Rapport d’analyse des tests

Introduction

L’objectif était d’identifier la combinaison optimale de paramètres de traitement de texte - notamment Text Splitter, Vectorstore, Chunk\_size et Chunk\_overlap - pour maximiser le f1\_score, un indicateur de performance basé sur la précision et le rappel des Bert\_Score. Nous cherchions également à comprendre si cette optimisation variait en fonction de la nature du texte, en l'occurrence entre les cours académiques, de structure plus aléatoire, et les articles scientifiques, généralement plus structurés.

**Méthodologie**

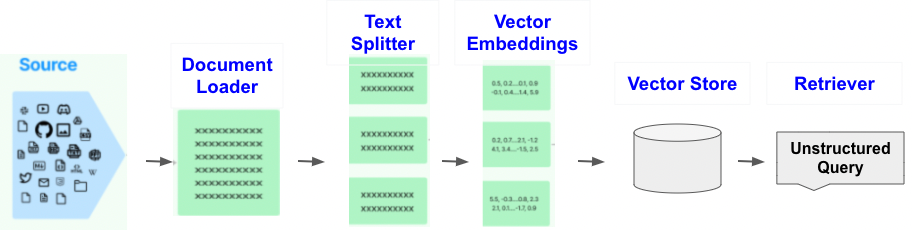
1. **Processus de Traitement de Texte**

Schéma de Traitement de Données pour la Récupération d'Information Textuelle Avant l'Utilisation d'un Modèle de Langage (LLM) [1]

Le ou les documents à interroger doivent subir certains traitements, décrits sur le schéma ci-dessus avant son utilisation par un modèle de langage. C’est une étape cruciale et indispensable pour augmenter la pertinence des réponses et ce n’est qu’après ce traitement que le choix du LLM peut être ajusté.

* **Document Loader** : Cette étape consiste à charger les données textuelles depuis une source. Dans notre code, le chargement inclue la prise en charge des pdf avec le package pypdf et le nettoyage (notamment les caractères spéciaux).
* **Text Splitter** [2]: Après le chargement, le texte est divisé en segments plus petits, appelés "chunks". Cette découpe en segments permet de gérer plus facilement de grands volumes de texte et d'améliorer la gestion de la mémoire. Les paramètres importants ici sont :
  + **chunk\_size** : qui définit la longueur de chaque segment. Une taille optimale est cruciale pour assurer une bonne cohérence du contenu sans perdre le contexte.
  + **chunk\_overlap** : qui permet à chaque segment de chevaucher légèrement le suivant. Ceci garantit qu'aucune information contextuelle importante n'est perdue entre les segments.
* **Vector Embeddings** : Chaque segment de texte est ensuite converti en un vecteur numérique, ce qui crée une représentation mathématique du texte. Cela permet au modèle de comprendre et de comparer sémantiquement les segments de texte.
* **Vector Store** : Les vecteurs sont stockés dans une base de données, souvent appelée "vector store". Cette base de données est conçue pour permettre des recherches rapides et efficaces parmi les vecteurs.
* **Retriever** : Enfin, lorsqu'une requête non structurée est soumise au modèle, le "retriever" recherche dans le "vector store" pour trouver les vecteurs les plus pertinents (et donc les segments de texte correspondants) pour répondre à la requête. Cette étape est essentielle pour sélectionner les informations les plus pertinentes à utiliser pour générer une réponse.

Nous avons choisi d’utiliser LangChain pour chacune des étapes pour faciliter largement le processus [3].

1. **Choix des paramètres**

Pour chaque sous-étape, il existe une multitude de méthodes. Il fallait donc en choisir certaines et en tester d’autres.

