Final Project

Natural Language Processing
Master in Machine Learning for Health
Academic year 2024/2025

March 10, 2025

Contents

1 Introduction

Large language models (LLMs) have revolutionized artificial intelligence, as well as the way we approach many tasks in our professional and everyday lives. Their true value, however, lies in their ability to tackle real-world problems beyond conventional uses, enabling advanced solutions in areas such as customer service, fact-checking, and research assistance.

Throughout this course, we have seen how the invention of the Transformer architecture in 2017 marked a turning point in natural language processing (NLP). By introducing attention mechanisms to handle long-range dependencies, Transformers eliminated the need for sequential models such as recurrent neural networks. The original **Encoder-Decoder** Transformer architecture is particularly effective for sequence transformation tasks like machine translation and text summarization. Additionally, two specialized variants exist: **Encoder-based** models (e.g., BERT), which excel at text analysis tasks such as classification and question answering, and **Decoder-based** models (e.g., GPT), which are designed for text generation.

The shift toward large language models has been primarily driven by the expansion and adaptation of these Decoder-based models. By dramatically increasing the number of parameters and the volume of training data, these models have developed a greater capacity to capture complex linguistic patterns, general world knowledge, and diverse linguistic styles.

However, LLMs are not infallible; they can generate inaccurate or fabricated responses—a phenomenon known as *"hallucination"*. This is where **Retrieval-Augmented Generation (RAG)** systems come into play. RAG aims to mitigate hallucinations and improve response accuracy by integrating external sources of knowledge. These systems are particularly valuable when an LLM needs to provide precise answers based on a specific set of documents rather than relying solely on the general knowledge acquired during pre-training.

Objective. The objective of this project is to help students develop a practical understanding of iterative workflows with language models and RAG systems through the creation of three applications:

A. A chatbot that allows answering questions based on a document database

B. A fact-checking system

C. An academic research assistance system

Guidelines.

- Each application must meet a set of **mandatory functional requirements**.
- Students **must use a vector database** to implement the RAG system. Direct queries on sources such as Wikipedia are **not valid**.
- All systems must be **evaluated based on the specified criteria** (Table 1).
- Students may enhance their system with **additional features**, either by incorporating suggested ideas or proposing their own. While these features are optional, choosing not to implement any will cap the maximum grade at 8.
- **For the basic project, only open-source LLMs can be used.**
- In all cases, a basic frontend can be developed using tools such as Streamlit[1] to enhance the presentation. However, this will **not** be considered an NLP improvement in the context of the extension work.

Working Groups. Students will work in teams of three or four members (**7 groups of 4 members and 1 group of 3**). Each group must select one of the three designated applications by **March 12**, with a limit of three groups per application. Group selection will be managed through Aula Global.

[1] https://github.com/streamlit/streamlit

Submission. Students must submit the following:

1. A GitHub repository with the implemented code, properly commented.
2. A PDF document of **maximum two pages** that includes:
   - The link to the GitHub repository.
   - A concise description of the design decisions, including but not limited to dataset selection, technologies used, database configuration, and any additional features implemented.
   - Configuration details for both the LLM and the vector database, as well as optimization techniques (parameter tuning, prompting strategies, etc.).
   - An explanation of how the system meets the functional requirements, how hallucinations are mitigated, and any other detail that you consider makes your work noteworthy / innovative.
   - Evaluation of the system based on the criteria in Table 1. A suggestion of metrics for assessing these dimensions is provided in Section 3.5, though alternative metrics may also be used. Vague statements such as *"the system works because it provides answers formatted correctly or appears to respond appropriately"* are insufficient—specific metrics must be provided to validate performance.
   - Conclusions and limitations of the system. Omitting either of these sections will result in a penalty. These sections must be specific and directly tied to the work performed—generic statements such as *"one future direction is to reduce the system's hallucinations"* are insufficient. Instead, the report should analyze why hallucinations occur, under what circumstances, and provide concrete suggestions for improvement.
3. A presentation that will take place on **May 7.**

Criterion	**Chatbot**	**Fact-Checking System**	**Research Assistance**
RAG System Quality	Retrieves relevant information and generates coherent, useful responses.	Retrieves relevant evidence and provides precise / accurate verdicts on claim veracity	Retrieves key articles and accurately compares them with prior research.
Document Coverage	Percentage of queries answered using database information.	Ability to find sufficient evidence to verify claims.	Ability to retrieve relevant articles related to the research topic.
Response Time	Speed at which responses are generated.	Speed at which claims are verified and verdicts provided.	Speed of article retrieval and comparison generation.

Table 1: Evaluation criteria for each application.

The PDF will be the only document uploaded to Aula Global, and only one group member needs to submit it. The deadline is April 30 at 23:59.

