Etude des outils d’exploration de corpus :

Recherche basée sur le nombre d’occurrence, fournit ensuite à ChatGPT :

* Fonctionne assez bien pour récupérer les documents pertinents, mais ne permet pas de savoir combien de documents doivent être gardé dans la requête (la valeur du cosinus peut beaucoup évoluer d’une requête à une autre). Problème lié au nombre de tokens que l’on peut donner à GPT3.5 (16.000 max), que l’on peut vite dépasser. Il faudrait alors faire un pipeline pour fournir en contexte les discussions précédentes (ce qui augmente donc le nombre de tokens utilisés et multiplie le nombre de « calls » de l’API). Les réponses sont cependant satisfaisantes, et c’est une solution à la fois facile à mettre en œuvre et rapide (4 sec pour obtenir les doc les plus pertinents + temps de traitement de l’API openai).

Recherche basée sur les embeddings, fournit ensuite à ChatGPT :

* Peut avoir de meilleur résultats que la recherche par occurrence (semantic search), mais problème dans la construction des embeddings, lié notamment aux documents très court (très peu de texte, ou NA à la place d’une partie du texte). Je n’ai pas encore pu tester le fonctionnement de cette solution car la récupération des documents pertinents n’est pas encore réussie, mais on pourra peut-être plus facilement sélectionner les documents pertinents sur la valeur du cosinus, car il y a une plus forte homogénéité des valeurs. On peut ainsi imaginer un « threshold » (pallier) à partir duquel on conserve les documents que l’on envoie à l’API ChatGPT pour résumer.

Dans ces deux premiers cas, on est limité par le nombre de tokens, donc il faudrait trouver des stratégies pour réduire le nombre de tokens (résumé de résumé ?) ou bien d’introduire les réponses précédentes (mais augmente le prix par requête).

Solution proposée par LangChain :

* Application (python) qui semble très adaptée à notre problème, et qui peut utiliser d’autres API que celle d’openai (Llama). D’une certaine manière, c’est le même travail qui est réalisé : sélection des documents pertinents, et interrogation à l’aide d’une requête. Mais on peut utiliser une seule et même requête car on interroge des embeddings, et la sélection des documents est faite par le programme, même si on ne sait pas précisément comment c’est réalisé (voir code source) [Note : On pourrait avoir une seule requête avec la technique ci-dessus].
* Le gros avantage est une application pensée pour ce type de travaux, et qui évite de tout programmer « à la main ». On cherche une application type « Q&A » qui travaille sur les données qu’on lui donne en entrée, ou sur un système de chat dont on modère fortement le prompting.
* <https://python.langchain.com/docs/use_cases/question_answering/>
* https://community.openai.com/t/strategies-to-reduce-conversational-gpt-4-costs/198413

Dans tous les cas : prompt ingeneering :

* C’est en travaillant au mieux les prompts que l’on pourra avoir les meilleures réponses :
* <https://wfhbrian.com/chatml-new-prompt-format-for-the-chatgpt-api/>
* <https://github.com/openai/openai-cookbook/blob/347e54c76ce969eac3495427fe003fc6626765d0/examples/How_to_format_inputs_to_ChatGPT_models.ipynb>
* <https://github.com/langchain-ai/langchain/blob/master/docs/extras/use_cases/question_answering/semantic-search-over-chat.ipynb>

L’avantage en particulier est que l’on peut utiliser une large variété de modèle en initialisant un objet llm. Ex :

llm = LlamaCpp(

model\_path="/Users/rlm/Desktop/Code/llama.cpp/llama-2-13b-chat.ggmlv3.q4\_0.bin",

n\_gpu\_layers=n\_gpu\_layers,

n\_batch=n\_batch,

n\_ctx=2048,

f16\_kv=True, # MUST set to True, otherwise you will run into problem after a couple of calls

callback\_manager=callback\_manager,

verbose=True,

)

Ou bien :

from langchain.chat\_models import ChatOpenAI

from langchain.llms import OpenAI

llm = OpenAI()