* **Text Splitter [1] :** Il existe des méthodes de split selon le code de programmation, HTML ou Markdown. Celles-ci ne sont pas nécessaires pour notre projet car nous traitons seulement le langage naturel. La segmentation par caractère est écartée également car nous avons besoin de comprendre le contexte complet et les mots entiers pour générer des réponses pertinentes et précises. La segmentation "semantic chunker" est prometteuse car elle regroupe le texte par similarité de sens, mais elle reste expérimentale et moins fiable pour une utilisation immédiate. Nous avons donc choisi de tester les méthodes « **token** » et « **recursive** » pour leur efficacité reconnue et l’utilisation facilitée par LangChain. La segmentation par "token" divise le texte en mots ou phrases, conservant le contexte vital pour l'interprétation sémantique, tandis que la segmentation "recursive" décompose récursivement le texte en sous-unités, préservant la structure hiérarchique.
* **Chunk\_Size et Chunk\_Overlap** [4]: Ici l’objectif est de trouver un équilibre entre la préservation du contexte et garder une certaine précision. Nous avons choisi de tester une large gamme d’ordre de grandeur : des tailles de **10, 50, 100, 500, 1000** et des overlaps **de 0,10,50, 100 et 1000**.
* **Embedding** [5]**:** Ici encore il existe de nombreuses méthodes. Celle que nous avons sélectionnée est BERT (transformer), capable de comprendre le texte et les mots en fonction du contexte (lecture bidirectionnelle). D'autres types d'embedding tels que Word2Vec [6] et TF-IDF ont été écartés pour des raisons spécifiques à notre projet, comme leur incapacité à comprendre le contexte ou leur dépendance à la fréquence d'apparition des mots. Une explication détaillée de certains d’entre eux est disponible ici [7]. Le modèle embedding d’OpenAI est aussi performant mais l’utilisation gratuite est limitée au nombre d'essais et on a besoin de faire tourner les tests de nombreuses fois. Le modèle embedding de Bert open-source utilisé disponible sur HuggingFace a été ‘sentence-transformer/all-mpnet-base-v2’. C’est le transformer le plus utilisé sur HuggingFace et le plus performant.
* **Vector Store** [8] : Dans cet article, l’efficacité de plusieurs méthodes sont comparées notamment Faiss, ChromaDB, Qdrant et PgVector. Finalement, nous avons décidé de tester les deux les plus précis (bien que plus longs) : Qdrant et ChromaDB car nous voulons privilégier la pertinence des réponses au temps de réponse. Ces derniers sont également facilement utilisables avec LangChain [9].

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

Comparaison de la performance de quatre méthode « Vector Store » en termes de temps de recherche et de précision[8]

* **Retriever**[10]**:** La méthode choisie a été celle basée sur vectorstore. Cette dernière utilise une représentation vectorielle des documents pour effectuer la récupération et capturer leur sens sémantique. Il existe une autre méthode classique de récupération (retrieval) qui utilise une recherche par mots-clés. Elle se concentre sur la correspondance exacte des termes. C’est une méthode simple mais qui peut ne pas capturer la signification sémantique complète, c’est pourquoi elle a été écartée de notre projet.

Cependant, il existe une méthode appelée hybride sur laquelle Microsoft a publié un article [11] qui paraît intéressante. La méthode hybride combine ces deux approches et utilise le Fusionnement de Rangs Réciproques (Reciprocal Rank Fusion - RRF) pour intégrer les résultats, offrant ainsi une recherche plus robuste et précise en tirant parti des avantages des deux méthodes. Pour aller plus loin, il aurait été intéressant de la tester.

1. **Méthodologie des Tests :**

Nous avons testé quatre fichiers divisés en deux catégories : deux cours académiques et deux articles scientifiques. Chaque fichier a été évalué sur cinq requêtes distinctes - trois résumés de longueurs différentes (court-moyen-long) et deux questions sur le contenu (facile-difficile). Nous avons testé chaque fichier avec toutes les combinaisons possibles (60) de paramètres que nous avons choisi, afin de couvrir un large spectre de configurations possibles. Nous avons donc obtenu au total 20 fichiers csv (un fichier = une requête) de 60 tests de combinaisons (une ligne = un test).

Le modèle mistralai/Mistral-7B-v0.1 a été sélectionné pour ses performances élevées et sa disponibilité en tant que modèle open-source, disponible sur HuggingFace, ce qui le rend idéal pour notre projet. L’importance ici est d’utiliser toujours le même pour évaluer chaque test de la même façon. Le Prompt était constant aussi et nous l’avons choisi de manière à être pertinent par rapport à nos besoins : répondre à la question, en français et en utilisant seulement le contenu présent dans le document et préciser de répondre qu’il ne sait pas si la réponse n’était pas présente dans le document.

L’objectif était de comparer les réponses générées par le LLM avec des réponses préparées manuellement pour évaluer leur pertinence et précision. Pour faciliter la préparation de ces dernières, des cours connus, propres à notre formation ont été utilisés ainsi que des articles spécifiques au domaine appliqué (biologie, bio-informatique). Les mots clés et les informations principales de chaque document ont été vérifiés avec ChatGPT pour ne rien oublier et améliorer la pertinence de nos réponses et résumés.

Pour comparer chacune des sorties, une évaluation à l’aide des Bert\_Scores [12] a été utilisée, permettant d'évaluer objectivement la qualité des réponses. De plus, cette méthode présente l'avantage de considérer le contexte sémantique, plutôt que de se limiter à la simple fréquence d'apparition des mots-clés. Trois scores sont donc attribués : la précision (mesure de la qualité des mots générés par rapport aux mots de référence), le rappel (évalue à quel point les mots de référence sont bien capturés par les mots générés), et le F1 (moyenne de la Précision et du Rappel). Un score de 1 serait parfait, tandis qu'un score de 0 indiquerait une performance très faible.

