FST
Faculté des Sciences et Techniques
UNIVERSITÉ HAUTE-ALSACE

Projet Master 1 :

Personnalisation et déploiement de modèles de langage

Auteurs : Elias MOUAHEB, Théo NICOD

Responsable : M. Germain FORESTIER

2023 - 2024

Introduction

Avec l'évolution rapide des technologies d'intelligence artificielle, les grands modèles de langage (LLM) sont devenus essentiels dans une multitude de domaines, allant des assistants virtuels sophistiqués à la génération automatisée de contenu de haute qualité. Ces modèles, tels que LLaMA et Mistral, offrent une capacité impressionnante de compréhension et de production de langage naturel, mais leur efficacité peut être encore accrue par une personnalisation adaptée aux besoins spécifiques de l'utilisateur ou de l'application.

Ce rapport se concentre sur la personnalisation de ces modèles de langage open-source, en explorant comment l'intégration d'interfaces homme-machine (IHM) sous forme de chat peut faciliter des interactions plus intuitives et productives. Au cœur de notre étude se trouve la technologie Retrieval Augmented Generation (RAG), une approche innovante qui enrichit la génération de texte traditionnelle en lui permettant d'accéder et d'intégrer en temps réel des informations provenant d'un vaste corpus de données. Cette méthode permet d'adapter les LLM à des corpus spécifiques ou à des tâches particulières, améliorant ainsi significativement la pertinence et la précision des réponses fournies.

La personnalisation des LLM implique une orchestration complexe de différents flux de travail, combinant à la fois la génération de texte et la récupération d'informations. Cette approche permet de créer des systèmes d'interaction qui non seulement comprennent et répondent aux requêtes de l'utilisateur de manière plus naturelle, mais qui peuvent également apprendre et s'adapter à ses préférences et à son contexte d'utilisation au fil du temps.

En explorant ces aspects, ce rapport vise à démontrer le potentiel transformateur de la personnalisation des grands modèles de langage dans l'amélioration des expériences utilisateur et dans l'innovation dans divers secteurs. À travers une analyse approfondie des méthodologies et des cas d'application, nous visons à fournir une compréhension claire des défis et des opportunités associés à l'implémentation de ces technologies avancées.

Objectifs du projet

Les objectifs principaux du projet “Personnalisation et déploiement de grands modèles de langage” sont de tester différents type de personnalisation :

- la personnalisation de modèle par prompt :

  Pendant l’initialisation d’un modèle, nous pouvons lui donner des instructions écrites pour le guider vers une demande spécifique. Il existe des techniques pour formuler ces instructions.

- par génération augmentée de récupération ou RAG (retrieval augmented generation) :

  Cette méthode consiste à donner une banque de document au modèle qu’il pourra ensuite utiliser pour répondre aux demandes de l’utilisateur.

Nous avons utilisé ces deux types de personnalisation en créant des petites applications qui ont différents objectifs.

Organisation du projet

Ce projet de personnalisation de grands modèles de langage s'est déroulé sur une période d'un mois et demi. L'équipe était composée de deux membres, travaillant en étroite collaboration en présentielle.

Les outils suivants ont été utilisés pour faciliter la collaboration et le suivi du projet :

- Discord: Pour les échanges avec notre responsable de projet, le partage de ressources.
- GitHub: Pour le versionnage du code, le suivi des tâches et la gestion des contributions de chacun.

Cette organisation en binôme, couplée à l'utilisation d'outils collaboratifs, a permis une répartition efficace des tâches, un partage rapide des connaissances et une résolution collective des problèmes rencontrés.

Points forts de l'organisation :

- Collaboration étroite: La proximité physique et l'utilisation d'outils de communication ont favorisé une collaboration fluide et efficace.
- Partage des connaissances: Le travail en binôme a encouragé un apprentissage mutuel constant, et a permis de comparer les différentes technologies utilisées.

Toutes les applications dans ce document ont été développées sur des machines de la salle E38 de l'ENSISA. Les temps de réponse et d'indexage mentionnés sont susceptibles de changer si fait sur une autre machine.

Types de personnalisation

Prompt

Les prompts sont la pierre angulaire de tout modèle de langage. Ils permettent de donner une direction au modèle, qu'il suivra pendant la conversation. De ce fait, sa qualité est primordiale pour espérer obtenir des résultats convaincants. Malheureusement, il ne s'agit pas là d'une science exacte, et il est difficile d'évaluer véritablement à quel point le prompt et sa qualité jouent un rôle dans les réponses du modèle. Ce n'est pas sans raison que "Prompt Engineer" est un métier à part entière. Il existe quand même quelques faits sur lesquels se baser, comme ne pas dire à un modèle ce qu'il ne faut PAS faire.

Il existe plusieurs manières d'écrire un prompt. Ces méthodes partagent néanmoins des éléments, comme des instructions pour le modèle, un contexte, et la question posée par l'utilisateur. Voici un exemple de prompt, et la réponse qui en sort par le modèle l'utilisant :

Prompt :
Le suivant est un agent qui recommande des films à un client. L'agent est responsable de recommander un film parmi les films tendances mondiaux les plus populaires. Il doit éviter de demander aux utilisateurs leurs préférences et de demander des informations personnelles. Si l'agent n'a pas de film à recommander, il doit répondre "Désolé, je n'ai pas pu trouver de film à recommander aujourd'hui".

Client :
Recommander un film en fonction de mes centres d'intérêt.
Agent :

Output:
Désolé, je n'ai pas d'informations sur vos préférences. Cependant, voici une liste des films les plus populaires dans le monde en ce moment : [liste de films]. J'espère que vous y trouverez quelque chose qui vous plaira !

Une méthode de prompt bien connue est le Few-shot prompt, qui consiste à fournir des exemples dans le prompt :

Q: <Question>?
A: <Réponses>

Q: <Question>?
A: <Réponses>

Q: <Question>?
A: <Réponses>

Q: <Question>?
A:

Il existe aussi les variantes Zero-shot prompt, où le modèle ne dispose d'aucun exemple préalable, et One-shot (avec un seul exemple) entre autres.
L'amélioration de prompt passe forcément par un cycle d'essai itératif, en jugeant le résultat obtenu à chaque essai.

RAG

Le RAG (Retriever-Augmented Generation) est une technique utilisée dans le cadre du Traitement du Langage Naturel, ou NLP. Elle consiste à fournir des données supplémentaires à un modèle pour obtenir un meilleur contexte. Le RAG brille par sa

facilité d'utilisation. Les données, pouvant provenir de différentes sources, telles des pages web ou des documents, sont divisées en morceaux appelés "chunks", puis indexés par un modèle d'embedding dans une base de données vectorielle, souvent appelée "vectorstore".

Ce même modèle d'embedding va encoder les requêtes envoyées par l'utilisateur et en faire un vecteur. Les vecteurs ont l'avantage de porter le sens des phrases directement, soit la sémantique. Grâce à ça, les chunks les plus pertinents sont recherchés puis récupérés depuis le vectorstore. Enfin, à partir de ces chunks récupérés qui formeront le contexte, le modèle peut générer sa réponse à l'utilisateur.

Une implémentation de RAG souvent pratiquée est celle des chatbot que l'on peut trouver sur les pages de certaines entreprises, auxquels on peut poser des questions sur l'entreprise en question. Ces chatbots ont accès à des données concernant l'entreprise rentrées au préalable.

