Developer Guide: *upsay_jupyter_ai* toolkit

Paulo Henrique Couto[1]

[1]Université Paris-Saclay, France
https://mydocker.universite-paris-saclay.fr/shell/join/upsayai
https://gitlab.dsi.universite-paris-saclay.fr/upsayai
https://pypi.org/project/upsay_jupyter_ai

November 13, 2024

1 Initialization

In order to provide flexibility to users during a first testing period, the toolkit currently requires the initialization as an object of the class UPSayAI, from the upsay_jupyter_ai python package, where several parameters can be configured. In the initialization, the parameters are made available for all the functions in the class, their values are checked to ensure that their values are in the expected range and format, the IR algorithm (for RAG) and LLM models are configured, and the magic commands are set.

```python
class UPSayAI:
    def __init__(self, model_name="aristote/llama", model_access_token=None,
         model_temperature=0.8, model_max_tokens=2048, rag_model="dpr",
        rag_min_relevance=0.5, rag_max_results=5, recommendation_min_relevance
        =0.7, quiz_difficulty="moyen", quiz_min_relevance=0.2,
        quiz_max_results=100, num_questions_quiz=5, course_corpus_file="
        corpus_ISD.csv"):

        self.model_name = model_name
        self.model_access_token = model_access_token
        self.model_temperature = model_temperature
        self.model_max_tokens = int(model_max_tokens)
        self.rag_model = rag_model
        self.rag_min_relevance = rag_min_relevance
        self.rag_max_results = int(rag_max_results)
        self.recommendation_min_relevance = recommendation_min_relevance
        self.quiz_difficulty = quiz_difficulty
        self.quiz_min_relevance = quiz_min_relevance
        self.quiz_max_results = int(quiz_max_results)
        self.num_questions_quiz = int(num_questions_quiz)
        self.course_corpus_file = course_corpus_file

        self.check_values()
        self.config_rag()
        self.config_llm()
        self.setup_magics()
```

All the parameters are described below:

- **model_name:** Model name among three possible alternatives: "huggingface/llama", which represents the model meta-llama/Meta-Llama-3-8B-Instruct imported from Hugging Face; and "aristote/llama" and "aristote/mixtral", which represent the casperhansen/llama-3-70b-instruct-awq and the casperhansen/mixtral-instruct-awq models, respectively, stored in the Aristote Dispatcher;

- **model_access_token:** Access token to use the LLM selected by the model_name parameter. In the case of "huggingface/llama", it should be a Hugging Face access token, whereas for either "aristote/llama" or "aristote/mixtral", it should be an Aristote Dispatcher access token. For now, the user must initialize the toolkit using a valid individual token;

- **model_temperature:** LLM's temperature parameter for generation, which determines its creativity. It must be a float between 0 and 1;

- **model_max_tokens:** LLM's max tokens parameter for generation. In the case of "huggingface/llama", it is used as max_new_tokens instead, since the model does not expect a max_tokens parameters. It must be an int between 512 and 4096;

- **rag_model:** IR model to be used by the RAG logic. Current options are "dpr", which uses the cosine similarity of dense embeddings generated by the AgentPublic/dpr-question_encoder-fr_qa-camembert model available in Hugging Face, and "bm25", which uses Okapi BM25;

- **rag_min_relevance:** Minimum relevance threshold for the cosine similarity in the RAG logic when using "dpr" as rag_model. It must be a float between 0 and 1;

- **rag_max_results:** Maximum number of extracts (in this case, Jupyter cells) to be passed to the LLM as context by the RAG logic. Used both by the "dpr" and "bm25" algorithms. It must be an int between 1 and 100;

- **recommendation_min_relevance:** Minimum relevance threshold for the cosine similarity in the recommendation system, which suggests notebooks to be reviewed by the students based on their questions. It must be a float between 0 and 1;

- **quiz_difficulty:** Level of difficulty of the questions generated for the quiz. It must be either "facile" for an easy level, "moyen" for an intermediate level, or "difficile" for a difficult level;

- **quiz_min_relevance:** Minimum relevance threshold for the cosine similarity in the RAG logic using "dpr" model to provide relevant extracts to the quiz generation. This is only used if the quiz is generated about a specific subject rather than about a specific notebook, since in this case the LLM will receive relevant passages from the classes to use as reference to create the questions. It must be a float between 0 and 1;

- **quiz_max_results:** Maximum number of extracts (in this case, Jupyter cells) to be passed to the LLM as context by the RAG logic to generate the quiz. This is only used if the quiz is generated about a specific subject rather than about a specific notebook. It must be an int between 10 and 1000;

- **num_questions_quiz:** Number of questions to be generated by the quiz. Note that, every time the LLM fails to correctly format a question (python dictionary), it is automatically deleted and the number of questions is reduced by one. Therefore it is possible that a quiz is generated with fewer questions that expected. It must be an int between 3 and 10;

- **course_corpus_file:** CSV file with all the course corpus (jupyter cells) for RAG and recommendation system. It contains every cell's content, alongside their tokenized version, for "bm25", and pre-calculated dense embeddings, for "dpr". Note that, for now, the ISD Corpus CSV file must be uploaded to the MyDocker environment so that the toolkit can access it.

