever*, otherwise incorrect contexts will be retrieved and the model will not be able to answer the user's queries.

We use different metrics to measure and evaluate the performance of the *language models*. For the *retriever* model, we measure its performance using the *normalized discounted cumulative gain (NDCG)*. As described, since the evaluation of *causal language models* proves to be difficult, several metrics come into play. For the *causal model*, *BLEU*, *ROUGE*, different versions of *Recall* and *Precision*, *F1* and *Judging LLM-as-a-Judge* are used for evaluation.

Normalized Discounted Cumulative Gain (NDCG) The *NDCG* is a metric for measuring rankings in which relevant documents are found and suggested [22]. In contrast to other ranking metrics, the *NDCG* enables us to give documents some kind of a continuous *relevance score* based on query, whereas other metrics only allow binary values for relevance.

The *NDCG* puts the resulting ranking in relation to the ideal ranking.

$$NDCG = \frac{DCG}{IDCG} \tag{11}$$

with

$$DCG = \sum_{i=1}^{k} \frac{rel_i}{\log_2(i+1)}$$

where:

$$i = \text{ranking index},$$
$$rel = \text{relevance score}.$$

and *IDCG* is same as *DCG* (resulting ranking order) but with optimal ranking order.

BLEU (Bilingual Evaluation Understudy) *BLEU* is a metric that is frequently used in *machine translation* [13]. Similar to generating text, as a *RAG system* does, *machine translation* also generates free text, but in a more dependent context.

BLEU compares the quality of a translation by searching for identical *n-grams* to evaluate the correspondence between the *machine translation* and a *human translation*.

$$
\begin{aligned}
BLEU &= BP \cdot \exp(\sum_{n=1}^{N} w_n \cdot \log p_n) \\
&\text{where:} \\
BP &= \text{brevity of the translation compared to the reference,} \\
p &= \text{precision value for n-grams is,} \\
N &= \text{the maximum n-gram size.)}
\end{aligned}
\tag{12}
$$

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) The *ROUGE* metric is mainly used to evaluate machine-generated summaries [8]. In contrast to *BLEU*, which is often used for the evaluation of *translations*, *ROUGE* focuses on assessing the quality of *text summaries*.

It takes into account aspects such as the overlap of words, sentences and phrases between the summary and the references. *ROUGE* has different variants such as *ROUGE-N* (for word match) and *ROUGE-L* (for longest common subsequence).

Precision *Precision* measures the ratio of correctly predicted positive instances to the total number of predicted positive instances. It is a measure of the model's *accuracy* in predicting positive instances.

$$
Precision = \frac{\text{True positives}}{\text{True positives} + \text{False positives}} \tag{13}
$$

Recall *Recall* measures the ratio of correctly predicted positive instances to the total number of actual positive instances. It indicates how many of the actual positive instances the model has correctly identified.

$$
Recall = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}} \tag{14}
$$

F1 The *F1 score* is a metric that balances *precision* and *recall*. The *F1 score* is calculated as a *harmonic mean* between *precision* and *recall* and takes into account both *false positive* and *false negative* predictions.

$$
F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{15}
$$

Judging LLM-as-a-Judge [24] describes a modern approach that uses a *large language model* to evaluate generated answers based on context, similar to human evaluation. It is crucial to use a language model with greater capacity for evaluation, which is why *GPT-3.5-turbo*[14] was used in this work. Only *answers* generated in combination with the best *retriever model* were evaluated.

[14]GPT-3.5-turbo documentation (visited 04.03.2024)

3.2.4 Result - SQuADv2

Retriever

Figure 7 shows the *NDCG* scores for the *SQuADv2* dataset. Each combination was tested on 500 examples. The examples were divided into 400 examples with questions that could be answered using a given context and 100 examples whose questions could not be answered using the context provided.