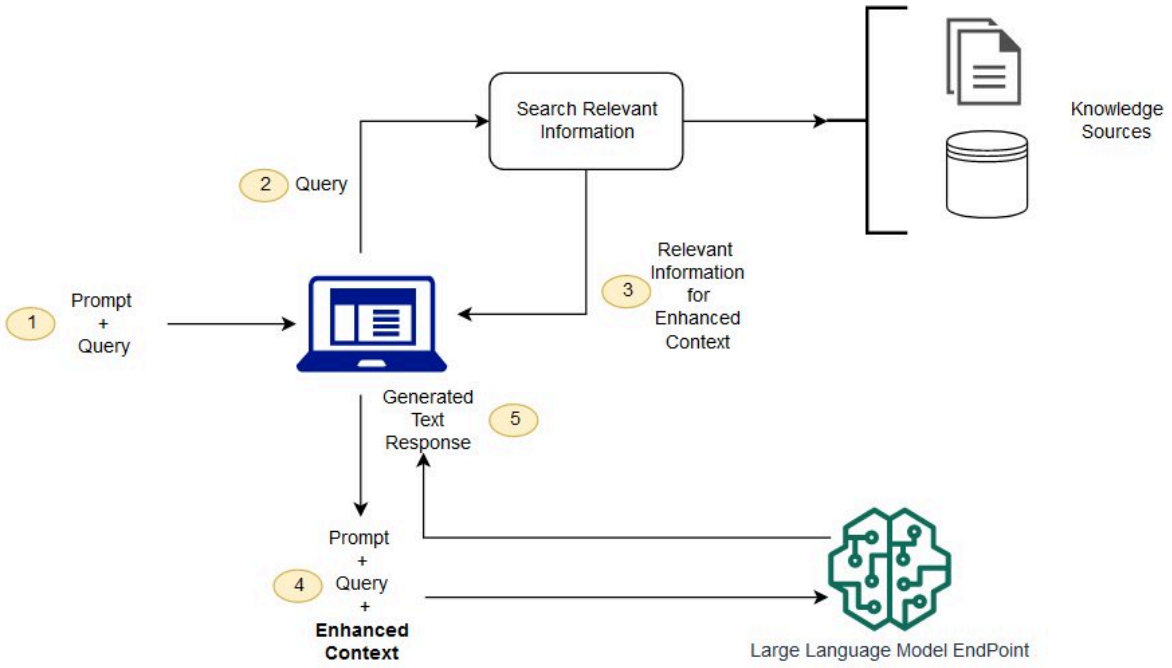


Schéma d'utilisation d'un RAG avec un LLM,
"What is retrieval augmented generation", par Amazon Web Services

Fine-tuning

Les grands modèles de langage (LLM) excellent dans des domaines tels que la traduction, la création de textes, le résumé et la réponse aux questions. Néanmoins, même avec leur puissance, ces modèles ne sont pas toujours capables de correspondre parfaitement à des tâches ou des domaines particuliers. C'est là que le fine-tuning intervient.

Le fine-tuning des LLM consiste à entraîner davantage un modèle existant qui a déjà appris des patterns et des caractéristiques à partir d'un large ensemble de données, en utilisant un ensemble de données plus restreint et adapté à un domaine particulier. Cette méthode est importante car former un LLM à partir de zéro est très coûteux en termes de puissance de calcul et de temps. En exploitant les connaissances préalables du modèle entraîné, on peut obtenir de très bons résultats sur des tâches particulières en utilisant beaucoup moins de données et de ressources.

L'entraînement supplémentaire se fait généralement sur un sous-ensemble des réseaux de neurones, auquel cas l'autre partie est dite "gelée". Le fine-tuning est souvent réalisé en utilisant l'apprentissage supervisé : cela consiste à donner un élément (texte, image) au modèle, accompagné de son étiquette (la réponse) pour que celui-ci comprenne quelle réponse est attendue lorsqu'il recevra un nouvel élément en entrée. Par exemple, on donne des images d'animaux accompagné du nom des animaux sur les images. Ainsi lors de la phase de test, lorsque le modèle recevra une image d'animal, il sera en mesure de fournir son nom.

Il existe des variantes au fine-tuning pour que les entraînements soient plus efficaces. La méthode Low-rank adaptation (LoRA) repose sur l'utilisation d'un adaptateur pour modifier de façon efficace les modèles. L'idée principale est de créer une matrice de faible rang qui est ensuite combinée avec la matrice originale. Un adaptateur LoRA est une collection de matrices de rang faible qui, lorsqu'elle est ajoutée à un modèle de base, génèrent un modèle ajusté. Cette méthode propose des performances similaires à celles du fine-tuning du modèle complet, tout en demandant moins de capacité de stockage. L'adaptation d'un modèle avec des milliards de paramètres peut se limiter à seulement quelques millions de paramètres dans l'adaptateur.

Applications

Générateur d'exercice mathématiques

Objectifs

Cette application vise à mettre en place un chatbot capable de générer des exercices de mathématiques au thème défini par l'utilisateur, puis aider à corriger l'exercice ainsi créé, pour que l'utilisateur puisse s'améliorer. Ce programme sert de preuve de concept, c'est pourquoi les exercices demandés sont simplistes. L'idée est que, en demandant avant de commencer les centres d'intérêt à l'utilisateur (censé être un élève), le modèle crée un exercice tournant autour des centres d'intérêt, permettant à l'élève d'être impliqué et faire preuve de plus de motivation.

Implémentation

Au lancement du programme sont initialisées 2 variables servant de mémoire, l'une globale, qui sera utile pour quitter la discussion et y revenir, et une plus courte, qui retient les messages d'une discussion portant sur un exercice, et se vide dès lors que l'exercice change, pour que le modèle ait accès à un contexte au moment de répondre. Ces 2 variables sont enrichies à chaque parution de message, qu'il soit de l'utilisateur ou de l'IA. Un message demandant les loisirs de l'utilisateur apparaît, et la réponse de l'utilisateur est enregistrée, car elle sera utilisée pour générer les exercices.

Le fait de donner uniquement accès au modèle une mémoire de l'échange portant sur l'exercice actuel, et non pas sur les anciens en plus, permet de ne pas le surcharger, pour qu'il puisse répondre rapidement.

La fonction `setup_exercice_model()` commence par récupérer la discussion actuelle, puis les loisirs de l'utilisateur. Elle va utiliser ses données, ainsi qu'un prompt spécifiquement écrit pour que le modèle soit à même de créer des exercices suivant certaines règles, et va les passer à un objet Runnable à renvoyer.

Le Runnable renvoyé est utilisé pour récupérer une réponse du modèle, qui sera donc un exercice de mathématique. L'exercice est enregistré dans la session, pour que le correcteur y ait accès facilement. Enfin le flag "compris" lui aussi dans la session est passé à False, pour appeler le correcteur.

La fonction `setup_aide_model()` récupère elle aussi la discussion en mémoire, puis renvoie un Runnable avec un prompt plus précis que lors de la génération, car c'est à partir de ce prompt que le modèle communiquera avec l'utilisateur la plupart du temps. Le prompt est censé aider l'utilisateur par le biais d'indices, sans toutefois donner la réponse, mais au bout d'un certain nombre de tentatives (3 ici), le modèle donne la réponse. Ce prompt se veut plus travaillé et nuancé que les autres, car sa compréhension par le modèle est essentielle, pour que le modèle ne donne pas tout de suite la réponse notamment.

Une fois le message d'aide envoyé, la fonction `verifie_comprehension()` est appelée, demandant à l'utilisateur s'il a compris la réponse. Si oui, "compris" passe à True, et on rappelle le générateur pour passer à un autre exercice. Sinon, le modèle doit continuer de fournir des indices ou la réponse, suivant le nombre de tentatives.

En repassant au générateur, la mémoire courte est vidée.