2 Assert of parameters values

During the initialization of an object of the class UPSayAI, the function check_values() is called to verify if the values are in the expected range and format:

```
def check_values(self):

    assert self.model_name in ["aristote/llama", "aristote/mixtral", "
       huggingface/llama"], "Invalid model_name. Current supported values
       are 'aristote/llama', 'aristote/mixtral', or 'huggingface/llama'"

    assert 0 <= self.model_temperature <= 1, "Invalid model_temperature. The
        temperature value has to be a float between 0 and 1"

    assert 512 <= self.model_max_tokens <= 4096, "Invalid model_max_tokens.
       The model_max_tokens value has to be an int between 512 and 4096"

    assert self.rag_model in ["dpr", "bm25"], "Invalid rag_model. Current
       supported values are 'dpr' or 'bm25'"

    assert 0 <= self.rag_min_relevance <= 1, "Invalid rag_min_relevance. The
        rag_min_relevance value has to be a float between 0 and 1"

    assert 1 <= self.rag_max_results <= 100, "Invalid rag_max_results. The
       rag_max_results value has to be an int between 1 and 100"

    assert 0 <= self.recommendation_min_relevance <= 1, "Invalid
       recommendation_min_relevance. The recommendation_min_relevance value
       has to be a float between 0 and 1"

    assert self.quiz_difficulty in ["facile", "moyen", "difficile"], "
       Invalid quiz_difficulte. Current supported values are 'facile', '
       moyen', or 'difficile'"

    assert 0 <= self.quiz_min_relevance <= 1, "Invalid quiz_min_relevance.
       The quiz_min_relevance value has to be a float between 0 and 1"

    assert 10 <= self.quiz_max_results <= 1000, "Invalid quiz_max_results.
       The quiz_max_results value has to be an int between 10 and 1000"

    assert 3 <= self.num_questions_quiz <= 10, "Invalid num_questions_quiz.
       The num_questions_quiz value has to be an int between 3 and 10"
```

3 Project Organization

This project can be organized into two main pillars: first, how to improve the performance of an LLM for questions and commands specific to a university course; second, how to best integrate this toolset into the Jupyter ecosystem (i.e., notebooks).

For the first pillar, several alternatives were discussed: optimizing the prompt passed to the language model (prompt engineering), enriching the prompt through course extracts pertinent to the student's question (RAG), or even calibrating the model using the course materials, such as Low-Rank Adaptation (LoRA) or complete fine tuning of all the weights. Due to the high computational costs involved in weight adjustment processes, especially in fine-tuning, coupled with the short duration of the project, the initial approach chosen was to implement a RAG to enrich the prompt passed to the LLM with a context made up of extracts from the course.

For the second pillar, the main point considered was how to make the user experience practical and satisfactory to the point of encouraging the use of this tool over third-party platforms. With this in mind, the chosen strategy was to implement LLM through custom python magic commands, allowing the user to invoke the model in a common code cell through simple command markers. Moreover, this allows the creation of task-specific command markers, such as question-answer (Q&A) and code generation, which enables the creation of custom-tailored prompts according to the desired task.

4 LLMs

Several open weight models were initially tested, however only three were chosen to be made available in the first prototype: meta-llama/Meta-Llama-3-8B-Instruct, available on the Hugging Face platform[1]; and casperhansen/llama-3-70b-instruct-awq and casperhansen/mixtral-instruct-awq, available on CentraleSupélec's Aristote Dispatcher[2].

For the model stored on Hugging Face, the AutoTokenizer and AutoModelForCausalLM functions from the transformers python package were used alongside a valid Hugging Face access token. Note that, for the Hugging Face model, a TextIteratorStreamer needs to be used for the token-by-token display of the answers.

```python
from transformers import AutoTokenizer, AutoModelForCausalLM,
    TextIteratorStreamer
# Config LLM for huggingface models (for now only Meta-Llama-3-8B-Instruct)
def config_llm_huggingface(self):
    login(token=self.model_access_token)
    model_id="meta-llama/Meta-Llama-3-8B-Instruct"
    self.device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    self.tokenizer=AutoTokenizer.from_pretrained(model_id)
    self.model=AutoModelForCausalLM.from_pretrained(
                model_id,
                torch_dtype=torch.bfloat16
               )

    self.model.to(self.device)

    self.terminators = [
                    self.tokenizer.eos_token_id,
                    self.tokenizer.convert_tokens_to_ids("<|eot_id|>")
                ]

    if self.tokenizer.pad_token is None:
        self.tokenizer.pad_token = self.tokenizer.eos_token

    # Configure streamer for token-by-token display
    self.streamer = TextIteratorStreamer(self.tokenizer, skip_prompt = True)
```

For the models stored on the CentraleSupélec's Aristote Dispatcher, the ChatOpenAI function from the langchain_openai python package was used alongside an API base URL and valid access token provided by CentraleSupélec. In this case, it is not necessary to use a TextIteratorStreamer for the token-by-token display of the answers, since we can simply define the model as shown below and use self.model.stream(messages) in the generation step.

[1] https://huggingface.co/
[2] https://github.com/CentraleSupelec/aristote-dispatcher

```python
from langchain_openai import OpenAI, ChatOpenAI
# Config LLM for aristote models
def config_llm_aristote(self):
    if self.model_name == "aristote/llama":
        self.model = ChatOpenAI(openai_api_base = "https://dispatcher.
            aristote.centralesupelec.fr/v1", openai_api_key = self.
            model_access_token, model = "casperhansen/llama-3-70b-instruct-
            awq", temperature=self.model_temperature, max_tokens=self.
            model_max_tokens)
    elif self.model_name == "aristote/mixtral":
        self.model = ChatOpenAI(openai_api_base = "https://dispatcher.
            aristote.centralesupelec.fr/v1", openai_api_key = self.
            model_access_token, model = "casperhansen/mixtral-instruct-awq",
            temperature=self.model_temperature, max_tokens=self.
            model_max_tokens)
```

5 RAG methodologies

The main point of a RAG model is to enrich the prompt given to the LLM through pertinent information about the subject addressed in the original question asked by the student. However, this means that the model is required to accurately and quickly find the extracts from the course corpus that reach a minimum level of relevance with respect to the question, especially considering that LLM models have a context window of limited length (e.g., 8K tokens in the case of Meta's Llama 3 Instruct).

The balance between precision and agility is a key consideration in selecting methodologies for Information Retrieval (IR), particularly when deciding on the type of vectorization for the course materials (corpus) and the student questions (queries). Sparse vectors, for instance, offer significant computational efficiency but come with notable drawbacks, including an inability to capture semantic relationships between synonyms and a lack of robustness against spelling errors. In contrast, dense embedding techniques provide improved handling of synonyms and are more resilient to typographical errors. However, these advantages come at the cost of increased computational intensity.