It is noteworthy that the *sentence-transformers/all-MiniLM-L6-v2*, the smallest of the models used, achieved the best scores. The *hybrid search* consistently improved the results for all combinations. It is peculiar to observe that the different distance metrics per model produced identical results. All models perform best when *cosine similarity* is used as the distance metric.

model	cosine	cosine hybrid	ip	ip hybrid	ℓ^2	ℓ^2 hybrid
sentence-transformers/all-MiniLM-L6-v2	0.823	**0.874**	0.823	**0.874**	0.823	**0.874**
WhereIsAI/UAE-Large-V1	**0.857**	0.870	**0.857**	0.870	**0.857**	0.870
BAAI/bge-m3	0.796	0.859	0.796	0.859	0.796	0.859
BAAI/bge-base-en-v1.5	0.856	0.870	0.856	0.870	0.856	0.870

Figure 7: *NDCG* results rounded to three decimal places.

Decoder

In fig. 8, the results for the metrics *ROUGE*, *Precision*, *Recall* and *F1* of the *causal language models* are presented. As previously mentioned, evaluating free text can be challenging, especially since the answers provided by *SQuADv2* are only individual words and not complete sentences. Therefore, the results do not provide much insight into the correctness and quality of the answers. It is evident that, for the most part, the small variant of the *gemma* models performed the best.

NOTE: It is worth noting that BLEU is not included in this evaluation as all models scored towards 0.

model	r1 precision	r1 recall	r1 f1	rL precision	rL recall	rL f1
mistralai/Mistral-7B-Instruct-v0.2	0.068	0.510	0.107	0.0631	0.477	0.092
mistralai/Mixtral-8x7B-Instruct-v0.1	0.0765	0.522	0.121	0.070	0.484	0.111
HuggingFaceH4/zephyr-7b-beta	0.041	**0.559**	0.071	0.037	**0.527**	0.065
google/gemma-2b-it	**0.090**	0.394	**0.133**	**0.080**	0.364	**0.119**
HuggingFaceH4/zephyr-7b-gemma-v0.1	0.028	0.475	0.050	0.026	0.454	0.046

Figure 8: Here, *r1* and *rL* represent *ROUGE-1* and *ROUGE-L*, respectively. (Results were rounded to three decimal places.)

Figure 9 shows the results for *Judging LLM-as-a-Judge*. This metric provides a better indication of the quality and accuracy of the answers generated. Unlike in fig. 8, *HuggingFaceH4-zephyr-7b-beta* performs the best in this scenario. It is worth noting that the largest model, *mistralai-Mixtral-8x7B-Instruct-v0.1*, only ranks third and is not significantly better than the smallest model *google-gemma-2b-it*.

model	correct	wrong	%
mistralai-Mistral-7B-Instruct-v0.2	400	100	80.0
mistralai-Mixtral-8x7B-Instruct-v0.1	396	104	79.2
HuggingFaceH4-zephyr-7b-beta	**427**	73	**85.4**
google-gemma-2b-it	387	113	77.4
HuggingFaceH4-zephyr-7b-gemma-v0.1	337	163	67.4

Figure 9: Results from *Judging LLM-as-a-Judge* on *SQuADv2*.

3.2.5 Result - Machine Learning Book

The complete raw pairs from *context*, *question* and *answer* can be found in LLM−as−a−judge−ML−BOOK.xlsx. A combination of the best *retriever model (sentence-transformers/all-MiniLM-L6-v)* and all *causal language models*, except for google/gemma−7b−it and mistralai/Mixtral−8x7B−Instruct−v0.1 (due to its size) were used to process the dataset. Overall, it can be said that all *causal language models* were able to answer most of the questions satisfactorily. Like section 3.2.4, the *Mistral.Ai* models architecture perform best.

model	correct	wrong	%
mistralai/Mistral-7B-Instruct-v0.2	**25**	2	**92.59**
HuggingFaceH4/zephyr-7b-beta	**25**	2	**92.59**
google/gemma-2b-it	21	6	77.77
HuggingFaceH4/zephyr-7b-gemma-v0.1	23	4	85.18

Figure 10: Results from *Judging LLM-as-a-Judge* on *Machine Learning Book* dataset.