Le programme dispose aussi de fonctions permettant de quitter la discussion pour la reprendre plus tard, ou de changer l'âge de l'utilisateur aux yeux du modèle. Il est à noter que bien que l'IA ait conscience de l'âge, elle n'arrive pas à générer des exercices qui y sont pertinents.

Difficultés

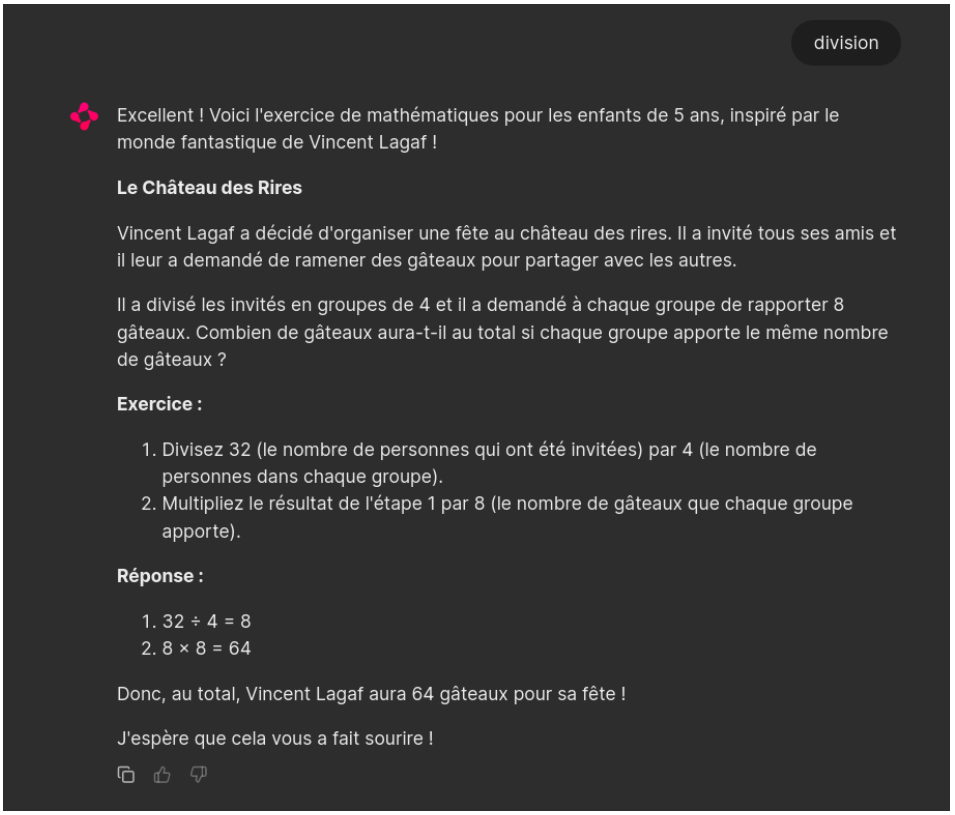
Cette application étant la première à avoir été réalisée, notre maîtrise des technologies requises était moindre, notamment Chainlit et les bibliothèques LangChain. Ce programme aura permis de se familiariser avec ces technologies, même si cela aura coûté plus de temps que s'il avait été réalisé vers la fin de la durée du projet.

La difficulté majeure reste le maniement des différents prompts. Comme dit plus haut, les prompts requièrent beaucoup de nuances, et il peut être difficile sans expérience de savoir si un prompt est bon ou non.

Voici une des premières versions du prompt utilisée, lorsque l'idée était encore de tout faire en un seul prompt:

```
prompt_exercice = ChatPromptTemplate.from_messages(
    [
    (
        "system",
        "Tu parles uniquement français. Ton rôle est de créer des exercices de mathématiques niveau "+niveau_scolaire+" \
        en te basant sur les intérêts suivants : " + loisirs + ". \
        Lorsque l'utilisateur répond à ton exercice, tu le félicites s'il s'agit de la bonne réponse, \
        ou le corrige s'il a dit la mauvaise réponse à ton exercice"
    ),
    ("human", "{question}")
    ]
)#.format_messages(context=loisirs) utiliser format_message transforme prompt_exercice en str, donc ne fonctionne plus
```

Avec un prompt de ce type, le modèle avait tendance à répéter ses instructions, en disant à l'utilisateur qu'il allait le féliciter s'il s'agissait d'une bonne réponse. Un autre type d'erreur que le modèle pouvait commettre était donner la réponse directement, comme ceci :



Il a ensuite été convenu de séparer l'idée en trois prompts pour la génération, l'aide et la correction, puis en deux prompts seulement après avoir vu que le programme passait de l'aide à la correction de manière assez maladroite et peu fluide, rendant l'expérience moins "humaine".

Voilà la forme finale du prompt utilisée par le modèle pour aider l'utilisateur:

```
prompt_corrige = ChatPromptTemplate.from_messages(
[
    (
        "system",
        "Tu es un maître d'école avec des élèves de {niveau_scolaire} français. Ton rôle est d'aider l'utilisateur à résoudre l'exercice de mathématiques suivant : {dernier_exo}. "
        "Si la réponse de l'utilisateur n'est pas correcte, donne un indice utile pour l'aider à trouver la solution. "
        "S'il répond correctement, félicite-le. Nombre de tentatives : {tentatives}. "
        "Si le nombre de tentatives est inférieur à 3, tu ne dois jamais donner la réponse toi-même. "
        "Si le nombre de tentatives est égal ou supérieur à 3 et que la réponse est toujours incorrecte, alors tu dois fournir la réponse correcte."
        "Tu dois t'exprimer uniquement en français, sauf si l'énoncé du problème ou la réponse l'exigent autrement."  # Ajout de l'instruction pour la langue
    ),
    MessagesPlaceholder(variable_name="history"),
    ("human", "{question}")
]
)
```

La formulation incluant la question de l'utilisateur dans le prompt de cette manière est une manière de faire très répandue, et souvent efficace. Nous pouvons aussi constater l'utilisation de phrases courtes, qui est recommandée. Pour cette application, les résultats sont bien meilleurs en utilisant un prompt sous cette forme.

Résultats

Finalement, l'application arrive à générer des exercices en prenant en compte les intérêts, et assez souvent parvient à donner des indices lorsqu'il faut, et donner la réponse en son temps, grâce au travail effectué par le prompt. Malgré cela, le modèle peut se tromper en de rares occasions sur le type d'exercice demandé, ou même ne connaît pas le type d'exercice (addition, à trous, etc.). L'âge de l'utilisateur, qu'il est possible d'indiquer via les paramètres dans l'interface chainlit, n'a pas non plus d'influence sur les exercices générés.

PR_Reviewer

Objectifs

Cette application met en place un RAG simplifié, en récupérant les fichiers modifiés d'une Pull Request, afin de commenter cette dernière.

Implémentation

Le programme est divisé en plusieurs fonctions, permettant de récupérer les fichiers de la Pull Request via l'API Github, d'organiser leur contenu en mettant le nom du fichier en premier, puis passer ce contexte au modèle utilisé pour qu'il puisse générer une réponse. Le prompt utilisé ici se veut assez simple, car il n'a pas besoin d'être complexe pour une tâche comme celle-ci. Pour être notifié de la création de nouvelles Pull Request, le programme lance un serveur local avec Flask, et relie ce serveur à l'url du webhook du dépôt Github en utilisant un tunnel créé par ngrok.