With this in mind, two IR models were tested internally and made available for the user to choose from in the first version of the prototype: Okapi BM25, using the top n highest scores (see subsection 5.1); and cosine similarity with dense embeddings generated by a CamemBERT DPR model, using a minimum relevance threshold (see subsection 5.2).

5.1 Okapi BM25

Okapi BM25 is a widely used IR model that excels in ranking documents by their relevance to a given query. In this project, this model was implemented using the rank-bm25 Python package.

The BM25 algorithm is grounded in probabilistic retrieval theory, leveraging term frequency-inverse document frequency (TF-IDF) principles to calculate a relevance score for each document in the corpus. This score reflects the presence and frequency of query terms within the document, adjusted for document length and overall term distribution. Essentially, BM25 aims to balance term frequency, document length, and term specificity to provide a robust measure of relevance.

In this implementation, we first prepared the course corpus by tokenizing and preprocessing the documents, that is, the Jupyter Notebooks' cells. Each cell was represented as a list of tokens to facilitate efficient processing by the BM25 model. The model was then initialized with the preprocessed corpus, using default hyperparameters typically effective for a wide range of text retrieval tasks.

```python
from rank_bm25 import BM25Okapi
self.bm25_index = BM25Okapi(self.df.tokenized_content.tolist())
```

For each student query, BM25 calculates a relevance score for every document in the corpus. To control the number of top-ranking extracts passed to the RAG model, a parameter called rag_max_results was introduced. This parameter specifies how many of the highest-scoring cells should be passed as context to the LLM.

```
# IR function based on BM25 to find relevant extracts
def bm25_search(self, search_string):
    search_tokens = word_tokenize(search_string)
    scores = self.bm25_index.get_scores(search_tokens)
    top_indexes = np.argsort(scores)[::-1][:self.rag_max_results]
    return top_indexes
```

5.2 CamemBERT DPR

The CamemBERT-based Dense Passage Retrieval (DPR) model used in this project was AgentPublic/dpr-question_encoder-fr_qa-camembert, designed to handle the semantic search tasks in the French language. It leverages the CamemBERT model, a robust French language model, as its base and is fine-tuned on a combination of three French Question & Answer (Q&A) datasets: PIAFv1.1, FQuADv1.0, and SQuAD-FR (the French translation of the SQuAD dataset).

```
from transformers import AutoTokenizer, AutoModel

self.tokenizer_dpr = AutoTokenizer.from_pretrained("etalab-ia/dpr-
    question_encoder-fr_qa-camembert", do_lower_case=True, resume_download=
    None)
self.model_dpr = AutoModel.from_pretrained("etalab-ia/dpr-question_encoder-
    fr_qa-camembert", return_dict=True, resume_download=None)
```

The DPR approach involves encoding both the course materials (corpus) and the student questions (queries) into dense vectors, capturing semantic similarities beyond mere keyword matches. This capability is particularly advantageous for handling synonyms, spelling variations, and context, thus providing more relevant search results compared to traditional sparse vector models like BM25.

In this implementation, the course materials were first preprocessed and the embeddings were pre-calculated using the CamemBERT DPR model. Each cell in the notebooks was transformed into a dense vector representation. Similarly, student queries were encoded into dense vectors in real time during the magic commands utilization. The toolkit then calculates the cosine similarity between the query vector and each cell vector in the corpus, identifying the most semantically relevant extracts.

To manage the number of cells passed to the RAG model, a minimum relevance threshold parameter rag_min_relevance was introduced. This threshold ensures that only documents with a cosine similarity score above a certain level are selected, maintaining the quality and relevance of the contextual information provided to the LLM.

```
# IR function based on DPR to find relevant extracts
def dpr_search(self, search_string, threshold, max_results):
    query_embedding = self.get_query_embedding(search_string)
    cos_scores = self.cos_sim(query_embedding, self.corpus_embeddings)[0]
    cos_scores_np = (-1) * np.sort(-cos_scores.detach().numpy())
    cos_idx_np = np.argsort(-cos_scores.detach().numpy())
    mask = cos_scores_np > threshold
    index_high_score = cos_idx_np[mask][:max_results]
    return index_high_score
```

6 *%ai_help* command

The *%ai_help* command is designed to open a quick user guide with a brief presentation of each command available in the toolkit. All formatting was done with the aim of making the toolkit user guide as intuitive and didactic as possible: in the examples of using the commands, the formatting simulates the look of a code cell in the Jupyter environment in order to highlight how the command must be entered in the notebook. The *%ai_help* command has no connection with the LLM, being just a series of *display(Markdown(...))* and/or *display(HTML(...))* in the background.

7 *%ai_question* command

7.1 Prompt

The prompt used by the *%ai_question* command is as shown below, where messages_history consists of the past exchanges between student and assistant, including the current question, since the initialization of the kernel (limited to 10 exchanges, i.e., 20 messages, due to the models' maximum context length). Therefore, the assistant can maintain a continuous interaction with the student, being able to handle follow-up questions.

```
self.messages_history.append({"role": "user", "content": question})

messages = [{"role": "system", "content": self.persona_prompt_qa +
    rag_prompt}] + [x for x in self.messages_history]
```

The persona_prompt_qa, declared in the config_llm function, is the prompt responsible for optimizing the LLM for a Q&A task, defining the context of the command it will receive and explaining the expected answer format.

self.persona_prompt_qa = "Vous êtes un assistant virtuel qui aide les étudiants de l'Université Paris-Saclay avec des questions dans le domaine de la programmation et de la science des données, en répondant toujours de manière pédagogique et polie. Lorsque c'est possible, essayez d'utiliser des informations et des exemples tirés du matériel de cours pour aider l'étudiant à comprendre, en soulignant dans votre explication où l'étudiant a vu ce contenu être employé pendant le cours et en mettant toujours en contexte pour une réponse bien structurée. N'incluez pas d'images ou de médias dans votre réponse, uniquement du texte en format markdown."