**Résultats**

Comme nous nous intéressons à la combinaison de paramètres la plus robuste et fiable à travers une variété de contextes, la méthode utilisée pour l’analyse des résultats est celle de calculer la moyenne des f1\_scores entre tous les tests pour chaque combinaison.

Cette approche présente l'avantage de permettre la comparaison de chaque combinaison dans tous les contextes, contrairement à celle qui se baserait uniquement sur les combinaisons obtenant le meilleur résultat le plus fréquemment.

Cependant, il faut noter que l’inconvénient de cette méthode est qu’elle est sensible aux valeurs particulièrement basses ou hautes, mais dans notre cas, peu d’entre elles ont été remarquées. Il faudrait réaliser plusieurs fois chacun des tests pour une meilleure fiabilité.

D’abord, la démarche a été réalisée en différenciant les catégories (cours, article, résumés, questions).

### Cours académiques

Les meilleures combinaisons de paramètres pour les cours sont :

1. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 100, Chunk Overlap: 50 — Mean F1 Score: 0.5333
2. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 500, Chunk Overlap: 10 — Mean F1 Score: 0.4861
3. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 100, Chunk Overlap: 10 — Mean F1 Score: 0.4749

### Articles

Les meilleures combinaisons de paramètres pour les articles sont :

1. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 50, Chunk Overlap: 0 — Mean F1 Score: 0.5105
2. Splitter: Recursive, Vectorstore: Chroma, Chunk Size: 1000, Chunk Overlap: 500 — Mean F1 Score: 0.4948
3. Splitter: Token, Vectorstore: Qdrant, Chunk Size: 50, Chunk Overlap: 0 — Mean F1 Score: 0.4775

### Résumés

Les meilleures combinaisons de paramètres pour les résumés sont :

1. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 100, Chunk Overlap: 50 — Mean F1 Score: 0.5333
2. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 50, Chunk Overlap: 0 — Mean F1 Score: 0.5105
3. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 500, Chunk Overlap: 10 — Mean F1 Score: 0.4963

### Questions

Les meilleures combinaisons de paramètres pour les questions sont :

1. Splitter: Recursive, Vectorstore: Chroma, Chunk Size: 1000, Chunk Overlap: 50 — Mean F1 Score: 0.4935
2. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 50, Chunk Overlap: 10 — Mean F1 Score: 0.4722
3. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 100, Chunk Overlap: 0 — Mean F1 Score: 0.4707

De manière rigoureuse, il faudrait utiliser des tests statistiques pour déterminer si la différence des combinaisons de paramètres est significative ou non et effectuer d’autres tests pour augmenter le jeu de données. Ici, on remarque que l’utilisation de Recursive et Qdrant est systématiquement performante sur tous les types de textes. Il est plus difficile de ressortir une tendance pour Chunk\_size et Chunk\_overlap mais à la vue des résultats on pourra considérer qu’il n’y a pas de différence significative entre les meilleures combinaisons de paramètres pour les articles et pour les cours, de même pour les questions et pour les résumés. Nous considérons donc que les conclusions suivantes sont valables dans tous les cas. Des tests supplémentaires seraient nécessaires pour confirmer cette hypothèse et approfondir l’analyse.

Le même raisonnement est donc effectué sans séparer les catégories et les cinq meilleures combinaisons de paramètres sont :

1. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 100, Chunk Overlap: 50 — Mean F1 Score: 0.5333
2. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 50, Chunk Overlap: 0 — Mean F1 Score: 0.5105
3. Splitter: Recursive, Vectorstore: Qdrant, Chunk Size: 500, Chunk Overlap: 10 — Mean F1 Score: 0.4861
4. Splitter: Token, Vectorstore: Qdrant, Chunk Size: 50, Chunk Overlap: 0 — Mean F1 Score: 0.4775
5. Splitter: Recursive, Vectorstore: Chroma, Chunk Size: 50, Chunk Overlap: 0 — Mean F1 Score: 0.4718

Celles-ci sont donc les plus fiables pour toutes les requêtes. Le fichier "conclusion\_results.csv" répertorie les meilleurs f1\_scores obtenus pour chaque test. Ces scores ont été sélectionnés manuellement en prenant en considération leur distinction par rapport aux autres. L'objectif de cette étape est de déterminer quelles combinaisons de paramètres, parmi celles que nous avons identifiées comme les plus fiables précédemment, se répètent le plus souvent. Cela nous permettra de déterminer quelles combinaisons sont les plus performantes en termes de précision.

Finalement, on trouve que la combinaison avec Splitter : Recursive, Vectorstore : Qdrant, Chunk Size : 100, et Chunk Overlap : 50 ainsi que la combinaison avec Splitter : Recursive, Vectorstore : Qdrant, Chunk Size : 500, et Chunk Overlap : 10 sont celles qui apparaissent le plus fréquemment parmi les meilleures performances, chacune apparaissant 3 fois.