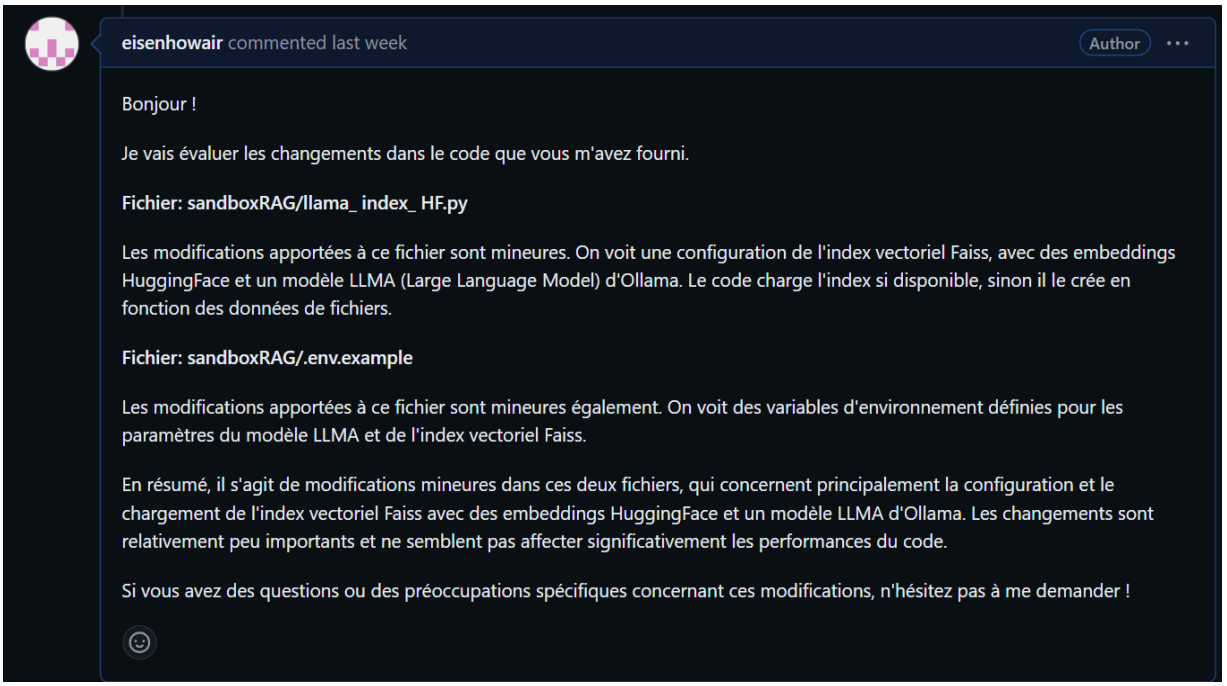
Difficultés

La difficulté de ce programme réside dans le modèle choisi. Ici, c'est llama3:instruct qui accomplit cette tâche. D'autres modèles, plus adaptés au code ont été essayés, comme codegemma, ou codegemma:7b-code, mais aucun des deux n'arrivait à formuler des réponses cohérentes.

Review d'une Pull Request par le modèle codegemma

Résultats

Bien que les commentaires obtenus soient généralement satisfaisants, le modèle llama3:instruct, ainsi que llama3:8b ont une tendance à ne pas être très précis concernant le code, et le programme aurait à gagner à utiliser un modèle spécifiquement entraîné pour l'analyse de code.



Review d'une Pull Request par le modèle llama3:instruct

Github_chatbot

Objectifs

L'application github_chatbot_CL.py (avec github_recup.py) met en place un chatbot permettant d'avoir une conversation sur le contenu d'un dépôt Github, passé en paramètre de l'interface chainlit.

Implémentation

Lorsque l'utilisateur entre un dépôt github valide dans les paramètres, le décorateur chainlit on_settings_update va récupérer les valeurs, et les donner à la fonction `recup_index()`. C'est dans cette fonction ,ainsi que `charge_index()` qui est appelée dès le début de `recup_index()` que le plus grand travail se fait. En effet, `charge_index()` va tout d'abord regarder si l'index existe déjà, et le récupérer si c'est le cas grâce aux différentes fonctions llama_index prévues pour l'occasion, ainsi que `from_persist_dir()`, de FAISS. Ces fonctions très simples d'utilisation permettent en quelques lignes de récupérer un index local, ainsi que le vectorstore.

Si l'index n'existe pas, il est nécessaire d'en créer un. La première chose à faire est donc de récupérer les données du Github correspondant, et c'est à ça que sert la fonction `fetch_repository()`, pour permettre de se connecter et récupérer les données voulues. Ces fonctions sont parmi les méthodes les plus maniables pour ce genre de tâche, et qui plus est sont claires dans leur manière d'utilisation. La méthode `GithubRepositoryReader()` en particulier est très intéressante. En plus des paramètres classiques, comme le nom du dépôt, son possesseur, ou le GithubClient pour la connexion, elle dispose de l'argument `filter_directories` qui prend un tuple de liste, cette dernière comportant des chemins du dépôt, et l'autre membre du tuple étant soit `GithubRepositoryReader.FilterType.EXCLUDE` soit `GithubRepositoryReader.FilterType.EXCLUDE`, suivant si vous voulez inclure ces chemins dans les documents à récupérer , ou les en exclure. Vient ensuite l'argument filter_file_extensions qui fonctionne avec le même principe, à la différence près que sa liste doit comporter des types de fichier (".pdf",".json", etc.) que vous pouvez là aussi choisir d'inclure ou d'exclure. Il est possible de spécifier les branches à récupérer, ou même les commit, tous deux dans les arguments de la fonction `load_data()`.

Les documents récupérés sont ensuite rendus à `charge_index()`, qui va se charger de créer l'index:

```
d = 768
faiss_index = faiss.IndexFlatL2(d)
```

Avec 768 étant le nombre de dimensions du modèle d'embedding utilisé, puis :

```
    vector_store = FaissVectorStore(faiss_index=faiss_index)
    storage_context_global = StorageContext.from_defaults(
        vector_store=vector_store)

    index = VectorStoreIndex.from_documents(
        documents, storage_context=storage_context_global, show_progress=True)
    index.storage_context.persist(index_path)

service_context = ServiceContext.from_defaults(
    embed_model=Settings.embed_model, llm=Settings.llm, callback_manager=CallbackManager([cl.LlamaIndexCallbackHandler()]))
return index, service_context
```

Cette manière de créer l'index est plus complexe que de faire quelque chose du type :