Evaluation. The project will be assessed according to the following criteria:

- **Basic Project** (8 points)
  - Compliance with functional requirements (2)
  - Methodology (2)
  - Quality of the report (1)
  - Quality of the code (1)
  - Presentation (2)
- **Maximum 2 points** depending on the scope of the work. The grade will have a factor $2 \times \alpha$, where α will be assigned by the course instructor based on the scope of the selected work.
  - Originality (0.4)
  - Quality of the work and methodology (1.6)

2 Description of the Practical Applications

2.1 Chatbot for Answering Questions from a Document Database

A chatbot without RAG may offer imprecise answers due to its lack of access to the relevant documents, relying solely on pre-trained general information. In this exercise, we will focus on overcoming this limitation by building a chatbot with RAG.

Functional Requirements. The implemented chatbot must meet the following requirements:

- The answers must include the specific document or documents that support the response.
- If the database does not contain a relevant document, the chatbot must not provide an answer.
- The answers must be in the same language as the question. This means that responses should match the language in which the user asks the question, which should also correspond to the language of your dataset.

Below is an example of the expected interaction with the chatbot:

User:
What are the requirements for renewing the national ID (DNI) in Spain?

Chatbot:
According to the BOE document with reference BOE-A-2005-21163, the renewal must be carried out in person by the DNI holder. You can find more information here.

User:
How can one access study scholarships?

Chatbot:
I'm sorry, I don't have any information on that in the available BOE documents. Please check the BOE directly or provide more details for a more accurate search.

Database. As a document database, any collection of documents (in plain text or PDF) on which a user might be interested in asking questions can be used. Below are some examples, but students may choose other document sets that they find more interesting or relevant.

- Documents from the Official State Gazette (BOE)[2]. Here[3] you can access a collection of 216,000 documents in HTML or XML format.
- Documents from the curriculum of the Master's in Machine Learning for Health (or from other master's programs at UC3M or another university).
- User manuals and online documentation of electronic products, such as the Sony manuals database[4]. Here you can access user manuals and product guides, such as for cameras or televisions, in PDF format.

Additional Features.

- Display suggestions for similar topics. For this, a topic model can be trained on the database, and topic/question suggestions can be shown once the user asks a question.
- Add the option to generate automatic summaries from which information has been extracted, so that the user can obtain a condensed version of the retrieved content. For this, a Transformer model fine-tuned on the training corpus and/or an LLM can be used.
- Allow the chatbot to handle queries in multiple languages, translating both the questions and the answers if necessary, using pre-trained (or fine-tuned) translation models.

[2] https://www.boe.es/
[3] https://github.com/PlanTL-INTELCOMP/DBimport/tree/master/data_Law_BOE
[4] https://www.sony.com/electronics/support

2.2 Fact-Checking System

This application focuses on developing a RAG system that automatically verifies the veracity of a statement using a specific corpus. The system will retrieve relevant information from the corpus and employ an LLM to assess the truthfulness of the statement.

Functional Requirements. The system must meet the following requirements:

- Provide a clear response that explains the veracity of the statement.
- Accompany the responses with the sources and excerpts from the corpus that justify the verdict.
- If the system cannot verify the veracity of the statement due to lack of information, it must indicate so without fabricating an answer.
- The answers must be in the same language as the question. This means that responses should match the language in which the user asks the question, which should also correspond to the language of your dataset.

Below is an example of the expected interaction with the system:

User:
Climate change does not impact the frequency of extreme weather events.

System:
False. According to recent scientific studies, climate change is increasing the frequency of extreme weather events such as hurricanes, heat waves, and floods.

- Increase in temperatures: climate change is causing a rise in global temperatures, which leads to more frequent and intense heat waves [...] [source]
- Increase in temperatures: climate change is causing a rise in global temperatures, which leads to more frequent and intense heat waves [...] [source]

User:
5G technology has a direct impact on human health.

System:
I'm sorry, I do not have enough information in the corpus to verify this claim.

Database. As a source of information to corroborate the facts, any textual database containing reliable information can be used. Below are some examples, but students may choose other document sets that they find more interesting or relevant.

- A Wikipedia dump, which can be easily obtained through the `wikipedia-api` library[5].
- PubMed. A collection of scientific articles on topics related to medicine or health sciences, available through its website[6].
- COVID-19. A corpus of academic articles on COVID-19 and research related to the coronavirus, which can be easily downloaded from the following repository[7].

Additional Features.

- Include a confidence score indicating how certain the model is about the veracity of the statement, based on the evidence retrieved from the corpus.
- Compare performance with a traditional fact-checking system.
- Add the option to generate automatic summaries from which the information has been extracted, so that the user can obtain a condensed version of the retrieved content. For this, a Transformer model fine-tuned on the training corpus and/or an LLM can be used.
- Allow the system to handle queries in multiple languages, translating both the questions and the answers if necessary, using pre-trained (or fine-tuned) translation models.