No evaluations with metrics *ROUGE*, *BLEU*, *Precision*, *Recall* and *F1* are available for this experiment as the dataset does not contain any reference answers.

3.2.6 Conclusion

RAG appears to be a promising alternative to address the limitations of *large causal language models*. However, as is often the case in *machine learning*, it is important to ensure the cleanliness of the data, as demonstrated in section 3.2.4.

The *retrievers* have performed exceptionally well, particularly in the *hybrid* variant. It would be interesting to compare their performance with that of *commercial embeddings*. To improve results, it may be possible to *fine-tune* the *retriever models* on the given data.

Training the *generative model* can also lead to better results, but it should be noted that the whole system is dependent on the good results of the *retriever*.

Another component that was not considered here is a *neural reranker* that uses *cross attention* to calculate a similarity between the question and the found contexts and reranks the results.

One problem with the evaluation of *generative language models* is the difficulty in assessing the correctness and semantic similarity of the generated text. Using another *large causal language model* as a *judge* seems to work well, but caution is advised, as *large language models* can generate unpredictable results and are therefore not always dependable as an evaluation system.

Another approach that follows this concept is *RAGAS (Automated Evaluation of Retrieval Augmented Generation)* [2]. [2] also uses a *large language model* to evaluate results. The evaluation provides several values that determine the quality of the generated answers.

A Appendix List of questions

1. What is overfitting?
2. What is underfitting?
3. How do we test a models generalization error?
4. Why should training data points not be in the test set?
5. What is cross validation?
6. What are commonly used metrics to measure the performance of a model?
7. What is linear regression?
8. What are problems of linear models?
9. What is a decision tree?
10. What is the McCulloch–Pitts model?
11. What is a neural network?
12. How is a neural network optimized?
13. How does backpropagation work?
14. What is deep learning?
15. What is ensemble learning?
16. What is the goal of ensemble methods?
17. What is supervised learning?
18. What is unsupervised learning?
19. What is the difference between supervised and unsupervised learning?
20. What is the goal of clustering?
21. How is k-Nearest Neighbor trained?
22. Which algorithm can be used to reduce dimensions?
23. What is semi-supervised learning?
24. What is reinforcement learning?
25. What is an Markov Decision Process used for in reinforcement learning
26. How interacts an reinforcment learning agent with its environment?
27. What is the goal of an agent in reinforcement learning?
28. What is the Exploration-Exploitation dilemma?

References

[1] Jacob Eisenstein. *Introduction to natural language processing*. 2019.

[2] Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. Ragas: Automated evaluation of retrieval augmented generation, 2023.

[3] Ayse Goker and John Davies. *Information retrieval: Searching in the 21st century*. John Wiley & Sons, 2009.

[4] Zellig S. Harris. Distributional structure. *WORD*, 10(2-3):146–162, 1954.

[5] Nathaniel Johnston. *Introduction to linear and matrix algebra*. Springer Nature, 2021.

[6] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2, 2019.

[7] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[8] C Lin. Recall-oriented understudy for gisting evaluation (rouge). *Retrieved August*, 20:2005, 2005.

[9] Ben Mann, N Ryder, M Subbiah, J Kaplan, P Dhariwal, A Neelakantan, P Shyam, G Sastry, A Askell, S Agarwal, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[10] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[11] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. Mteb: Massive text embedding benchmark, 2023.

[12] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback, 2022. *URL https://arxiv. org/abs/2203.02155*, 13, 2022.

[13] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Pierre Isabelle, Eugene Charniak, and Dekang Lin, editors, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[14] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding with unsupervised learning. 2018.

[15] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[16] 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. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[17] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[18] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, nov 1975.

[19] Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[20] Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[22] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, and Tie-Yan Liu. A theoretical analysis of ndcg type ranking measures. In *Conference on learning theory*, pages 25–54. PMLR, 2013.

[23] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

[24] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36, 2024.

[25] Zhi-Hua Zhou. *Machine Learning*. Springer Singapore, Singapore, 1st ed. 2021. edition, 2021.