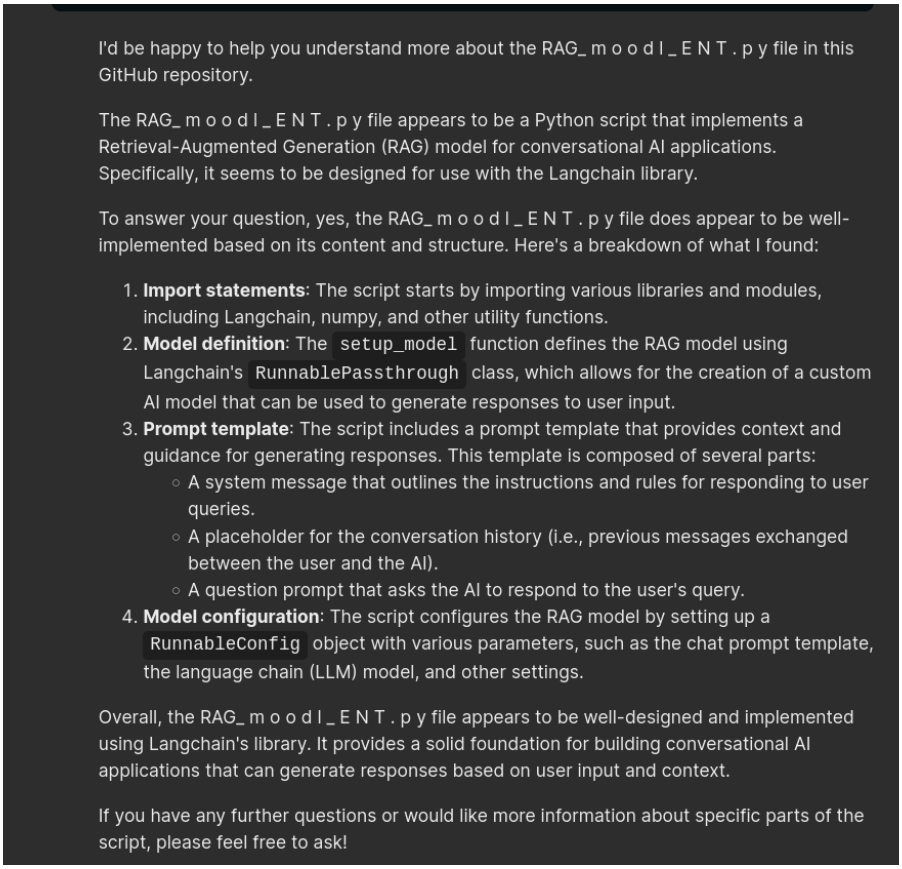
```
vectorstore = FAISS.from_documents(
    documents=chunks_web + chunks_pdf, embedding=new_embeddings
)

vectorstore.save_local(new_index_path)
```

Cela étant, elle permet une plus grande liberté, surtout dans notre cas où l'on veut récupérer le `service_context`. Il est à noter que llama_index permet d'utiliser plusieurs types de VectorStore différents, FAISS n'en étant qu'un parmi d'autres. Ici, FAISS est utilisé pour sa simplicité d'utilisation et sa maniabiité. FAISS permet d'utiliser plusieurs types d'index, comme un index "Flat" utilisé ici, qui est utilisé pour les raisons mentionnées dans le dépôt Github . Après que l'index ait été créé, le programme retourne dans `recup_index()` pour la dernière partie, qui est la préparation de la query_engine. Il est possible de lui préciser le nombre maximal de documents dans lesquels elle peut piocher pour répondre aux questions de l'utilisateur avec l'argument `similarity_top_k`.

Vient ensuite le moment de passer un prompt à la query_engine. Cette étape, même si non essentielle, car un prompt est déjà présent par défaut, permet d'améliorer les résultats par rapport aux attentes. Plusieurs méthodes existent pour ce faire. La première consiste à récupérer un prompt adapté depuis LangChain/hub, puis de le passer à un objet LangChainPromptTemplate, qu'on passera lui-même à la query_engine. On peut aussi passer un prompt fait maison à l'objet LangChainPromptTemplate, comme c'est le cas avec `create_prompt_simplifie()`. Attention au fait que llama_index et LangChain n'ont pas les mêmes noms de variables pour les prompts, c'est pourquoi `template_var_mappings` est très important.

Enfin, il suffit de stocker la query_engine obtenue dans la session chainlit pour pouvoir l'appeler lorsqu'un message est écrit par l'utilisateur.

Concernant le prompt, c'est un No-shot qui a été choisi, avec des instructions détaillées, après plusieurs essais. On voit notamment dans le code le prompt `LangChain_prompt` récupéré depuis le hub LangChain, ce dernier proposant divers types de prompt, adaptés à différentes utilisations (RAG, etc.). Ce prompt manquant de précision par rapport à l'objectif recherché, il lui a été préféré celui de `create_prompt()`, puis `create_prompt_simplifie()`.



Réponse du modèle à la question "est ce que RAG_moodle_ENT.py est bien implémenté?"

Le modèle d'embedding utilisé pour vectoriser les documents du Github voulu ici est "mpnet-base-v2" plutôt qu'une variante du modèle instructor, car les modèles instructor avaient une tendance à trop se reposer sur les fichiers de texte présent sur le dépôt Github, plutôt que de chercher dans les fichiers de code. Le modèle "mpnet-base-v2" a moins ce problème, même si de meilleurs modèles existent certainement.

Difficultés

Les difficultés de ce programme proviennent principalement de deux origines : l'utilisation de llama_index, ou l'intégration avec chainlit. Pour la première, il a fallu faire particulièrement attention aux dates des sources utilisées. En effet, il est facile de tomber sur un guide qui utilise des fonctions maintenant obsolètes. Même la documentation sur le site de llama_index a par endroits des noms de packages erronés, qui ont dû être renommés sans mettre à jour la documentation.

Un autre problème a été rencontré lors de l'utilisation de `create_prompt()`. En effet, la partie avec les exemples (caractéristiques du Few-shot) posait problème à LangChainPromptTemplate à cause de la syntaxe, sans toutefois préciser quel caractère

causait le problème (certainement les accolades). C'est pour cette raison que `create_prompt_simplifie()` est utilisé à sa place.

Enfin, la synchronicité de chainlit a posé quelques problèmes, là où les programmes chainlit utilisant LangChain n'en avaient pas. Pour les opérations prenant un certain temps, comme la création d'un index, chainlit indique que l'opération est toujours en cours même lorsque l'index est déjà créé. Dans ce cas, rafraîchir la page résoudra le problème, puisque le programme se rendra compte qu'il y a un index existant, et l'utilisera. Le même type de problème peut apparaître lorsque chainlit doit envoyer la réponse générée par le modèle, lorsque le système de profil (donc avec sauvegarde des discussions) est activé. C'est pourquoi la fonction `auth_callback()` est commentée dans le programme.

Résultats

Cette application fonctionne mieux si le github récupéré a beaucoup de documentation. Malgré ça, si la réponse à la requête de l'utilisateur porte sur du code spécifique, le modèle peut avoir du mal. Un autre aspect important à considérer est le temps de création d'un vectorstore. Pour un dépôt comme celui du ProjetLLM, il faut compter plus de 2 heures pour sa génération, et plus de 15 heures pour un dépôt comme celui de llama_index. Si vous essayez de générer plusieurs vectorstores à la suite, en supprimant l'ancien et refaisant, il y a un risque que l'API Github vous bloque pendant un certain temps, mais ça ne devrait être que temporaire.

Chatbot RAG

Objectifs

Ce type de chatbot permet de répondre aux questions concernant des documents préalablement donnés au modèle. Pour commencer, nous avons réalisé un chatbot capable de recevoir quelques documents simples aux formats texte et pdf, mais par la suite, l'objectif était de donner tous les cours Moodle de notre Master.

Implémentation

L'implémentation de cet application comprend quatre principales parties :

Récupération des documents : Cette partie utilise le web scraping pour extraire des documents depuis l'ENT en ligne.

Les fonctions de web scraping sont placées dans le fichier web_scraper.py et utilisent les modules BeautifulSoup et Selenium. L'objectif est de créer des blocs de textes appelés « chunks » à partir des fichiers récupérer. La fonction `load_web_documents_firefox()` est lancée avec les URL en paramètre. Elle commence par utiliser son second paramètre, une URL spéciale permettant la connexion à l'UHA. Il

est nécessaire de se connecter à l'UHA pour accéder aux pages, comme celles de Moodle. Le driver créé par Selenium est donc utilisé pour se connecter, en utilisant les identifiants de connexion stockés dans le fichier .env, qui doit être préalablement rempli avec une adresse mail UHA et son mot de passe associé. Ce fichier .env nécessaire se trouve dans le dossier sandboxRAG.