[5] https://pypi.org/project/Wikipedia-API/
[6] https://ftp.ncbi.nlm.nih.gov/pubmed/baseline/
[7] https://github.com/allenai/cord19

2.3 Academic Research Assistance System

A key aspect of research is comparing current work with previous studies through a state-of-the-art review. This application aims to support researchers by allowing them to specify a research direction, after which the system will retrieve relevant articles on the topic. Additionally, it will highlight key differences between existing work and the researcher's proposed approach.

Functional Requirements.

- The system must retrieve relevant articles based on the research direction provided by the user.
- The system must be capable of identifying and displaying key differences between the user's research direction and existing work.
- If the system does not find relevant articles, it must not fabricate articles in order to provide an answer.
- The answers must be in the same language as the question. This means that responses should match the language in which the user asks the question, which should also correspond to the language of your dataset. As an exception, the titles of the articles may remain in their original language.

Below is an example of the expected interaction with the system:

Researcher:
I am researching implementations of federated neural topic models.

System:
I have found several relevant articles, including "Federated topic modeling" and "Federated non-negative matrix factorization for short texts topic modeling with mutual information". These works focus on implementing federated approaches to Bayesian topic models, such as LDA or NMF, but none provide an implementation based on neural topic models.

Researcher:
I am researching systems for assisting researchers.

System:
I'm sorry, but I do not have any additional specific articles in the current corpus on this topic. This does not imply that none exist. You may try expanding your search in other databases.

Database. As a source of information for searching academic articles, we must rely on traditional databases, such as:

- Articles published in ACL. For example, this dataset[8] contains 80 000 articles/posters dated up to September 2022.
- PubMed. A collection of scientific articles on topics related to medicine or health sciences, available through its website[9].
- arXiv. Kaggle[10] offers a dataset with more than 1.7 million abstracts and metadata of articles from arXiv.

Additional Features.

- The system can provide similarity metrics between the proposed idea and the articles found.
- Automatic state-of-the-art classification: The system can organize the retrieved articles into different categories based on their relevance, methodological approach, or contributions.
- For each retrieved article, the system can generate automatic summaries that highlight the key points. For this, a Transformer model fine-tuned on the training corpus and/or an LLM can be used.
- Allow the system to handle queries in multiple languages, translating both the questions and the answers if necessary, using pre-trained (or fine-tuned) translation models.

[8] https://paperswithcode.com/dataset/acl-anthology-corpus-with-full-text
[9] https://ftp.ncbi.nlm.nih.gov/pubmed/baseline/
[10] https://www.kaggle.com/datasets/Cornell-University/arxiv

3 Complementary Information

This section introduces some relevant concepts for the development of the practical assignment.

3.1 Prompting

Despite having very powerful pre-trained Transformer models, it is often necessary to perform fine-tuning to achieve good results in solving specific tasks. However, the general language knowledge possessed by these pre-trained models allows the fine-tuning dataset to be much smaller.

Nevertheless, it is not always possible to fine-tune, such as when we do not have access to the model parameters and can only interact with it through an API. In these cases, the concept of **"in-context learning" (ICL)** comes into play: the ability of LLMs to solve tasks using the context provided in the interaction, without needing to retrain the model or update its weights. ICL leverages examples and instructions within the message to "learn" to perform a task.

A key technique to leverage ICL is **prompting**, which consists of carefully designing the text (prompt, e.g., *"Generate a 3-paragraph text about Artificial Intelligence"*) to guide the model's response toward the desired task. Depending on the use case, prompting may sometimes be preferred over fine-tuning, and vice versa[11]. Related to this is the concept of **"prompt tuning"**, which aims to optimize prompts for specific tasks. There are two types: **"hard tuning"** and **"soft tuning"**. While in hard tuning the prompt is simply modified (e.g., by varying the examples), in soft prompt tuning a tensor (a learnable parameter) is concatenated to the input embeddings. Several ICL techniques are summarized in Table 2.

Technique	**Description**
Few-Shot Prompting	The LLM learns to complete a task from a few examples. The selection of examples in the prompt directly affects the results; aspects such as the number, order, quality of labels, and format of the examples should be considered.
Zero-Shot Prompting	No examples are used in the prompt. Techniques such as Role Prompting, Style Prompting, and System 2 Attention (S2A) are examples of Zero-Shot prompting.
Chain-of-Thought Generation	Encourages the LLM to articulate its reasoning before giving the final answer. It includes both Few-Shot and Zero-Shot CoT. The former uses an inductive phrase ("Let's think step by step"), while the latter uses examples with chains of thought. This improves performance on complex tasks.
Decomposition	Breaks down a complex problem into simpler sub-questions. Similar to Chain-of-Thought, but more focused on the problem's structure.
Ensembling	Uses different prompts to solve the same problem, and combines the responses into a final result (for example, by majority voting). This reduces the variance of the results but increases computational cost.
Self-Criticism	The LLM critiques its own responses (for example, by asking whether the result is correct) or provides feedback that is used to improve the answer.