The rag_prompt is the prompt responsible for introducing the relevant course extracts to help the LLM in the answer.

rag_prompt = f" Voici un extrait du support de cours qui pourra vous être utile dans votre réponse à l'étudiant : {rag}"

Finally, the rag variable is the ensemble of relevant extracts concatenated into a single string. If the IR algorithm has failed in finding any relevant extract, the rag variable is then loaded with a message informing the LLM about the absence of relevant extracts in the course material and suggesting the model to either answer the question, should it know the answer (e.g. general knowledge questions), or inform the student that it has not found the information and recommend them to contact the professor. Below is a part of the get_context function, which is responsible for setting the value of the rag variable, where index_high_score is a list of the indeces of relevant extracts (jupyter cells) found in the corpus. If no relevant cell was found, it returns a pre-defined message. It was noted during tests that this strategy of passing the predefined message in the case of no relevant extracts being identified helps the LLM to reduce the frequency of hallucinations, especially for course-specific questions which the IR algorithm could not find any relevant context.

```
if(len(index_high_score) > 0):
    rag_content = "\n".join(self.df["content"].iloc[index_high_score])
    return rag_content

return "Aucun extrait du support de cours n'a ete trouve contenant des
   informations pertinentes sur la question posee par l'etudiant. Si vous ne
    connaissez pas non plus la reponse, informez l'etudiant que vous n'avez
   trouve aucune information sur ce qui lui est demande et recommandez-lui
   de contacter les professeurs responsables du cours."
```

7.2 Post-processing

The post-processing of the *%ai_question* command is very simple, from which we can highlight two main points: display of the response being generated token by token, as opposed to a single display of the entire response at the end of generation; and the reading recommendation at the end of the answer if the IR algorithm found relevant extracts.

For the first point, the TextIteratorStreamer[3] was used for the model from the Transformers library (Hugging Face), while the funciton call .stream() was implemented for the models in Aristote Dispatcher. To illustrate, below is the function responsible for generating a response when an Aristote Dispatcher model is chosen by the user, where the variable messages represent the full prompt to be passed to the LLM.

```
def aristote_generate(self, messages, initial_response):
        generated_text = initial_response
        for chunk in self.model.stream(messages):
            generated_text += chunk.content
            update_display(Markdown(generated_text),
                           display_id=self.display_output.display_id)
        return generated_text
```

For the second point, in the case of relevant extracts with respect to the query (question) being found in the course corpus, the message "Pour plus d'informations sur ce sujet, il peut être utile de cliquer sur les liens pour réviser les cours :", followed by the hyperlinks to the relevant course notebooks, is displayed in the same token-by-token style, event though it is not generated by the LLM, to ensure a smooth and seamless integration to the rest of the answer. At the end of each iteration, both the student's question (without extracts from the IR algorithm) and the LLM's answer are appended to the messages_history:

```
self.messages_history.append({"role": "assistant",
                              "content": generated_text})
```

8 *%ai_code* command

8.1 Prompt

The prompt used by the *%ai_code* command is as shown below, where code_inst consists of the code instructions passed by the student when invoking the magic command.

```
messages = [{"role": "system", "content": self.persona_prompt_code},
           {"role": "user", "content": code_inst}]
```

[3] https://huggingface.co/docs/transformers/internal/generation_utils

The persona_prompt_code, declared in the config_llm function, is the prompt responsible for optimizing the LLM for a Code Generation task, defining the context of the command it will receive and explaining the expected answer format.

self.persona_prompt_code = “Vous êtes un assistant virtuel qui écrit des codes python selon les instructions d’un étudiant de l’Université Paris-Saclay. Utilisez toujours des commentaires et documentez bien vos codes pour les rendre faciles à comprendre pour l’étudiant. Mettez toujours tout le code, y compris les éventuels exemples pratiques d’utilisation, dans un seul bloc délimité par ’“‘python’ au début et ’“‘ à la fin. Ne générez pas plus d’un bloc de code, générez toujours un seul bloc avec tout le code et les exemples d’utilisation à l’intérieur afin que l’étudiant puisse tout exécuter dans une seule cellule jupyter. Utilisez des bibliothèques et des fonctions avec lesquelles l’étudiant est plus susceptible d’être familier, donnez la préférence à des solutions plus simples tant qu’elles sont correctes et entièrement fonctionnelles. Assurez-vous que votre code est correct, fonctionnel et que les éventuels exemples d’utilisation fonctionneront parfaitement et donneront des résultats corrects lorsqu’ils seront exécutés par l’étudiant. Terminez toujours par un court paragraphe après le bloc de code python (délimité par ’“‘python’ au début et ’“‘) avec une description textuelle et des explications pour l’étudiant afin d’améliorer sa compréhension du sujet et du code généré.”

8.2 Post-processing

While the LLM’s response is being generated, that is, being displayed token by token as a Markdown, an initial message is kept at the begining of the answer stating that “Une fois la réponse complétée, le code sera déplacé dans une nouvelle cellule juste en dessous.” Then, once the response is complete, the post-processing finds the code block inside the answer, remove it from the markdown with the explanations, and add it to a new cell. For a responsible use, comments about the command that originated the code and highlighting the risk of errors due to the AI-generated nature of the code are added in the beginning. This post-processing logic is shown below, where the the block of the string containing the code is generated_text[code_init:code_end].

~~~
# Check for the code block inside of the generated answer
if '```python' in generated_text and '```\n' in generated_text:
    code_init = generated_text.find('```python') + len('```python')
    code_end = generated_text.find('```\n')
    # Put a comment before
    if(cell == ""):
        code = "# Code genere par une intelligence artificielle sujet a des
            erreurs\n" + f"# Instructions pour la generation : '{code_inst}'\n"
             + generated_text[code_init:code_end]
    else:
        code = "# Code genere par une intelligence artificielle sujet a des
            erreurs\n" + f"'''\nInstructions pour la generation : \n{code_inst}
            '''\n" + generated_text[code_init:code_end]

    code_remove = generated_text[(code_init - len('```python')):(code_end +
        len('```\n'))]
    generated_text = generated_text.replace(code_remove, "")
    generated_text += "\n\n" + "Le code genere par l'IA a ete insere dans la
        cellule ci-dessous"
    self.ip.set_next_input(code) # Generate new cell with code

display(Markdown(generated_text), clear = True)
~~~