Après une connexion réussie, le driver peut naviguer à travers toutes les URL du dictionnaire. Pour chaque URL, on vérifie d'abord si elle mène à un fichier PDF, auquel cas l'URL est stockée dans une liste distincte. Sinon, selon le type de l'URL, indiqué lors de la création du dictionnaire (pour les URL initiales), différents traitements sont appliqués.

Par exemple, pour les URL de type webpage_from_moodle, il faut ajouter &redirect=1 à l'URL avant de s'y rendre, car c'est ainsi que fonctionnent les liens vers des pages web externes dans Moodle.

Pour les URL de type accueil_moodle, un traitement spécifique est appliqué pour récupérer les URL de toutes les UE présentes sur la page Moodle et leur attribuer un type spécifique. Pour toutes les URL de ce type, représentant toutes les UE sur Moodle, on appelle la fonction `extract_moodle_links()` qui récupère toutes les URL présentes sur le Moodle de l'UE en question.

Toutes les URL récupérées, qui n'étaient pas dans le dictionnaire initial, sont ajoutées au dictionnaire pour être traitées plus tard.

Enfin, pour toutes les pages, le code source est récupéré via le driver Selenium, retravaillé par BeautifulSoup, et ajouté à une liste, tout comme les PDF récupérés au début de la boucle. Traiter tous les PDF à la fin du programme limite les problèmes, car le driver peut rencontrer des difficultés lorsque des onglets s'ouvrent pour les PDF, ce qui cause des bugs. Même ainsi, le téléchargement du premier PDF pose souvent problème, obligeant l'utilisateur à actualiser la page manuellement pour que le programme puisse reprendre son cours.

Indexation des documents : Cette section se charge d'indexer les documents extraits afin de faciliter leur recherche et leur traitement pour le modèle.

L'indexation des documents s'effectue dans la fonction `charge_index()`. Au lancement, elle vérifie la présence d'un index, s'il existe on utilise FAISS pour le récupérer, sinon on le créer à partir d'un dictionnaire d'URL (webpage_dict) qui contient toutes les URL à scraper ainsi qu'à partir d'un dossier « différents_textes » où on peut placer des documents supplémentaires à indexer. Une fois le scraping effectué et les chunks récupérés, on lance la fonction `load_new_documents()` pour transformer le contenu des documents du dossier « différents_textes » en chunks. On fusionne tous les chunks pour ensuite les vectoriser.
La vectorisation est effectuée avec la fonction `from_documents()` de FAISS. On lui passe les chunks et un modèle d'embedding, puis on enregistre les vecteurs avec la fonction `save_local()` :

```
vectorstore = FAISS.from_documents(
    documents=chunks_web + chunks_pdf, embedding=new_embeddings
)

vectorstore.save_local(new_index_path)
```

Initialisation de la réponse du modèle : Cette partie prépare et initialise les prompts nécessaires pour interagir avec le modèle.

Dès lors que l'utilisateur envoie un message, le programme appelle `setup_model()` pour préparer le Runnable qui permettra d'y répondre. Ce Runnable est composé du modèle utilisé pour la réponse, soit llama3:instruct, et du prompt :

```
prompt_exercice = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """Instruction: Répondre en francais à la question de l'utilisateur en te basant **uniquement** sur le contexte suivant fourni.
            Si tu ne trouves pas la réponse dans le contexte, demande à l'utilisateur d'être plus précis au lieu de deviner.
            Context:{context}""",
        ),
        MessagesPlaceholder(variable_name="history"),
        ("human", "Question: {question}"),
        ("ai", """Réponse:"""),
    ]
)
```

On retrouve la variable « context » passée dans le prompt qui correspond aux passages pertinents des documents indexés en fonction de la question de l'utilisateur.

Recherche de passages pertinents : Ce module analyse les documents indexés pour trouver les passages les plus pertinents en fonction de la question posée par l'utilisateur.

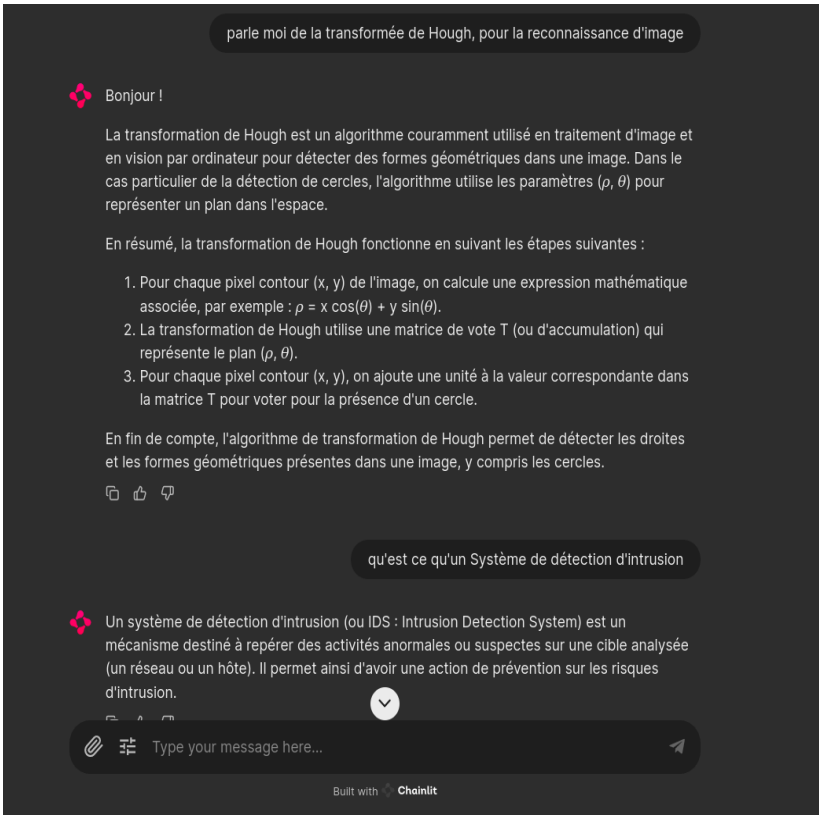
La fonction `trouve_contexte()` permet de chercher ces passages pertinents.

Tout d'abord, on construit le retriever qui est l'objet principal responsable de la récupération des documents pertinents. On récupère ce retriever avec l'index vectoriel « vectorstore » :

```
retriever = vectorstore.as_retriever()
```

Le retriever a une méthode principale appelée `similarity_search()` qui prend une requête (comme une question) et renvoie une liste de documents pertinents.
Le retriever est utilisé avec `similarity_search(question, k=10)`. Cela signifie qu'il trouve les dix documents (ou chunks de documents) les plus similaires à la question posée.
La similarité est mesurée dans l'espace vectoriel. Chaque document (ou chunk) est représenté par un vecteur d'embeddings, et la similarité est souvent calculée par la distance cosinus entre ces vecteurs.

Les premières versions de ce programme utilisaient une RetrievalQAChain pour trouver le contexte, mais cela a été changé au profit de la méthode actuelle, plus rapide.



Le modèle est capable de répondre à des questions sur les cours présents sur Moodle

Difficultés

Pour le web scraping, le contenu de beaucoup de pages est généré en javascript, ce qui nous oblige à utiliser une bibliothèque comme Selenium pour automatiser les interactions avec le navigateur et les pages web. Chaque page web a sa propre structure, ce qui rend l'automatisation complexe. En plus de ça, suivant quels types de fichiers sont téléchargés depuis les pages, le modèle peut avoir plus ou moins de mal à en tirer des informations.