**Conclusion et Limites**

Avec cette méthode nous avons conclu sur deux combinaisons de paramètres qui semblent fiables et robustes : Recursive-Qdrant-Chunk\_Size 100-Chunk\_Overlap 50 et Recursive-Qdrant-Chunk\_Size 500-Chunk\_Overlap 10. Dans la plupart des cas, les résultats les plus satisfaisants ont été obtenus en utilisant une combinaison avec "Qdrant" et "Recursive". En ce qui concerne les paramètres "Chunk\_Size" et "Chunk\_Overlap", il est plus difficile de tirer une conclusion définitive car nous n'avons pas observé de tendance claire en fonction de la taille ou du type de document. Il serait donc judicieux de réaliser davantage de tests pour évaluer ces deux derniers paramètres de manière plus approfondie. Il est également essentiel de garder à l'esprit que l'optimisation du prétraitement des données est une tâche spécifique à chaque document, ce qui signifie qu'il serait idéal d'adapter notre approche à chaque nouveau document pour obtenir les meilleurs résultats possibles.

Il est important de noter que la performance de nos tests a montré quelques particularités :

* Les réponses générées par le LLM n'ont pas toujours respecté la demande de taille, produisant généralement des paragraphes d'environ 65 mots avec des phrases tronquées à la fin. Il serait utile d'explorer d'autres modèles de langage naturel et de modifier le prompt pour obtenir des réponses plus conformes à nos attentes.
* Nous avons rencontré des réponses vides, notamment pour les deux premiers fichiers, en particulier lorsqu'on demandait des résumés. Cependant, ce problème ne s'est pas reproduit pour les deux derniers fichiers on suppose que le problème ne vient pas de l’algorithme, et suggère potentiellement une structure spécifique de l’article en question.
* Les réponses complètement hors sujet du contenu du fichier pouvaient quand même obtenir un score de Bert de 0.2 à 0.35 environ, tandis que les réponses pertinentes, contenant les mots-clés principaux, avaient des scores à partir de 0.55. Cependant, nous n'avons pas obtenu de scores supérieurs à 0.75.
* Malgré nos instructions de répondre en français dans le prompt, les réponses étaient fournies en anglais. (Cette méthode avait fonctionné pour des tests indépendants). Le deuxième cours académique fourni était en français, les autres en anglais. Dans tous les cas le fichier a été traité et les réponses fournies en anglais.
* De même, le modèle répond aux questions même si l’information n’est pas contenue dans le document (un seul test réalisé : test\_5).

**Bibliographie**

[1] « Text Splitters | 🦜️🔗 Langchain ». Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://python.langchain.com/docs/modules/data\_connection/document\_transformers/

[2] V. Sulgante, « Splitting large documents | Text Splitters | Langchain », Medium. Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://medium.com/@cronozzz.rocks/splitting-large-documents-text-splitters-langchain-7c7bfa899267

[3] O. Mishra, « Using langchain for Question Answering on own data », Medium. Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://medium.com/@onkarmishra/using-langchain-for-question-answering-on-own-data-3af0a82789ed

[4] R. Schwaber-Cohen, « Chunking Strategies for LLM Applications | Pinecone ». Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://www.pinecone.io/learn/chunking-strategies/

[5] P. Srivatsavaya, « Different Types of Embeddings », Medium. Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://medium.com/@prudhviraju.srivatsavaya/different-types-of-embeddings-1693dad922be

[6] « Gensim: topic modelling for humans ». Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://radimrehurek.com/gensim/models/word2vec.html

[7] « 5 Types of Word Embeddings and Example NLP Applications », Swimm. Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://swimm.io/learn/large-language-models/5-types-of-word-embeddings-and-example-nlp-applications

[8] J. Cao, « How to pick a Vector Database — Quantitative Analysis », EvaDB Blog. Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://medium.com/evadb-blog/how-to-pick-a-vector-database-quantitative-analysis-afae5ea9e5b1

[9] « Vector stores | 🦜️🔗 Langchain ». Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://python.langchain.com/docs/modules/data\_connection/vectorstores/

[10] « Vector store-backed retriever | 🦜️🔗 Langchain ». Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://python.langchain.com/docs/modules/data\_connection/retrievers/vectorstore

[11] « Azure AI Search: Outperforming vector search with hybrid retrieval and ranking capabilities », TECHCOMMUNITY.MICROSOFT.COM. Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://techcommunity.microsoft.com/t5/ai-azure-ai-services-blog/azure-ai-search-outperforming-vector-search-with-hybrid/ba-p/3929167

[12] H. Özbolat, « Text Summarization: How to Calculate BertScore », Medium. Consulté le: 31 janvier 2024. [En ligne]. Disponible sur: https://haticeozbolat17.medium.com/text-summarization-how-to-calculate-bertscore-771a51022964