Table 2: Summary of prompting techniques for LLMs. Each of the techniques includes several sub-techniques. For more information, see [10].

3.2 Key Parameters in Text Generation with LLMs

When working with LLMs, such as those in the GPT or LLaMA families for text generation, there are several parameters that allow us to control the generated responses. Among them we find [2, 8]:

- **Temperature.** This parameter adjusts the randomness in the model's responses. A low temperature value (e.g., 0.2) will cause the model to generate text more deterministically, choosing the most probable words. Conversely, a high temperature value (e.g., 0.7) will generate more varied and creative responses. The typical range is between 0 and 1.
- **Top-p.** Also known as nucleus sampling, this parameter defines the size of the set of tokens considered during sampling for generation. A high top-p value means that the model will take into account a larger number of possible words, including some less probable ones, thereby increasing the diversity of the generated text without introducing too much randomness. This parameter also ranges between 0 and 1.

[11] Prompting vs Fine-tuning: https://www.prompthub.us/blog/fine-tuning-vs-prompt-engineering

- **Frequency Penalty.** This parameter controls the repetition of words within the generated response. A higher penalty reduces the probability that certain tokens will appear repeatedly in the same response. It typically has a value between 0 and 2.
- **Presence Penalty.** Similar to the frequency penalty, this parameter influences the likelihood of introducing new topics or concepts in a response. A higher value increases the probability that the model will explore new terms or topics that have not yet been mentioned, reducing the chance of repeating tokens already used. This parameter also typically ranges between 0 and 2.

3.3 Retrieval-Augmented Generation (RAG)

The best way to understand the concept of RAG is to imagine a technology company with private internal documentation. The company wants employees to quickly access this information through a platform that leverages LLMs to answer their questions. However, an LLM alone may struggle to respond accurately using only the company's documents. This is where a RAG system becomes essential [6].

A RAG system enhances AI responses by incorporating contextual information from an external document source. The process works as follows: when a user asks a question through a conversational interface, instead of relying solely on its pre-trained general knowledge, the AI first retrieves relevant information from a knowledge base. It then uses this information to generate a more accurate, contextually relevant response—without requiring retraining. This approach significantly reduces the likelihood of the model generating incorrect or fabricated information (i.e., hallucinations).

3.3.1 Components of a RAG System

For the LLM to answer the user using context, a RAG system combines two phases (see Figure 2): retrieval and generation, assuming that the documents have been previously indexed in a knowledge base (see Figure 1), typically a vector database.

Retrieval. This phase is crucial for the quality and relevance of the responses provided by the LLM. If the appropriate information is not retrieved, the quality of the answer will be poor. To obtain relevant documents, techniques such as keyword search, semantic search based on contextual embeddings, or more traditional information retrieval methods like BM25 can be used.

Generation. In this phase, the LLM generates a coherent and relevant answer using the documents retrieved during the retrieval phase (i.e., the context).

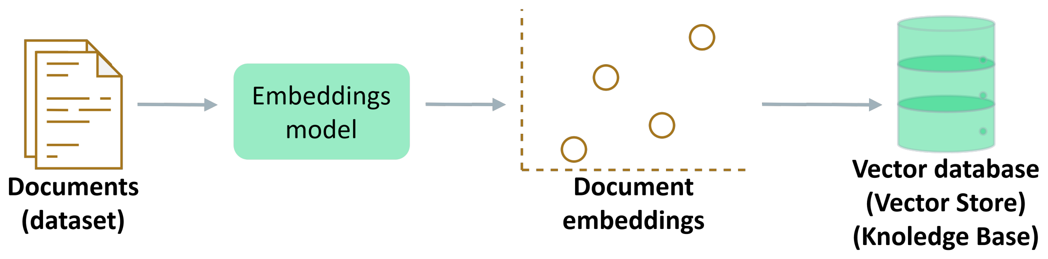

Figure 1: Indexing of documents in a knowledge base.

Several methods can improve the performance of the basic RAG system described above; some of these are summarized in Table 3.

Technique	Description
Text Fragmentation	Dividing large documents into smaller fragments to improve information retrieval, since contextual embeddings have a context limit. This facilitates retrieving more relevant information for a given query.
Query Expansion	Generating several sub-queries from the initial query, which increases the chances of retrieving the most important documents by associating embeddings with multiple queries.
Re-ranking	Reordering or filtering the retrieved documents based on their relevance to the user's query. An LLM can be used to identify the most important information among the initially retrieved documents.