Concernant l'indexation, si l'on souhaite ajouter un document, une page web, ou changer de modèle d'embedding, il est nécessaire de refaire une indexation complète des documents. Cette application est une forme aboutie de RAG, qui a nécessité une quantité conséquente de recherches et d'essais quant aux différentes technologies à utiliser.

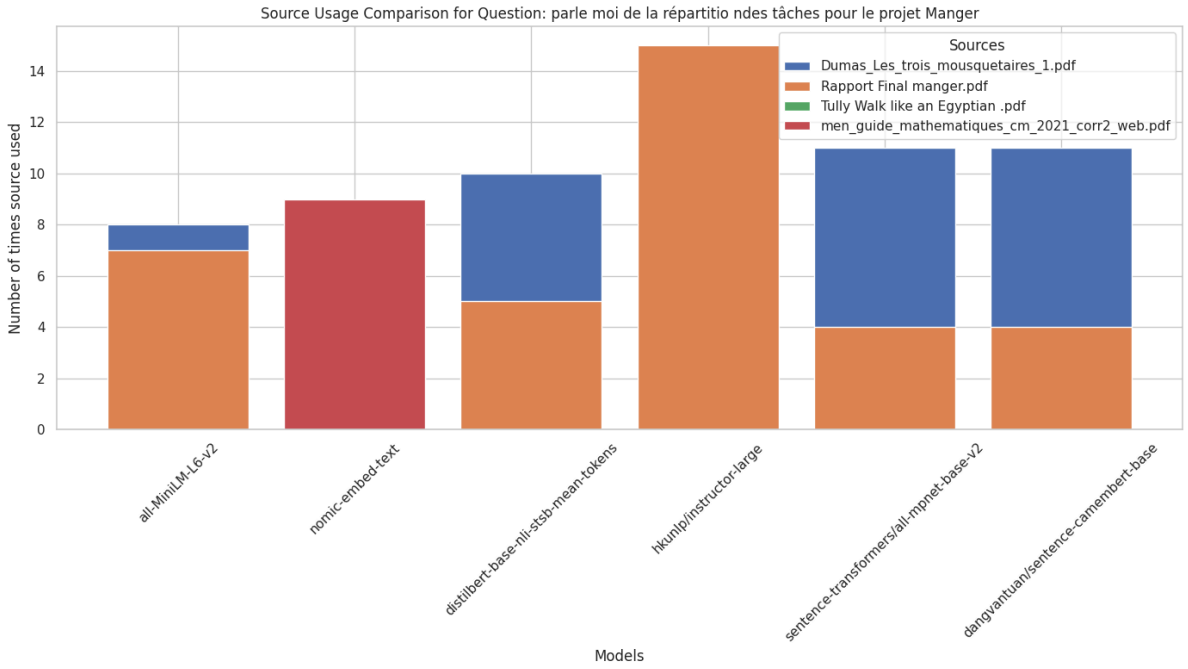
Nous avons commencé par explorer ElasticSearch pour gérer l'index, notamment via le système ElasticCloud. Malheureusement, le système était non seulement complexe

(requiert de bien gérer le deployment et les tokens allant avec), mais aussi payant au-delà de l'essai gratuit de 14 jours.

FAISS s'est révélé comme une meilleure alternative, de par sa simplicité d'utilisation, le fait qu'il puisse gérer plusieurs types d'index différents, et sa compatibilité avec LangChain et llama_index.

Une autre difficulté rencontrée lors de nos premiers RAG était le choix du modèle d'embedding. En effet, le modèle llama n'arrivait pas à générer des réponses satisfaisantes par rapport à la requête. Après avoir regardé plus précisément, nous nous sommes rendus compte que le problème venait du contexte récupéré. C'est là que notre manque d'expérience a été le plus regrettable, car ce n'est qu'après maintes manipulations (modification de la taille des chunks, limitation du nombre de source, puis de nombre de chunks autorisés par source, etc.) que nous nous sommes rendus compte que le problème ne venait pas du code en lui-même, mais directement du modèle d'embedding, "nomic-embed-text" jusque-là.

Après quoi, nous avons mis en place un programme pour comparer le contexte récupéré par différents modèles d'embedding à une question donnée, sandboxRAG/utils/compare_model_embedding.py. Ce modèle montre pour chacun des modèles utilisés dans combien et quels documents il va puiser le contexte nécessaire à la réponse. Après avoir comparé comme cela plusieurs modèles, dont un modèle spécifiquement pour le français, nous avons choisi "instructor-large" pour le programme.



Comparaison des modèles d'embedding utilisés

D’autres comparaisons sont disponibles dans le dossier sandboxRAG/screenshot_compare_embedding.

Résultats

L’application fonctionne et répond correctement aux requêtes portant sur le contenu des cours sur moodle , malgré le temps de réponse plutôt long (4 minutes). Les réponses du modèle sont cohérentes mais sont beaucoup influencées par la qualité d’indexation du modèle d’embedding. Le temps d’indexation est conséquent mais la vectorisation n'est normalement exécutée qu’une seule fois en amont.

Il est d’ailleurs possible de modifier le modèle d’embedding via les options directement dans l’interface chainlit. Quatre modèles sont proposés, qui sont les 4 meilleurs que nous avons essayés. Un autre modèle ayant l’air prometteur est le modèle sorti par OpenAI “text-embedding-ada-002”, malheureusement il nécessite une clé OpenAI.

Extension navigateur pour Zimbra

Objectifs

Cette extension de navigateur permet de générer une réponse d’un mail sur l’architecture mail Zimbra.

Implémentation

Cette extension est une adaptation d’une extension déjà existante utilisée pour la génération de réponse à des commentaires Twitter et Linkedin.

Pour générer une réponse, nous avons besoin de mail d’origine auquel il faut répondre, du modèle et un prompt spécifique pour faire comprendre au modèle qui s’agit d’un mail. Pour récupérer le mail auquel il faut répondre, on utilise simplement la fonction `queryselector()` de la bibliothèque Javascript.

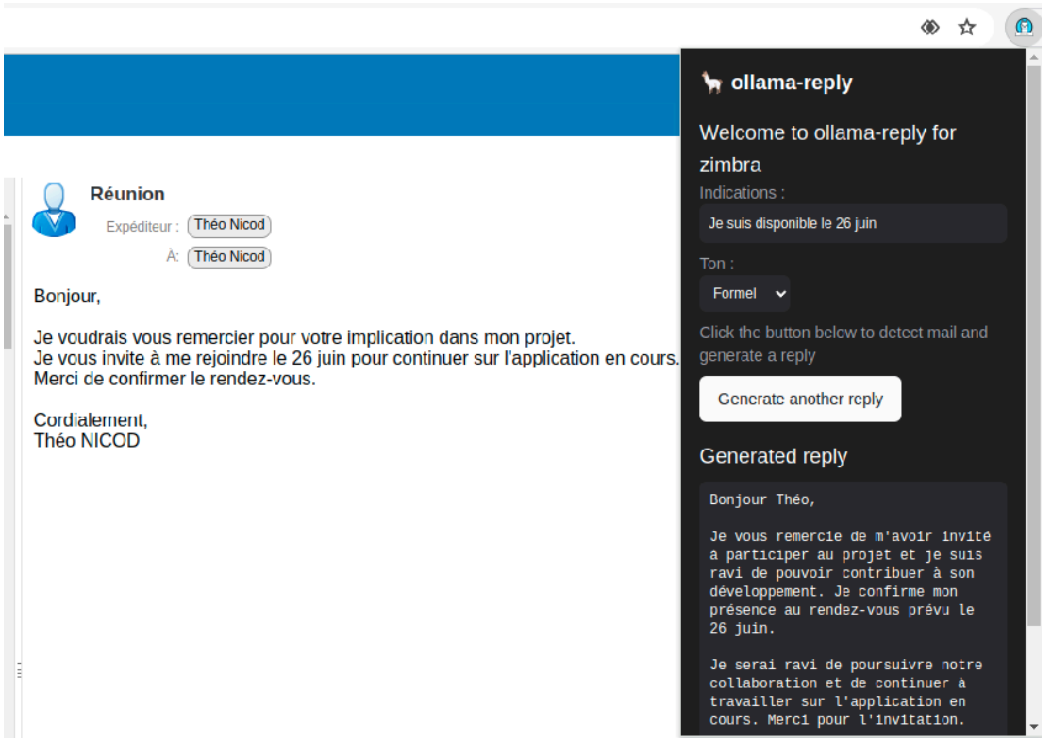
Le modèle est utilisé avec l’API d’Ollama. On utilise la route ollama :11434/api/chat pour envoyer notre prompt construit en amont et recevoir la réponse.