9 *%ai_explain* command

9.1 Prompt

The prompt used by the *%ai_explain* command is as shown below, where clean_cell consists of the cell content, that is, the code to be explained by the LLM, after a preprocessing to avoid errors (i.e. replace “{” and “}” by “{{” and “}}”, respectively, because curly brackets raise errors in f-strings as they usually represent variables). The cell content (code) may be passed directly (when using the %%ai_explain command on the first line of the cell with the code to be explained) or may be passed indirectly using a reference by the cell number. In this second case, the actual code to be explained is retrieved using self.ip.user_ns[‘In’][number_cell].

```
def ai_explain(self, line="", cell=""):
    initial_response = ""
    try:
        # For the "%ai_explain XX" format
        number_cell = int(line.strip())
        cell_content = self.ip.user_ns['In'][number_cell]
        # Since {} are seen as placeholders in f-strings and cause errors,
            we change them for {{}}
        clean_cell = cell_content.replace('{', '{{').replace('}', '}}')
    except:
        # For when the user puts the command on the first line of the code
            cell itself
        # Since {} are seen as placeholders in f-strings and cause errors,
            we change them for {{}}
        clean_cell = cell.replace('{', '{{').replace('}', '}}')

    messages = [{"role": "system", "content": self.persona_prompt_explain},
                {"role": "user", "content": clean_cell}]
```

The persona_prompt_explain, declared in the config_llm function, is the prompt responsible for optimizing the LLM for a Code Explanation task, defining the context of the command it will receive and explaining the expected answer format.

self.persona_prompt_explain = “Vous êtes un assistant virtuel qui explique des codes python à un étudiant de l’Université Paris-Saclay d’une manière claire et informative. Fournissez une analyse textuelle complète en français du code que vous sera donné, en expliquant son objectif global, la logique utilisée, les principales variables, les bibliothèques et les fonctions utilisées par le programmeur, et les éventuels outputs attendus s’il y en a. Vous pouvez utiliser des extraits du code dans votre explication, délimité par ’“‘python’ au début et ’“‘ à la fin, si vous pensez que cela aidera la compréhension de l’étudiant, et inclure des exemples possibles de quand et comment le code pourrait être utilisé par l’étudiant. Préférez expliquer l’objectif global et la logique dans un flux continu à travers un paragraphe plutôt que d’utiliser des sous-titres, mais énumérez les principales variables sous forme de puces. L’étudiant doit être en mesure de comprendre pleinement le code après avoir lu votre explication. Terminez toujours par une conclusion résumant l’explication. Rédigez toute votre explication en français.”

9.2 Post-processing

The post-processing of the *%ai_explain* command is almost identical to the one done in the *%ai_question* command, where the explanation is generated and displayed token-by-token to the user using the same techniques. When using the %%ai_explain command on the first line of the cell with the code to be explained, the execution of said cell runs only the magic command therefore generating an explanation of it, without executing the code itself within the cell. To remind the user of this and reinforce the need to remove the magic command in order to execute the actual code of the cell, a message is displayed in this case.

```
except:
    # When %%ai_explain command on the first line of the code cell itself
    initial_response = "Note : Afin d'executer le contenu de cette cellule
       plutot que d'en generer une explication, vous devez supprimer la
       commande magique `%%ai_explain` sur la premiere ligne et executer
       cette cellule a nouveau.\n\n"
```

10 *%ai_debug* command

10.1 Prompt

The prompt used by the *%ai_debug* command is as shown below, where clean_cell consists of the cell content, that is, the code presenting an error to be debugged by the LLM, after a preprocessing to avoid errors (i.e. replace "{" and "}" by "{{" and "}}", respectively, because curly brackets raise errors in f-strings as they usually represent variables). The cell content (code) may be passed directly (when using the %%ai_debug command on the first line of the cell with the code to be debugged) or may be passed indirectly using a reference by the cell number. In this second case, the actual code to be debugged is retrieved using self.ip.user_ns['In'][number_cell]. In both cases, the code within the cell is executed using self.ip.run_cell to get the error message to be able to pass it to the LLM.

```
def ai_debug(self, line="", cell=""):
    initial_response = ""
    try:
        # For the "%ai_explain XX" format
        number_cell = int(line.strip())
        cell_content = self.ip.user_ns['In'][number_cell]
        # Since {} are seen as placeholders in f-strings and cause errors,
           we change them for {{}}
        clean_cell = cell_content.replace('{', '{{').replace('}', '}}')
        cell_out = self.ip.run_cell(self.ip.user_ns['In'][number_cell])
        clear_output(wait=True)
        if not cell_out.success:
            error_message = f"error_before_exec='{cell_out.error_before_exec}',
               error_in_exec='{cell_out.error_in_exec}'"
    except:
        # For when the user puts the command on the first line of the code
           cell itself
        # Since {} are seen as placeholders in f-strings and cause errors,
           we change them for {{}}
        clean_cell = cell.replace('{', '{{').replace('}', '}}')
        cell_out = self.ip.run_cell(cell)
        clear_output(wait=True)
        if not cell_out.success:
                error_message=f"error_before_exec='{cell_out.error_before_exec}',
                    error_in_exec='{cell_out.error_in_exec}'"
    if not cell_out.success:
                messages = [{"role": "system", "content": self.
                    persona_prompt_debug}, {"role": "user", "content": f"Code
                    :\n{clean_cell}\nMessage d'erreur:\n{error_message}"}]
```

The persona_prompt_debug, declared in the config_llm function, is the prompt responsible for optimizing the LLM for a Code Debugging task, defining the context of the command it will receive and explaining the expected answer format.

self.persona_prompt_debug = "Vous êtes un assistant virtuel qui aide les étudiants de l'Université Paris-Saclay à déboguer des codes Python qui ne s'exécutent pas. Analysez le code et le message d'erreur que vous seront donnés, en générant un rapport de débogage avec une analyse textuelle expliquant les raisons qui causent les erreurs d'exécution indiquées par le message d'erreur. Enfin, proposez des solutions, délimitées par '"'python' au début et '"' à la fin, que l'étudiant peut implémenter dans le code pour le corriger afin qu'il s'exécute correctement sans erreurs."