Table 3: Techniques to improve the performance of a RAG system[12].

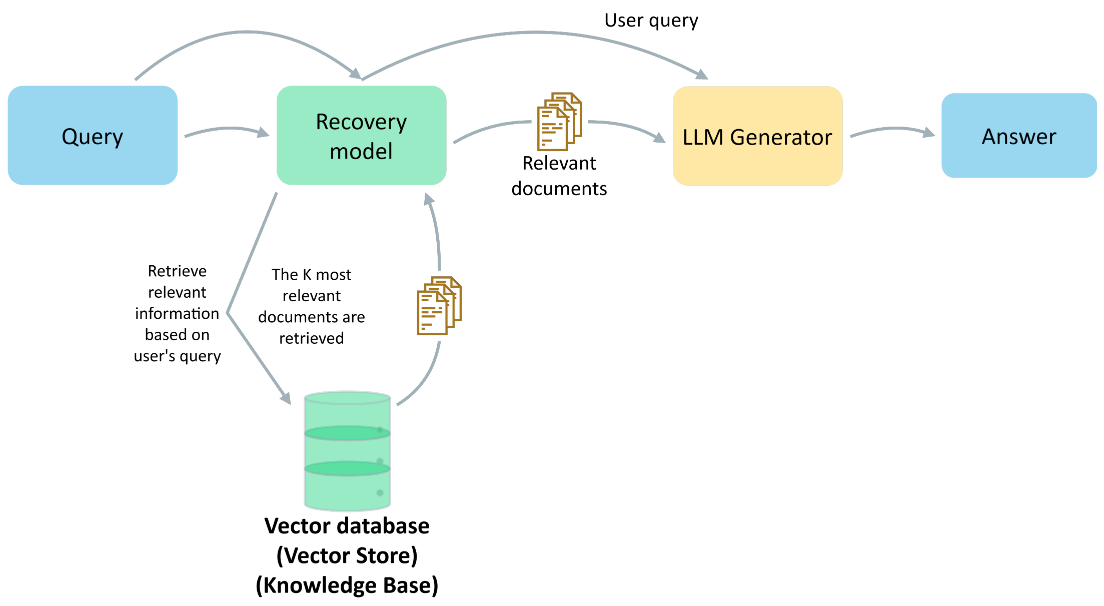
User query
Query
Recovery model
Relevant documents
LLM Generator
Answer
Retrieve relevant information based on user's query
The K most relevant documents are retrieved
Vector database (Vector Store) (Knowledge Base)

Figure 2: The process begins with a user query, which is sent to the retriever model. The retriever searches a vector database (vector store) using semantic search techniques to find the most relevant documents. These documents are then passed to a large language model (LLM), which generates a response by combining the retrieved information with the original query.

3.4 Vector Databases

To efficiently extract documents through semantic search, the embeddings must be stored in a suitable database. Vector databases are specialized systems designed for the optimal storage and retrieval of vector representations of data.

Some examples of such databases include **FAISS** (Facebook AI Similarity Search)[12], **Chroma**[13], and **ColBERT**[14] (Contextualized Late Interaction over BERT)[4]. Note that ColBERT is not a vector database per se, but rather an architecture that combines efficient semantic search with language models like BERT. ColBERT is based on a technique known as "contextualized late interaction", where the documents and queries are transformed into vector representations before the search interactions are performed[15].

3.5 Evaluation Metrics

Some evaluation metrics used in the context of RAG systems include:

- **Recall@K:** Measures how well the relevant documents are ranked among the top K results [3].
- **Mean Reciprocal Rank (MRR):** Evaluates the order in which the relevant documents are presented in the results list, measured by the ranking of the first relevant response [1].
- **FactScore:** Measures the accuracy of the generated text based on its comparison with reference documents. A high FactScore indicates that the generated answer matches the original documents, ensuring that the system provides reliable and accurate information [7, 11].
- **BERTScore:** Measures the semantic similarity between two text fragments using contextual embeddings such as those from BERT, providing precision, recall, and F1 values. A higher BERTScore indicates that the generated answers are semantically similar to the reference answers, reflecting their quality and contextual suitability [13, 9, 5].

4 Technologies

Below is a list of libraries that enable interaction with LLMs and can be used during the development of the assignment. Note that these are not the only ones; students may use others if they wish.

[12]https://github.com/facebookresearch/faiss
[13]https://docs.trychroma.com/
[14]https://github.com/stanford-futuredata/ColBERT
[15]https://jina.ai/news/what-is-colbert-and-late-interaction-and-why-they-matter-in-search/

4.1 DSPy

DSPy[16] is a library that abstracts the interaction with LLMs through Python classes. Additionally, it facilitates prompt optimization and the systematic adjustment of LLM weights. It provides support for HuggingFace models, OpenAI models, as well as locally hosted models.