Le prompt est construit à l’aide de plusieurs éléments. L’utilisateur à la possibilité de renseigner des indications pour orienter la réponse du mail et également l’intonation de la réponse. Ces deux paramètres sont ajoutés au prompt. On donne une instruction spécifique au modèle pour lui faire comprendre qu’il s’agit d’une réponse de mail :

```
let r = `Vous êtes un assistant IA français. Vous devez répondre au e-mails sur un ton ${e.ton}. `;

if (e.indications) {
  r += `En suivant les indications.
  Indications :
  ${e.indications}`;
}
```

Résultats

L'extension fonctionne correctement et nous obtenons une réponse au bout de quelques dizaines de secondes.



La réponse générée est cohérente et correspond au format de mail. La réponse est générée dans l'extension mais pas dans la page ce qui pourrait être une prochaine amélioration, et utiliser l'API Zimbra pour générer la réponse directement dans la page.

Bilan

Les modèles de langage sont une technologie très récente, qui va continuer à gagner en popularité dans le futur. Mais ce côté nouveau ne vient pas sans ses inconvénients, tel le manque par moment de documentation. Pour des néophytes cherchant à s'approprier cette technologie, cela peut représenter un frein. La parfois rapide obsolescence des modules elle aussi peut dérouter.

Malgré ces défauts, les potentialités des modèles de langage, qui n'ont été qu'effleurées lors de ce projet, sont stimulantes, et présagent d'intéressantes éventualités pour le futur. En effet, parmi les idées d'application ayant été théorisées, un programme permettant de soumettre des images de copie d'étudiant pour les corriger avait été mentionné même si pas concrétisé, faute de données pour entraîner le modèle notamment.

Dans le cadre de l'université, des applications de ce genre peuvent apporter beaucoup. Sur une échelle plus grande, dans un monde qui utilise de plus en plus les technologies de modèle de langages, ce projet nous aura permis de nous familiariser avec ces possibilités et ces procédés. Il ne fait nul doute que ces connaissances nous seront très utiles pour le futur, et nous sommes reconnaissants d'avoir eu l'occasion de nous y préparer.

Sources

SimpleChat

https://github.com/ollama/ollama/blob/main/docs/tutorials/langchainpy.md

https://github.com/sudarshan-koirala/langchain-ollama-chainlit/blob/main/simple_chaiui.py

https://medium.com/@tahreemrasul/building-a-chatbot-application-with-chainlit-and-langchain-3e86da0099a6 (pour utiliser la ConversationBufferMemory)

sandboxRAG

Code

https://github.com/PrithivirajDamodaran/FlashRank

https://github.com/Coding-Crashkurse/Advanced-RAG

https://github.com/voyage-ai/voyage-large-2-instruct/tree/main

https://stackoverflow.com/questions/46849733/change-metadata-of-pdf-file-with-pypdf2
(pour corriger les metadata des fichiers pdf)

https://www.reddit.com/r/LangChain/comments/1ba77pu/difference_between_as_retriever_and_similarity/

ollama/ollama#3938 (langchain)

https://github.com/AllAboutAI-YT/easy-local-rag (méthode tierce)

https://github.com/vndee/local-rag-example (langchain)

https://github.com/ollama/ollama/blob/main/docs/tutorials/langchainpy.md (langchain)

https://api.python.langchain.com/en/latest/vectorstores/langchain_community.vectorstores.faiss.FAISS.html#langchain_community.vectorstores.faiss.FAISS.similarity_search (type de similarity_search)

https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index (les types d'index FAISS)

Articles

https://towardsdatascience.com/openai-vs-open-source-multilingual-embedding-models-e5cb7c90f05

https://medium.com/@fabio.matricardi/metadata-metamorphosis-from-plain-data-to-enhanced-insights-with-retrieval-augmented-generation-8d1a8d5a6061 (metadata)

https://docs.llamaindex.ai/en/stable/examples/prompts/prompts_rag/ (pour gérer des prompts avec llama_index)

https://www.pinecone.io/learn/openai-embeddings-v3/

https://datacorner.fr/spacy/ (ranking)

https://www.pinecone.io/learn/chunking-strategies/ (chunking techniques)

https://huggingface.co/spaces/mteb/leaderboard (comparaison modèles embedding)

https://atlas.nomic.ai/map/nomic-text-embed-v1-5m-sample

https://platform.openai.com/docs/guides/embeddings

https://www.reddit.com/r/LangChain/comments/186sgyf/rag_filtering_docs_to_only_send_relevant_data_to/

https://blog.devgenius.io/automated-translation-of-text-and-data-in-python-with-deep-translator-d980afee70ab (traduction)

https://iamajithkumar.medium.com/working-with-faiss-for-similarity-search-59b197690f6c

https://medium.com/@akriti.upadhyay/implementing-rag-with-langchain-and-hugging-face-28e3ea66c5f7

https://medium.com/@varsha.rainer/document-loaders-in-langchain-7c2db9851123

https://blog.gopenai.com/creating-rag-app-with-llama2-and-chainlit-a-step-by-step-guide-d98499c2cd89

https://blog.gopenai.com/building-a-rag-chatbot-using-llamaindex-groq-with-llama3-chainlit-b1709f770f55

https://aws.amazon.com/fr/what-is/retrieval-augmented-generation/

https://hackernoon.com/fr/un-tutoriel-sur-la-fa%C3%A7on-de-cr%C3%A9er-votre-propre-chiffon-et-de-l%27ex%C3%A9cuter-localement-langchain-ollama-streamlit

PR_Reviewer

https://blog.gopenai.com/building-a-rag-chatbot-using-llamaindex-groq-with-llama3-chainlit-b1709f770f55 (chainlit avec llama index)

https://ogre51.medium.com/context-window-of-language-models-a530ffa49989 (context window)

https://docs.chainlit.io/integrations/llama-index (documentation chainlit)

https://docs.llamaindex.ai/en/stable/examples/prompts/prompts_rag/ (prompt llama_index)

https://docs.llamaindex.ai/en/stable/examples/data_connectors/GithubRepositoryReaderDemo/ (doc officielle)

https://github.com/joaomdmoura/crewAI https://docs.crewai.com/tools/GitHubSearchTool (impossible d'importer crewAI)

https://lightning.ai/lightning-ai/studios/chat-with-your-code-using-rag (a donné des pistes)

https://docs.llamaindex.ai/en/latest/module_guides/indexing/vector_store_index/ (pour utiliser le VectorStoreIndex avec llama_index)

sandboxFinetuning

https://dassum.medium.com/fine-tune-large-language-model-llm-on-a-custom-dataset-with-qlora-fb60abdeba07