10.2 Post-processing

The post-processing is similar to the one in *%ai_explain*. However, when the users pass a code which does not present an error message when executed, a message is displayed informing the users and recommending the use of *%%ai_question* should they want to ask questions about codes that run but give unexpected results. Below is the same snippet shown in the Prompt section above, now complete.

```
try:
    number_cell = int(line.strip())
    cell_content = self.ip.user_ns['In'][number_cell]
    clean_cell = cell_content.replace('{', '{{').replace('}', '}}')
    cell_out = self.ip.run_cell(self.ip.user_ns['In'][number_cell])
    clear_output(wait=True)
    if not cell_out.success:
        error_message = f"error_before_exec='{cell_out.error_before_exec}',
           error_in_exec='{cell_out.error_in_exec}'"
    else:
        display(HTML("<h4>Le code fourni ne presente aucun message d'erreur
           !</h4>"))
        display(Markdown("Si vous avez des doutes sur la logique utilisee
           dans le code ou sur les resultats generes, vous pouvez utiliser
           la commande `%%ai_question` pour poser des questions specifiques.
           "))
except:
    clean_cell = cell.replace('{', '{{').replace('}', '}}')
    cell_out = self.ip.run_cell(cell)
    clear_output(wait=True)
    if not cell_out.success:
        error_message = f"error_before_exec='{cell_out.error_before_exec}',
           error_in_exec='{cell_out.error_in_exec}'"
        initial_response = "Note : Apres avoir corrige le bug, afin de bien
           executer le contenu de cette cellule plutot que d'en generer un
           rapport de debogage, vous devez supprimer la commande magique `%%
           ai_debug` sur la premiere ligne.\n\n"
    else:
        display(HTML("<h4>Le code fourni ne presente aucun message d'erreur
           !</h4>"))
        display(Markdown("Si vous avez des doutes sur la logique utilisee
           dans le code ou sur les resultats generes, vous pouvez utiliser
           la commande `%%ai_question` pour poser des questions specifiques.
           "))
        display(Markdown("Pour bien executer le contenu de cette cellule,
           vous devez supprimer la commande magique `%%ai_debug` sur la
           premiere ligne."))
```

11 *%ai_quiz* command

11.1 Prompt

The prompt used by the *%ai_quiz* command is as shown below, where instructions_prompt consists of the instructions for the quiz generation.

```
self.quiz_messages = [{"role": "system", "content": self.persona_prompt_quiz
    }, {"role": "user", "content": instructions_prompt}]
```

The persona_prompt_quiz, declared in the config_llm function, is the prompt responsible for optimizing the LLM for a Quiz Generation task, defining the context of the command it will receive and explaining the expected answer format. The variable cours_content receives the ensemble of course extracts (cells) to be used as reference for the course generation at every query, and it can be either an entire notebook or just the relevant cells with respect to the chosen subject. The quiz_level variable is meant to improve the prompt engineering by trying to better define the expected difficulty level, since "facile", "moyen", and "difficile" can be somewhat abstract. Therefore, by creating a parallel with french higher education levels, the expected difficulty becomes more clear. The prompt is split into different strings just for organization purposes, where the variables are used in the f-strings, and the formatting instructions (which uses double quotes) is used in the triple-quoted string.

```
# If student chooses a specific subject for the quiz
# (will be used in the instructions_prompt inside the ai_quiz function)
self.quiz_subject = ""
# The relevant cours content (updated by get_quiz_context at every query)
self.cours_content = ""
# Prompt engineering, to help LLM to understand expected difficulty level
if(self.quiz_difficulty == "facile"):
    self.quiz_level = "licence L1"
elif(self.quiz_difficulty == "moyen"):
    self.quiz_level = "licence L3"
elif(self.quiz_difficulty == "difficile"):
    self.quiz_level = "master M2"
```

self.persona_prompt_quiz = f"Vous êtes un générateur de quiz style QCM de niveau de difficulté {self.quiz_difficulty} pour des étudiants de {self.quiz_level} qui souhaitent tester leurs connaissances dans le cadre de leurs études et de leur préparation aux examens. Le quiz a {self.num_questions_quiz} questions en français, où chaque question n'a qu'une seule réponse correcte et trois réponses incorrectes. Essayez de rédiger les quatre choix de réponses de longueur similaire pour éviter que la bonne réponse soit toujours la plus longue." + """ Organisez les questions générées dans une liste de dictionnaires python, où chaque dictionnaire représente une question formatée comme suit : ["énoncé": "Question à l'élève (1) ?", "bonne_réponse": "Ceci est la réponse correcte à la question 1.", "mauvaises_réponses":["Ceci est la première réponse incorrecte à la question 1.", "Ceci est la deuxième réponse incorrecte à la question 1.", "Ceci est la troisième réponse incorrecte à la question 1."], "énoncé": "Question à l'élève (2) ?", "bonne_réponse": "Ceci est la réponse correcte à la question 2.", "mauvaises_réponses":["Ceci est la première réponse incorrecte à la question 2.", "Ceci est la deuxième réponse incorrecte à la question 2.", "Ceci est la troisième réponse incorrecte à la question 2."], ...]."""
+ f" Cours à utiliser pour générer le quiz : {self.cours_content} "