There are three basic concepts for the operation of DSPy: *signatures*, *modules*, and *optimizers*:

- **Signature.** Defines the behavior of the task we want our LLM to perform by specifying the program's inputs and outputs. It is equivalent to defining a "prompt", but instead of passing text directly to the LLM, DSPy handles the construction and optimization of that definition. For example, the following signature defines a task for extracting facts from a text:

```python
class GenerateFacts(dspy.Signature):
    """
    Extract self-contained and fully contextualized facts from the given
    ↪ passage.
    """
    passage = dspy.InputField(
        desc="The passage may contain one or several claims")
    facts = dspy.OutputField(
        desc="List of self-contained and fully contextualized claims in the form
        ↪ 'subject + verb + object' without using pronouns or vague
        ↪ references", prefix="Facts:")
```

Figure 3: *DSPy Signature* for extracting facts from a text.

- **Module.** Implements a complete workflow, including the logic to connect various operations (which can be defined by one or more Signatures).

```python
class FactsGeneratorModule(dspy.Module):
    def __init__(self):
        super().__init__()
        self.generate_facts = dspy.Predict(GenerateFacts)

    def process_facts(self, facts):
        process_facts = # Logic to process facts
        return process_facts

    def forward(self, passage):
        facts = self.generate_facts(passage=passage).facts
        processed_facts = self.process_facts(facts)
        return dspy.Prediction(facts=processed_facts)
```

Figure 4: *DSPy Module* for extracting facts from a text.

  DSPy offers several modules that facilitate the creation of pipelines[17], the most important being:

  - **ChainOfThought**: Encourages the LLM to "think" (follow a chain of reasoning) before arriving at the answer.
  - **Retrieve**: Retrieves relevant information from a database in response to a query.
  - **Predict**: Generates a prediction based on the provided input.

- **Optimizer.** The optimizers in DSPy are algorithms that automatically adjust prompts and LLM weights, maximizing the quality of the results according to specific metrics. Some of the most relevant optimizers include:

  - **BootstrapFewShot**: Automatically generates examples to implement few-shot learning and improves the quality of the program by optimizing these examples, i.e., it finds the examples that maximize the metric on which the optimization is being carried out.
  - **BootstrapFewShotWithRandomSearch**: Expands the BootstrapFewShot approach by applying a random search over the generated examples, iteratively selecting the best configurations.

[16] https://dspy-docs.vercel.app/
[17] https://dspy-docs.vercel.app/api/category/modules

– **MIPRO and COPRO**: Allow for the optimization of instructions, and in the case of MIPROv2, also the few-shot examples.

– **BootstrapFinetune**: Allows distilling a prompt-based program through weight updates. The output is a DSPy program with the same weights, but where each step is carried out using a fine-tuned model instead of prompts to the LLM.

For example, code blocks 6–7 demonstrate the optimization of the `FactsGeneratorModule`. Through the `FactsDataset` class, we transform our training data into DSPy-compatible input data (`dspy.Example`). The function `optimize_module` optimizes the module based on the metric defined in `combined_score`. The parameters `mbd=4`, `mld=16`, `ncp=2`, and `mr=2` control the optimization as follows:

– `mbd` (*max_bootstrapped_demos*): Maximum number of initial automatically generated (bootstrapped) examples to feed the model.

– `mld` (*max_labeled_demos*): Maximum number of labeled examples used to fine-tune the model.

– `ncp` (*num_candidate_programs*): Number of candidate programs to be tested during optimization.

– `mr` (*max_rounds*): Maximum number of optimization rounds.

```python
def optimize_module(self, data_path, mbd=4, mld=16, ncp=2, mr=2,
↪ dev_size=0.25):
    dataset = FactsDataset(data_fpath=data_path, dev_size=dev_size)

    trainset = dataset._train
    devset = dataset._dev
    testset = dataset._test

    config = dict(max_bootstrapped_demos=mbd, max_labeled_demos=mld,
    ↪ num_candidate_programs=ncp, max_rounds=mr)
    teleprompter =
    ↪ BootstrapFewShotWithRandomSearch(metric=self.combined_score, **config)

    compiled_pred = teleprompter.compile(FactsGenerator(), trainset=trainset,
    ↪ valset=devset)
    return compiled_pred
```