The instructions_prompt, on the other hand, is declared within the ai_quiz function itself. Its main purpose is to reinforce the instructions given by the persona_prompt_quiz, in addition to introducing the course extracts to be used to generate the quiz. In this sense, there are to versions of this prompt, one for a quiz about a specific notebook and another for a quiz about a specific subject (which may contain cells from different notebooks). In this second case, the subject itself can also be passed to the LLM alongside the cells, allowing the model to ask questions about related aspects that may not explicitly appear in the given extracts. The choice between the two prompts is based on whether the get_quiz_context function received a subject for which course extracts were found in the class repository.

```
if(self.quiz_subject != “”):

    instructions_prompt = f“Générez {self.num_questions_quiz} questions en français style QCM
    de niveau de difficulté {self.quiz_difficulty} (niveau {self.quiz_level} à l’université) sur le
    cours de {self.quiz_subject} donné. Basez vos questions principalement sur les contenus
    liés à {self.quiz_subject} couverts par le cours susceptibles d’être abordés lors d’examens
    futurs. N’hésitez pas à poser des questions sur {self.quiz_subject} concernant des détails
    qui n’apparaissent pas explicitement dans le cours mais que l’étudiant devrait connaître,
    à condition qu’elles soient pertinentes et au même niveau de difficulté. N’oubliez pas de
    bien séparer les éléments des dictionnaires et de la liste par des virgules conformément aux
    instructions données et aux standards python. Ne retournez rien d’autre que la liste des
    dictionnaires python dans le bon format avec les questions QCM en français sur le sujet
    {self.quiz_subject}.”

else:

    instructions_prompt = f“Générez {self.num_questions_quiz} questions en français style QCM
    de niveau de difficulté {self.quiz_difficulty} (niveau {self.quiz_level} à l’université) sur le
    cours donné. Basez vos questions principalement sur les contenus liés à l’Introduction à la
    Science des Données (statistiques, probabilités, mathématiques, programmation python, ap-
    prentissage statistique, etc) couverts par le cours susceptibles d’être abordés lors d’examens
    futurs. N’hésitez pas à poser des questions sur le même sujet concernant des détails qui
    n’apparaissent pas explicitement dans le cours mais que l’étudiant devrait connaître, à con-
    dition qu’elles soient pertinentes et au même niveau de difficulté. N’oubliez pas de bien
    séparer les éléments des dictionnaires et de la liste par des virgules conformément aux in-
    structions données et aux standards python. Ne retournez rien d’autre que la liste des
    dictionnaires python dans le bon format avec les questions QCM en français sur le cours
    donné.”
```

The get_quiz_context function is responsible for processing the compilation of course extracts to be passed to the LLM. There are three possible use cases: when the user passes the *%ai_quiz* command by itself and the function returns all the cells from a random notebook; when the user passes the *%ai_quiz* command alongside the name of a notebook and the function returns all the cell from that specific notebook; and when the user passes the *%ai_quiz* command alongisde a class-related subject and the function returns a compilation of cells that are relevant to the chosen subject. To determine whether the subject is class-related and whether the cells are relevant to it, the DPR model is used with a minimum threshold for the cosine similarity (therefore, the user must chose DPR as RAG model in the initialization, the quiz generation is not available with BM25). For the class-related aspect, this minimum value was empirically set to 0.35 and cannot be changed by the user. To the relevance aspect, the threshold is defined by the variable quiz_min_relevance which can be modified by the user during the initialization. If the model cannot find at least 3 cells with a cosine similarity that satisfies the threshold, the function returns None and the *%ai_quiz* command will inform the user that it was not possible to produce the quiz about the given subject. Note that, as mentioned, the get_quiz_context function sets the quiz_subject variable that defines which version of instructions_prompt will be used.

```
def get_quiz_context(self, query):
    # For command with no argument
    if(query == ""):
        file_list = self.df.file.unique().tolist()
        file_list_filtered = [file for file in file_list if file != "
            CM1_1_presentation_UE.md"]
        notebook = random.choice(file_list_filtered)
        self.quiz_subject = ""
        return "\n".join(self.df[self.df["file"] == notebook]["content"].
            dropna())
    # For %ai_quiz CMx
    elif(query in self.df.file.unique().tolist()):
        self.quiz_subject = ""
        return "\n".join(self.df[self.df["file"] == query]["content"].dropna
            ())
    # For %ai_quiz CMx.md
    elif(query in [x.replace(".md", "") for x in self.df.file.unique().
        tolist()]):
        self.quiz_subject = ""
        return "\n".join(self.df[self.df["file"] == query+".md"]["content"].
            dropna())
    # For %ai_quiz subject
    else:
        # The quiz for specific subject only works with dpr
        if self.rag_model != "dpr":
            return "error_rag"
        else:
            # For now, check if there are at least three cells with cos_sim
                >= threshold to see if subject is class-related
            check_subject_relevance = self.dpr_search(query, 0.35, 5)
            if(len(check_subject_relevance) > 3):
                index_high_score = self.dpr_search(query, self.
                    quiz_min_relevance, self.quiz_max_results)
                self.quiz_subject = query
                return "\n".join(self.df["content"].iloc[index_high_score])
            return None
```

11.2 Post-processing

One of the key aspects in the user experience of the *%ai_quiz* command is the display of the questions as soon as the first one is ready, opposed to waiting for the generation of all questions before displaying them. It is worth noticing that, for now, only the Aristote models support this feature in the toolkit, whereas for the Hugging Face model the LLM needs to generate all questions before they can be displayed. To enable the display since the generation of the first question is complete, the Thread method from the threading package is used to run the LLM generation in the background while the code is locked in the widgets display function (display_question).

```
from threading import Thread
self.questions_llm = []
if self.model_name == "aristote/llama" or self.model_name == "aristote/
   mixtral":
    Thread(target=self.generate_quiz_questions_aristote).start()

elif self.model_name == "huggingface/llama":
    clear_output(wait=True)
    display(HTML("<h4>Chargement des questions, veuillez patienter...</h4>")
       )
    display(HTML("<h5>Les modeles Hugging Face ne permettent pas encore d'
       afficher les questions des qu'elles sont pretes, il faut donc
       attendre que toutes les questions soient generees.</h5>"))
    generated_text = self.generate_quiz_questions_huggingface()
    clear_output(wait=True)

self.display_output = display(Markdown(""), display_id=True)
self.out = widgets.Output()
display(self.out)
# Start the interactive loop
self.display_question(0)
```

In the background, the LLM generates a string which is analysed after each token is appended. Whenever a new question is identified (based on the format of a dictionary delimited by curly brackets, as instructed in the prompt), it is then appended to a global list (questions_llm) with the alternatives shuffled, which the display_question function can access. If there is a issue with the generated question, it is then discarded and the number of expected questions is reduced by 1.

```
def generate_quiz_questions_aristote(self):
    chunks = ""
    for chunk in self.model.stream(self.quiz_messages):
        chunks += chunk.content
        if "{" in chunks and "}" in chunks:
            init = chunks.find("{")
            end = chunks.find("}") + 1
            dict_string = chunks[init:end]
            try:
                true_dict = json.loads(dict_string)
                answers = [true_dict["bonne_reponse"]] + true_dict["
                   mauvaises_reponses"]
                random.shuffle(answers)
                true_dict["toutes_reponses"] = answers
                self.questions_llm.append(true_dict)
            except:
                self.num_questions -= 1
            chunks = ""
```

Therefore, for each question index (in the quiz initialization the passed value is 0), the display_question is locked in a while loop waiting for this global list to receive the next question (at least for the Aristote model, since in the Hugging Face case all the questions will have been previously generated before the display of the widgets so the loop never locks the code). When the following question has been appended to the global list, the loading message is then replaced by the actual question alongside Previous and Next buttons. Note that, if the students take some time reading the question, there is a good chance that the next question will already have been generated when they click the Next button. If all the questions fail, the function displays an error message.

```
def display_question(self, index):
    if index >= self.num_questions:
        self.finish_quiz()
        return
    with self.out:
        clear_output(wait=True)
        if len(self.questions_llm) <= index:
            display(HTML("<h4>Chargement de la question, veuillez patienter
                ...</h4>"))
            while(len(self.questions_llm) <= index):
                if(self.num_questions <= 0):
                    display(HTML("<h4>La generation du quiz a echoue.
                        Veuillez relancer cette cellule.</h4>"))
                    return
                elif index >= self.num_questions:
                    self.finish_quiz()
                    return
        question = self.questions_llm[index]["enonce"]
        right_answer = self.questions_llm[index]["bonne_reponse"]
        answers = self.questions_llm[index]["toutes_reponses"]
        clear_output(wait=True)
        display(Markdown(f"<h4>Question {index+1} /{self.num_questions} :
                {question} </h4>"))
        radio_buttons = widgets.RadioButtons(
            options=answers,
            value=self.selected_answers[index],  # Selected value
            description="Choisissez l'option correcte:",
            disabled=False,
            layout={'width': 'max-content'},
        )
        display(radio_buttons)
        if index > 0:
            button_back = widgets.Button(description='Precedent', disabled=
                False)
        else:
            button_back = widgets.Button(description='Precedent', disabled=
                True)
        button_back.style.button_color = 'lightblue'
        if index < (self.num_questions - 1):
            button_next = widgets.Button(description='Suivant')
        else:
            button_next = widgets.Button(description='Finaliser')
        button_next.style.button_color = 'lightgreen'
        print("\n")
        button_box = widgets.HBox([button_back, button_next])
        display(button_box)

        def on_button_back_click(b):
            self.selected_answers[index] = radio_buttons.value
            self.display_question(index - 1)

        def on_button_next_click(b):
            self.selected_answers[index] = radio_buttons.value
            self.display_question(index + 1)

        button_back.on_click(on_button_back_click)
        button_next.on_click(on_button_next_click)
```

Note in the code above that, if the index is larger than the number of questions, that means that the user has answered the final question and the finish_quiz function is called. This function provides a performance overview, showing the number of correct answers and the alternative that should have been selected for the questions that were incorrectly answered.

```
def finish_quiz(self):
    final_answers = self.questions_llm.copy()
    number_correct = 0
    with self.out:
        clear_output(wait=True)
        display(HTML("<h2>Quiz termine !</h2>"))
        for i in range(len(final_answers)):
            final_answers[i]["reponse_selectionnee"] = self.selected_answers
                [i]
            if final_answers[i]["reponse_selectionnee"] == final_answers[i][
                "bonne_reponse"]:
                 number_correct += 1
                 display(HTML(f"<h4>Question {i+1} : {final_answers[i]['enonce']}
                     </h4>"))
                 display(Markdown(f"**Reponse Selectionnee :**" +
                     final_answers[i]['reponse_selectionnee']))
                 print("\n")
            else:
                 display(HTML(f"<h4>Question {i+1} : {final_answers[i]['enonce']}
                     </h4>"))
                 display(Markdown(f"**Reponse Selectionnee :**" +
                     final_answers[i]['reponse_selectionnee']))
                 display(Markdown(f"**Reponse Correcte :**" + final_answers[i
                     ]['bonne_reponse']))
                 print("\n")
        display(HTML(f"<h4>Votre nombre de reponses correctes est de
              {number_correct} /{self.num_questions} .</h4>"))
        print("\n")
        button_feedback = widgets.Button(description='Feedback')
        button_feedback.style.button_color = 'lightgreen'
        display(button_feedback)
        button_feedback.on_click(on_button_feedback_click)
```

Finally, should the user click the Feedback button below the performance overview, the LLM is invoked in within the on_button_feedback_click function in order to generate a textual analysis of the answers given by the student, discussing the questions and explaining the answers. The prompt for the feedback generation is shown below.

prompt_feedback = f"Générez un feedback pour un étudiant à partir des réponses qu'il a données à un quiz de type QCM. Organisez votre feedback en rappelant toujours les questions et les réponses lorsque l'étudiant n'a plus accès au quiz, en expliquant pourquoi la bonne alternative est correcte et pourquoi les mauvaises alternatives sont incorrectes. Si l'élève a choisi l'alternative incorrecte, renforcez votre explication sur la source possible de la confusion qui l'a conduit à se tromper dans la question et soulignez, parmi les alternatives disponibles, celle qui est correcte. Récapitulatif du quiz : {final_answers} "