Figure 5: Function for optimizing the `FactsGeneratorModule`.

```python
class FactsDataset(Dataset):
    def __init__(
        self,
        data_fpath: str,
        dev_size: Optional[float] = 0.2,
        test_size: Optional[float] = 0.2,
        text_key: str = "passage",
        seed: Optional[int] = 11235,
        *args,
        **kwargs
    ) -> None:
        super().__init__(*args, **kwargs)

        self._train, self._dev, self._test = [], [], []

        train_data = pd.read_csv(pathlib.Path(data_fpath))

        train_data, temp_data = train_test_split(
            train_data, test_size=dev_size + test_size, random_state=seed)
        dev_data, test_data = train_test_split(
            temp_data, test_size=test_size / (dev_size + test_size),
            ↪ random_state=seed)

        self._train = [dspy.Example({**row}).with_inputs(text_key) for row in
        ↪ self._convert_to_json(train_data)]
        self._dev = [dspy.Example({**row}).with_inputs(text_key) for row in
        ↪ self._convert_to_json(dev_data)]
        self._test = [dspy.Example({**row}).with_inputs(text_key) for row in
        ↪ self._convert_to_json(test_data)]
```

Figure 6: Class for converting data into DSPy-compatible examples.

```python
def combined_score(example, pred, trace=None):
    def sbert_similarity_score(example, pred, trace=None):
        try:
            scores = []
            predicted_lst = pred["facts"]
            try:
                gt_lst = ast.literal_eval(example.facts)
            except Exception as e:
                print("Error in parsing ground truth facts: ", e)
                gt_lst = example.facts.split(".")

            min_facts = min(len(predicted_lst), len(gt_lst))

            # Generate embeddings for predicted and ground truth facts
            predicted_embeddings = tr_model.encode(predicted_lst[:min_facts])
            gt_embeddings = tr_model.encode(gt_lst[:min_facts])

            # Calculate cosine similarity for each pair of embeddings
            for pred_emb, gt_emb in zip(predicted_embeddings, gt_embeddings):
                similarity = 1 - cosine(pred_emb, gt_emb)
                scores.append(similarity)

            return np.mean(scores)  # Return the average similarity score

        except Exception as e:
            return 0.0

    return sbert_similarity_score(example, pred, trace)
```

Figure 7: Metric for optimizing the `FactsGeneratorModule`.

4.2 ollama

`ollama`[18] allows you to manage and deploy large language models (LLMs) locally. Although `ollama` simplifies local deployment of LLMs, it is important to note that certain minimum resources are required to run these models: at least 8 GB of RAM for 7B models, 16 GB for 13B models, and 32 GB for 33B models.

Once a model is deployed, interaction with `ollama` models can be performed either via the API or using the Python library[19]. In interactions with `ollama`, you can use both the functions `ollama.chat` and `ollama.generate` in Python, as well as the API endpoints `/api/chat` and `/api/generate`, which accept different parameters. The main difference between them is that `ollama.chat` maintains the context of previous interactions (ideal for conversations with history), whereas `ollama.generate` does not. Code blocks 8 and 9 show how to interact with both via Python code. Additionally, Section 5 shows how you can call the `llama3.1` and `llama3.2` models that we have deployed on the department's servers. More details can be found in the documentation[20].

The following links might be of interest for building RAG systems with `ollama`:

- Ollama Embedding models
- RAG implementation from scratch

```python
import requests
import json

def generate_response(model, system, prompt,
↪ url="http://kumo01:11434/api/generate"):
    data = {
        "model": model,
        "system": system,
        "prompt": prompt,
        "stream": False,
    }

    headers = {"Content-Type": "application/json" }

    try:
        response = requests.post(url, headers=headers, data=json.dumps(data))
        if response.status_code == 200:
            return response.json()
        else:
            return f"Error: {response.status_code}, {response.text}"

    except Exception as e:
        return f"An error occurred: {str(e)}"

generate_response("llama3.2", "You are a helpful AI Assistant", "What are you?")
```

Figure 8: Example function to call the `ollama` API.

```python
import ollama

response = ollama.chat(
    model=llm_model,
    messages=[
        {
            'role': 'user',
            'content': 'Describe me what topic modeling is as if I were a kid',
        },
    ])
print(response['message']['content'])
```

Figure 9: Example function to call the `ollama` API through the Python library.

[18] https://ollama.com/library
[19] https://pypi.org/project/ollama-python/
[20] https://github.com/ollama/ollama/tree/main/docs

4.3 LlamaIndex

`LlamaIndex`[21] (formerly `GPT-Index`) facilitates the development of applications with LLMs. It stands out for its ability to load data from multiple formats (e.g., PDFs, SQL databases), index data in various vector databases, perform RAG queries, and offer modules for evaluation and performance tuning. Additionally, it allows connections with both free and commercial language models. Below are links with detailed instructions for instantiating and generating text with free models, as well as for developing a RAG system:

- Hugging Face LLMs
- HuggingFaceInferenceAPI
- Building RAG from Scratch (Open-source only!)

4.4 LangChain

`LangChain`[22] is a library that, like `LlamaIndex`, abstracts interaction with LLMs through various components. Both offer similar functionalities, but they differ in focus and optimization. `LlamaIndex` emphasizes efficiency and simplicity, particularly for search and information retrieval tasks, whereas `LangChain` is more suitable for complex applications involving multiple LLM workflows, such as chained text generation, integration with external APIs, and advanced context management[23].

Below are some useful links if you prefer to use `LangChain` for developing the assignment:

- Implementing RAG with LangChain and Hugging Face
- LangChain Document QA with ollama
- RAG Pipeline with Ollama and ChromaDB
- Building an Open-Source RAG Application Using Ollama, TextEmbed, and LangChain

5 Deployment Environment at UC3M

Students will have access to the models deployed at UC3M through the institution's virtual labs available **Monday to Friday from 18:00 to 02:00, and all day on weekends (Saturday and Sunday)**. To access these models, you must follow the instructions provided in this manual (you need to log in with a UC3M institutional account to view the manual), using the following credentials:

- **Username:** mastermlh
- **Password:** IeV=ie8o

Students can develop their projects on the virtual lab machines if they prefer. However, **it is important that you always save your changes**, either on Google Drive or, preferably, in a GitHub repository, as there is a possibility that these may be lost when logging off the machines.

The currently deployed LLMs are `llama3.1:8b-instruct-q8_0` and `qwen2.5:7b-instruct`. Additionally, the embedding model `mxbai-embed-large`[24] is available. However, students may request the deployment of other models if needed for their assignment.

We can interact with these LLMs using a `curl` command, as shown in code block 10, or via the technologies mentioned in Section 4. Note that students will need to adjust the model URL, as these are not deployed on local machines and thus are not accessible via `localhost`.

```
curl http://kumo01:11434/api/generate -H "Content-Type: application/json" \
-d '{"model": "llama3.1:8b-instruct-q8_0", "prompt": "Convince me that you are not
↪  a machine", "stream": false}'
```

Figure 10: Example call to the `llama3.2` model using curl.

[21] https://github.com/run-llama/llama_index
[22] https://github.com/langchain-ai/langchain
[23] https://www.gettingstarted.ai/langchain-vs-llamaindex-difference-and-which-one-to-choose/
[24] https://ollama.com/blog/embedding-models

References

[1] Nick Craswell, Onno Zoeter, Michael Taylor, and Bill Ramsey. An experimental comparison of click position-bias models. In *Proceedings of the 2008 international conference on web search and data mining*, pages 87–94, 2008.

[2] Miguel de la Vega. Explorando los hiperparámetros temperature y top p en los modelos de lenguaje de openai. https://medium.com/@1511425435311/explorando-los-hiperparámetros-temperature-y-top-p-en-los-modelos-de-lenguaje-de-openai-7ee74d8912c0, 2023.

[3] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[4] Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pages 39–48, 2020.

[5] Hwang Youn Kim, Ghazanfar Ali, and Jae-In Hwang. Enhancing doctor-patient communication in surgical explanations: Designing effective facial expressions and gestures for animated physician characters. *Computer Animation and Virtual Worlds*, 35(3):e2236, 2024.

[6] Patrick S. H. 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. *CoRR*, abs/2005.11401, 2020.

[7] Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen-tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. FActScore: Fine-grained atomic evaluation of factual precision in long form text generation. In *EMNLP*, 2023.

[8] Anaya Multimedia. *Inteligencia artificial generativa con modelos de ChatGPT y OpenAI*. Anaya Multimedia, 2023. Títulos Especiales.

[9] Mercy Ranjit, Gopinath Ganapathy, Ranjit Manuel, and Tanuja Ganu. Retrieval augmented chest x-ray report generation using openai gpt models. In Kaivalya Deshpande, Madalina Fiterau, Shalmali Joshi, Zachary Lipton, Rajesh Ranganath, Iñigo Urteaga, and Serene Yeung, editors, *Proceedings of the 8th Machine Learning for Healthcare Conference*, volume 219 of *Proceedings of Machine Learning Research*, pages 650–666. PMLR, 11–12 Aug 2023.

[10] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, et al. The prompt report: A systematic survey of prompting techniques. *arXiv preprint arXiv:2406.06608*, 2024.

[11] Sheikh Shafayat, Eunsu Kim, Juhyun Oh, and Alice Oh. Multi-fact: Assessing multilingual llms' multi-regional knowledge using factscore, 2024.

[12] Syntonize. Sistema rag: Qué es y conceptos claves. https://www.linkedin.com/pulse/sistema-rag-qué-es-y-conceptos-claves-syntonize-kxqge/, 2023.

[13] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with BERT. *CoRR*, abs/1904.09675, 2019.